MuJoCo Playground

Kevin Zakka[*,2], **Baruch Tabanpour**[*,1], **Qiayuan Liao**[*,2], **Mustafa Haiderbhai**[*,4], **Samuel Holt**[*,3], **Jing Yuan Luo**[1], **Arthur Allshire**[2], **Erik Frey**[1], **Koushil Sreenath**[2], **Lueder A. Kahrs**[4], **Carmelo Sferrazza**[†,2], **Yuval Tassa**[†,1] **and Pieter Abbeel**[†,2]
[1]Google DeepMind, [2]UC Berkeley, [3]University of Cambridge, [4]University of Toronto, [*]Equal contributions, [†]Equal advising

We introduce MuJoCo Playground, a fully open-source framework for robot learning built with MJX, with the express goal of streamlining simulation, training, and sim-to-real transfer onto robots. With a simple `pip install playground`, researchers can train policies in minutes on a single GPU. Playground supports diverse robotic platforms, including quadrupeds, humanoids, dexterous hands, and robotic arms, enabling zero-shot sim-to-real transfer from both state and pixel inputs. This is achieved through an integrated stack comprising a physics engine, batch renderer, and training environments. Along with video results, the entire framework is freely available at `playground.mujoco.org`.

1. Introduction

Reinforcement learning (RL) [28] with subsequent transfer to hardware (sim-to-real) [69], is emerging as a leading paradigm in modern robotics [27, 31, 40]. The benefits of simulation are obvious – safety and cheap data. The recipe involves four steps:

1. Create a simulated environment that matches the real world.
2. Encode desired robot behavior with a reward function.
3. Train a policy in simulation.
4. Deploy to the robot.

The key enabler of this approach is a simulator that is realistic, convenient, and fast.

The realism requirement is self-evident, the "digital twin" of step 1 demands a minimal level of fidelity [69]. Convenience and usability are equally critical, streamlining the creation, modification, composition, and characterization (system identification) of simulated robots.

The importance of speed is less obvious – why does it matter if training takes ten minutes or ten hours? The answer lies in reward design (step 2), which cannot be easily automated: what the robot *ought* to do is an expression of human preference. Even if reward design is semi-automated [37], the process remains iterative: RL excels at finding policies that obtain reward, but the resulting behavior is often irregular in unexpected ways. Since steps 2 and 3 (and occasionally step 4) must be repeated [8], *time-to-robot* becomes critical: the time from when you ask the robot to do something until you see what it thinks you meant.

RL is computationally intensive, requiring an enormous number of agent-environment interactions to train effective policies [25]. GPU-based simulation can significantly accelerate this process for two key reasons. First, the median GPU is far more powerful than the median CPU [65], and while high core-count CPUs exist, they are uncommon. Second, by keeping the entire agent-environment loop on device, we can harness the high-throughput, highly parallel architecture [14, 39]. This is especially true for *on-policy* RL [4, 51], which employs GPU-friendly, wide-batch operations. Locomotion and manipulation tasks which previously required days of training on multi-host setups [5, 59], can now be solved within minutes or hours on a single GPU [20, 50].

MuJoCo [63] is an open-source simulator publicly developed and maintained by Google DeepMind. Designed to support complex, high-fidelity simulation, it provides a rich model-definition language and model-editing APIs, which are well-documented and conveniently exposed. The simplicity and self-consistency of MuJoCo's data-structures and pipeline make it particularly well-suited for transcription to parallel compute frameworks. We build upon MuJoCo XLA [43] (MJX), a JAX-based branch of MuJoCo that runs on GPU, enabling training directly on device.

Corresponding author(s): btaba@google.com, zakka@berkeley.edu

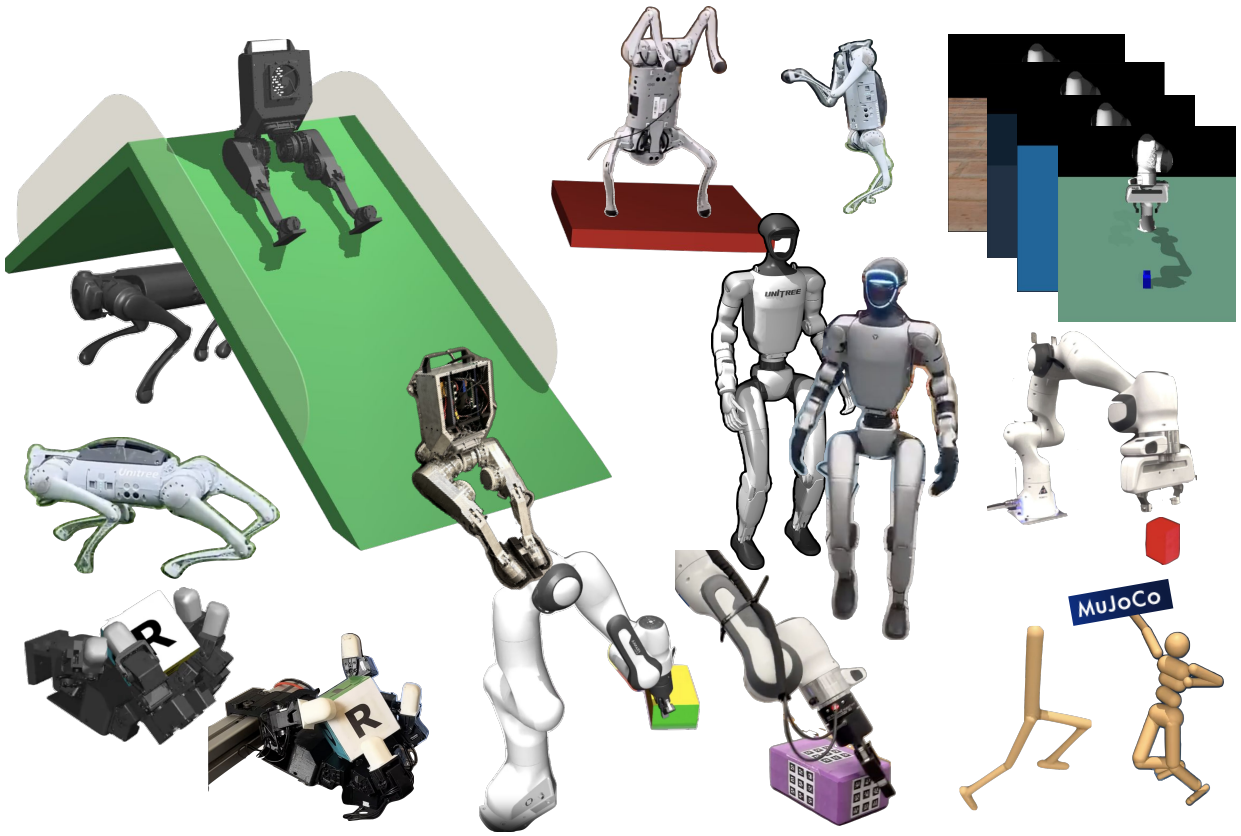

Figure 1 | A cartoon of MuJoCo Playground's diverse environments that were successfully transferred to real hardware, including Berkeley Humanoid, Unitree Go1 and G1, LEAP hand and Franka Arm.

We introduce MuJoCo Playground, a fully open-source framework for robot learning designed for rapid iteration and deployment of sim-to-real reinforcement learning policies. Besides physics and learning, we incorporate on-device rendering through the Madrona batch renderer [53], facilitating training of vision-based policies. With a straightforward installation process (`pip install playground`) and cross-platform support, users can quickly train policies on a single GPU. The entire pipeline—from environment setup to policy optimization—can be executed in a single Colab notebook, with most tasks requiring only minutes of training time.

MuJoCo Playground's lightweight implementation greatly simplifies sim-to-real deployment, transforming it into an interactive process where users can quickly tweak parameters to refine robot behavior. In our experiments, we deployed both state- and vision-based policies across six robotic platforms in less than eight weeks. We hope that MuJoCo Playground becomes a valuable resource for the robotics community and expect it to continue building on MuJoCo's thriving open-source ecosystem.

Our work makes three main contributions:

1. We develop a comprehensive suite of robotic environments using MJX [43], demonstrating sim-to-real transfer across diverse platforms including quadrupeds, humanoids, dexterous hands, and robot arms.
2. We integrate the open-source Madrona batch GPU renderer [53] to enable end-to-end vision-based policy training on a single GPU device, achieving zero-shot transfer on manipulation tasks.
3. We provide a complete, reproducible training pipeline with notebooks, hyperparameters, and training curves, enabling rapid iteration between simulation and real-world deployment.

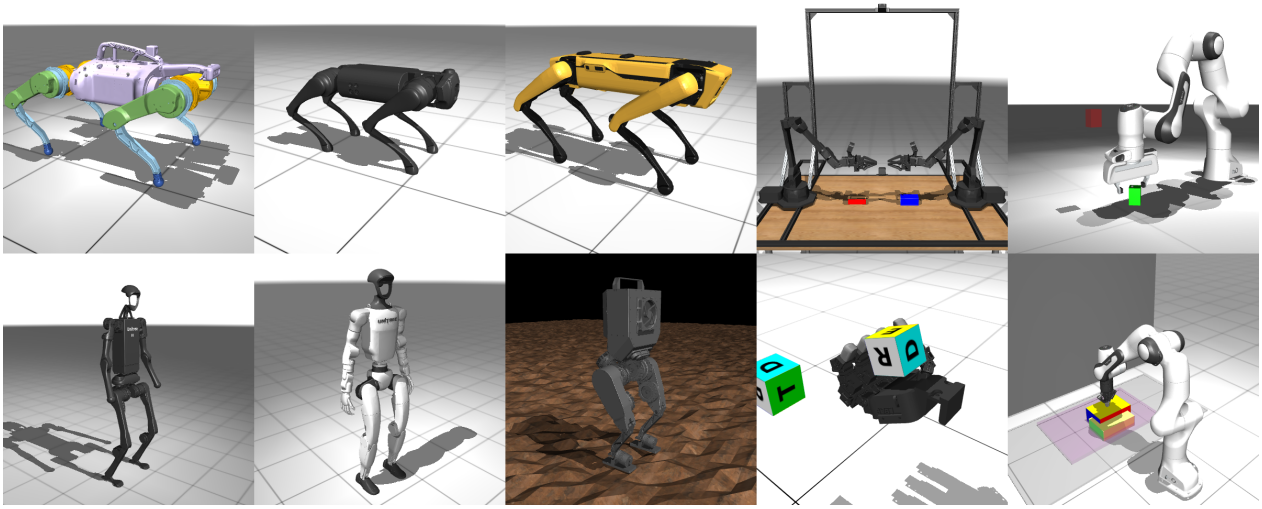

Figure 2 | A preview of locomotion and manipulation environments available in MuJoCo Playground.

2. Environments

MuJoCo Playground contains environments in 3 main categories: DeepMind (DM) Control Suite, Locomotion, and Manipulation, which we briefly describe in this section. Locomotion and manipulation environments are tailored to robotic use-cases and we show zero-shot sim-to-real transfer in many of the available environments. Playground directly utilizes MuJoCo Menagerie [68] which offers a suite of robot assets and configurations tailored to run in MuJoCo.

2.1. DM Control Suite

The majority of RL environments from [61] are re-implemented in MJX, and serve as entry-level tasks to familiarize users with MuJoCo Playground (Figure 3).

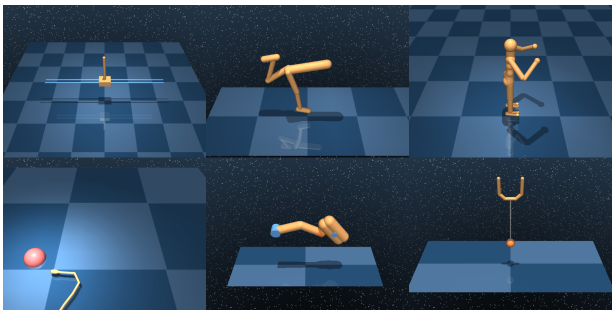

Figure 3 | Several DM Control Suite environments.

2.2. Locomotion

Locomotion environments in MuJoCo Playground are implemented for multiple quadrupeds and bipeds (Figure 2 left). The quadrupeds include the Unitree Go1, Boston Dynamics Spot, and Google Barkour [7], while the humanoids include the Berkeley Humanoid [35], Unitree H1 and G1, and the Robotis OP3. For each robot embodiment, we implement a joystick environment that learns to track a velocity command consisting of base linear velocities in both the forward and lateral directions, as well as a desired yaw rate. On the Unitree Go1, we additionally implement fall recovery and handstand environments. A complete list of locomotion environments is provided in Table 5 in the appendix.

We demonstrate sim-to-real transfer in two main sets of experiments. First, on the Unitree Go1, we deploy joystick, fall recovery, and handstand policies. Second, we demonstrate joystick-based locomotion on the Berkeley Humanoid and the Unitree G1. More details on these sim-to-real experiments can be found in Section 4.2.

2.3. Manipulation

Manipulation environments in MuJoCo Playground are implemented for both prehensile and non-prehensile tasks (Figure 2 right). With the Leap Hand [56] robot, we demonstrate contact-rich dexterous re-orientation of a block. Using

the Franka Emika Panda and Robotiq gripper, we show re-orientation of a yoga block using high frequency torque control. We implement a simple vision-based pick-cube environment on a Franka arm using the Madrona batch renderer. A few additional environments, such as bi-arm peg-insertion with the Aloha robot [3], are also available. We refer to Table 8 in the appendix for a full set of environments.

We demonstrate sim-to-real transfer on the Leap Hand and Franka arm robots, including an environment trained from vision for the pick-cube task. More details on the sim-to-real experiments are available in Section 4.3.

3. Batch Rendering with Madrona

MuJoCo Playground enables vision-based environments through an integration of MJX with Madrona [54]. Madrona is a GPU-based entity-component-system (ECS), which contains GPU implementations of high throughput rendering [49]. Madrona provides two rendering backends: a software-based batch ray tracer written in CUDA (used for the experiments in this work) and a Vulkan-based rasterizer. The raytracing backend supports features including complex lighting scenarios, shadows, textures, and geometry materials. See Figure 4 for examples of rendered images using the batch ray tracer. Some features such as deformable materials, moving lights, and terrain height fields will be added in the future.

The Madrona Batch Renderer is integrated with MJX through low-level JAX [6] primitives that connect to the initialization and render functions exposed by Madrona. These JAX primitives allow for Madrona to interact seamlessly with JAX transformations such as *jit* and *vmap*. Mujoco Playground provides two examples: (`cartpole-balance` and `PandaPickCubeCartesian`) to showcase the implementation of vision-based environments and training of vision-based policies.

The Madrona MJX integration also supports customization of each environment instance, allowing for domain randomization [62] of visual properties such as geometry size, color, lighting conditions, and camera pose. These randomizations play a crucial role in the sim-to-real transfer of vision-based policies, which we discuss more in Section 4.3.3.

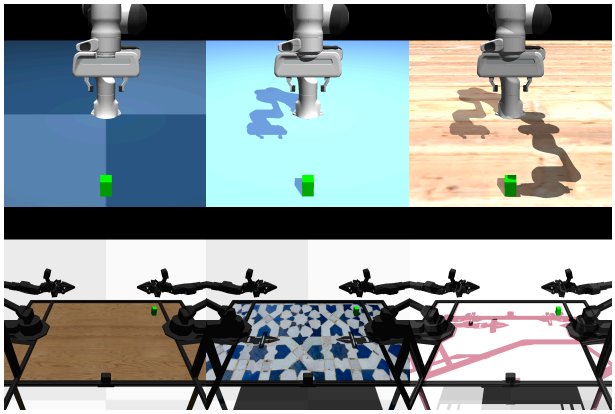

Figure 4 | Sample renders from the Madrona batch renderer for the Panda and Aloha environments. Left-most images are the original environments. The remaining images highlight the the support for lighting, shadows, textures, and colors, including the ability to domain randomize these parameters during training.

4. Results

In this section, we report RL and sim-to-real results for environments in MuJoCo Playground. Sim-to-real experiments (see some examples in Figure 5) are performed for locomotion and manipulation environments from both proprioceptive state and from vision. We briefly discuss RL training on different hardware devices and RL libraries.

4.1. DM Control Suite

We train state-based policies for all available tasks, with most environments training in under 10 minutes on a single GPU device. More details on the training process can be found in Appendix B.2. All available environments in the MJX port of the DM Control Suite, including any modifications, are detailed in Appendix B.1.

Using the batch renderer, we also implement pixel-based observations for the CartpoleBalance environment. These observations are generated on the GPU, allowing us to keep physics, rendering, and training entirely on-device. Although

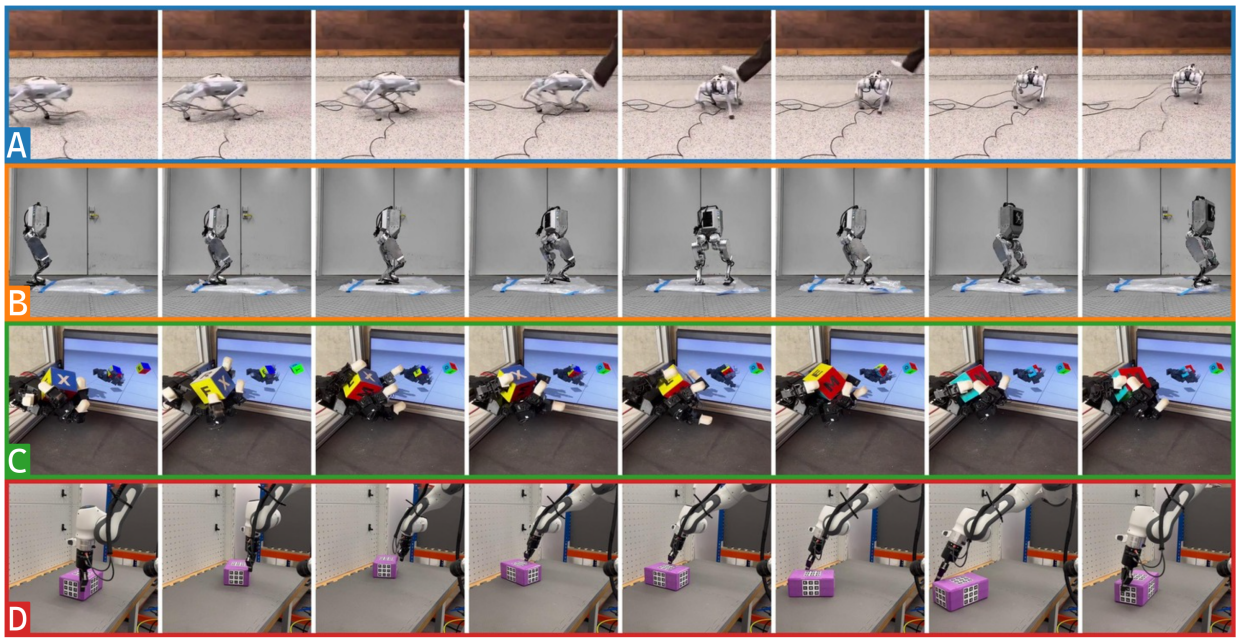

Figure 5 | Footage from four of our deployed policies. a) Go1 joystick policy recovering from a kick while travelling at ~ 2m/s, b) Berkeley humanoid joystick policy tracking an angular velocity command on a slippery surface. c) In-Hand Cube Reorientation transitioning between two target poses. d) Non-prehensile policy issuing torque commands to rotate a block by 180 degrees.

other DM Control Suite environments can also be rendered with Madrona, we demonstrate end-to-end RL training on only one task, leaving a more comprehensive exploration for future work. Appendix E provides more information on how CartpoleBalance was modified and trained for pixel observations.

4.2. Locomotion

We present sim-to-real locomotion results on both a quadruped (Unitree Go1) and two humanoid platforms (Berkeley Humanoid and Unitree G1). Further details on the MDP formulation, including rewards, observation spaces, and action spaces, are provided in Appendix C.

4.2.1. Quadruped Locomotion

Task definition. We implement a joystick locomotion task as in [26, 50], where the command is specified by three values indicating the desired forward velocity, lateral velocity, and turning rate of the robot's root body. Additionally, we design policies for handstand and footstand tasks, in which the robot balances on the front or hind legs, respectively, while minimizing actuator torque. For fall recovery, we follow [31, 58], enabling the robot to return to a stable "home" posture from arbitrary fallen configurations.

Hardware. We deploy on the *Unitree Go1*, which is a quadruped robot with four legs, each possessing three degrees of freedom. Trained policies run on real-world outdoor terrain (grass and concrete) and indoor surfaces with different friction properties.

Training. We domain randomize for sensor noise, dynamics properties and task uncertainties. We firstly train the policy in flat ground with restricted command ranges within 5 minutes (2x RTX 4090). and finetune it in rough terrain with wider ranges. See Appendix C for more detail.

Results. All four policies (joystick, handstand, footstand, and fall recovery) transfer robustly from simulation to reality, coping with uneven terrain and moderate external perturbations without additional fine-tuning. Videos of these deployments are provided on our project website.

4.2.2. *Humanoid Locomotion*

Task definition. We implement the same joystick locomotion task as shown for the quadruped environment.

Hardware. We perform sim-to-real experiments on two different humanoid platforms: a) *Berkeley Humanoid* [35], a low-cost, lightweight bipedal robot with 6 DoF per leg, and *Unitree G1*, a humanoid robot featuring 29 DoF in total. Both systems are evaluated in indoor environments, with slight variations in surface friction and ground compliance.

Training. We follow the domain randomization and finetuning strategies of the quadruped robot. Training on flat ground lasts under 15 minutes for the Berkeley Humanoid, and under 30 minutes for the Unitree G1 on two RTX 4090.

Results. We successfully deploy joystick-based locomotion on the Berkeley Humanoid, demonstrating robust tracking of velocity commands on surfaces ranging from rigid floors to soft and slippery terrains. On the Unitree G1, our zero-shot policy similarly achieves stable walking and turning on standard indoor floors. Although minor tuning for each platform's unique dynamics may further enhance performance, these results confirm that our approach generalizes across a range of legged robot morphologies.

4.3. Manipulation

In this section, we present sim-to-real results for a broad range of manipulation tasks, including dexterous in-hand manipulation, non-prehensile manipulation, and vision-based grasping. These tasks illustrate Playground's ability to address a diverse segment of the manipulation spectrum and highlight its robust deployment in real-world settings.

4.3.1. *In-Hand Cube Reorientation*

Task definition. We implement an in-hand cube reorientation task using the low-cost, dexterous LEAP hand platform [56], closely following previous works on in-hand manipulation [5, 20]. The task involves reorienting a 7 cm cube repeatedly from random initial poses to new target orientations in SE(3) without dropping it. Further task details are provided in Appendix D.4.

Table 1 | In-hand reorientation results on the LEAP hand over 10 trials, reporting the number of consecutive successful rotations before failure. The final two columns show the median and mean of the #Rotations metric.

	Trial										**Summary**	
	1	2	3	4	5	6	7	8	9	10	Median	Mean
# Rotations	3	27	8	2	15	3	4	1	3	5	**3.5**	**7.1**

Hardware. We employ the same hardware configuration as in [2], mounting the LEAP hand on an 80/20 frame with a 3D-printed bracket that tilts the palm downward by 20°. A single Intel RealSense D415 camera, positioned above the workspace, provides pose estimates of the cube via a pretrained detector [20]. Although occlusions can introduce observation noise, we leave multi-camera extensions to future work. The policy operates at 20 Hz, which remains comfortably below the USB-Dynamixel control bandwidth.

Training. To promote sim-to-real transfer, we apply domain randomization on the robot parameters as well as cube mass and friction. We also include sensor noise, and we finetune with a progressive curriculum to increase both noisy pose estimates and action regularization. The policy trains within 30 min on two RTX 4090 GPUs. Further training details are provided in Appendix D.4.

Results. As summarized in Table 1, our learned policy demonstrates early signs of robust in-hand reorientation with MuJoCo Playground. The most frequent failure occurs when the cube becomes wedged in the space present between the fingers and the palm of the LEAP hand, causing the policy to stall. Although less common, we also observe accidental interlocking of the index and thumb, attributed to physical flex in the low-cost hardware. Videos of real-world deployments can be found on our project page. We note that improved camera coverage and more accurate collision geometries could mitigate these edge-case failures, which we leave for future work.

4.3.2. *Non-Prehensile Block Reorientation*

Task definition. We present a sim-to-real setup for non-prehensile reorientation of a yoga block on a commonly available Franka Emika Panda robot arm with a Robotiq gripper, achieving high zero-shot success. The task involves moving a yoga block from a random initial pose in the robot's workspace to a fixed goal pose. A trial is deemed successful if the agent reorients the block within 3 cm of the goal position and within 10° of the desired orientation.

Hardware. The policy receives estimates of the block's position and orientation from an open-source camera tracker [44]. We use direct high-frequency torque control at 200 Hz, where the RL policy outputs motor torques for the arm's seven joints (with the gripper closed). By learning to control torques rather than joint positions, the agent develops smooth, compliant behavior that transfers effectively to hardware, delivering superior performance even when direct torque control at high frequencies poses learning challenges [23]. This recipe, therefore, holds broad value for practitioners.

Training. Robust zero-shot transfer is enabled by stochastic delays and progressive curriculum learning. Each training episode injects randomization into initial poses, joint positions, and velocities, while also imposing action and observation stochastic delays to mirror practical hardware latency. A simple curriculum gradually increases the block's displacement and orientation range upon each success, preventing overfitting to easier conditions. Training takes 10 minutes on 16x A100 devices.

Results. These techniques, combined with 200 Hz direct torque control, produce a policy resilient to real-world perturbations. The agent reliably reorients the block on hardware with no additional fine-tuning as shown in Table 2. Videos of real-world deployments are provided on our project website. Additional implementation details are given in Appendix D.5.

Table 2 | Sim-to-real reorientation performance on the Franka Emika Panda robot, evaluated across 35 hardware trials. Each metric is reported as the median and mean (with a 95% confidence interval). The success rate is bolded to highlight final task performance. The training was done on 16x A100 GPUs.

Metric	Median	Mean ± 95% Confidence Interval
Real Success (%) ↑	**100**	**85.7 ± 12.2**
Position Error (cm) ↓	1.95	5.28 ± 3.26
Rotation Error (°) ↓	1.72	3.32 ± 1.59

4.3.3. *Pick-Cube from Pixels*

Task definition. We demonstrate sim-to-real transfer with pixel-based policies on a Franka Emika Panda robot. The robot must reliably grasp and lift a small $2 \times 2 \times 3$ cm block from a random location on the table and move it 10 cm above the surface. The policy receives a 64×64 RGB image as input and outputs a Cartesian command, which is processed by a closed-form inverse kinematics solution to yield joint commands. To simplify the task, we restrict the end-effector to a 2D Y-Z plane (while always pointing downward) and provide a binary jaw open/close action.

Hardware. We use a Franka Emika Panda robot with a single Intel RealSense D435 camera mounted to capture top-down RGB images. The policy operates at 15 Hz, and we run inference on an RTX 3090 GPU. Our setup ensures that the block starts within the field of view over a 20 cm range along the y-axis.

Training. To bridge the sim-to-real gap, we apply domain randomization across visual properties such as lighting, shadows, camera pose, and object colors. We also add random brightness post-processing, and introduce a stochastic gripping delay of up to 250 ms. We choose a reduced action dimension of three (Y-movement, Z-movement, and discrete jaw control) for training sample efficiency, but we have found that the task can also be solved in full Cartesian or joint space given additional camera perspectives and more training samples. Training in simulation takes ten minutes on a single RTX 4090.

Results. Our policy achieves a 100% success rate in 12 real-world trials, robustly grasping the block and lifting it clear of the table. It demonstrates resilience to moderate variations in lighting and minor camera shaking, as shown

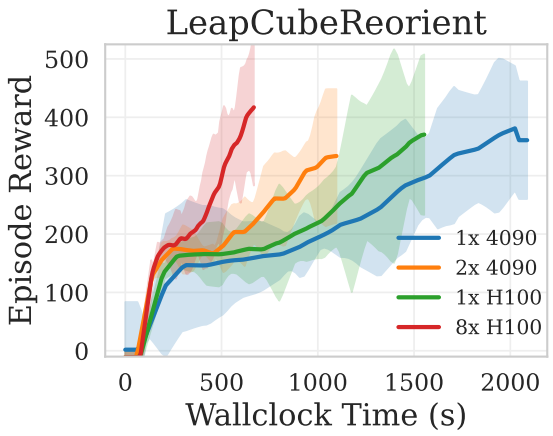

Figure 6 | Training wallclock time for LeapCubeReorient on different GPU device topologies. 1x 4090 takes ~ 2080 (s) to train and 8x H100 takes ~ 670 (s) to train. All runs use the *same hyperparams* (e.g. 8192 num envs); we leave tuning hyperparams per topology as a future exercise.

in the videos on our project website. These findings highlight MuJoCo Playground's capacity for training pixel-based policies that transfer reliably to real hardware in a zero-shot manner. Additional implementation details are described in Appendix E.

4.4. Training Throughput

Across our sim-to-real studies, we used several GPU hardware setups and topologies, including NVIDIA RTX 4090, A100, and H100 GPUs. In Figure 6, we break down the training performance of the LeapCubeReorient environment on different configurations for a fixed set of RL hyperparameters, demonstrating that MJX is effective on both consumer-grade and datacenter graphics cards. We see that GPUs with higher theoretical performance and larger topologies can reduce training time by a factor of 3x on a contact-rich task like in-hand reorientation. We leave optimization of topology-specific hyper-parameters as future work (e.g. the number of environments should ideally increase for larger topologies to maximize throughput, as long as the RL algorithm can utilize the increase in data per epoch). In Table 4, Table 7, and Table 9 in the appendix, we report RL training throughput for all environments

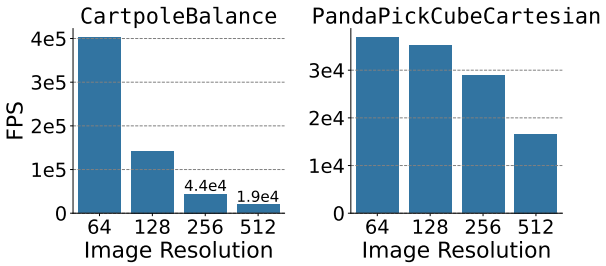

Figure 7 | Environment steps per second on the single-camera CartpoleBalance and PandaPickCubeCartesian environments with pixel-based observations from our on-device renderer.

in MuJoCo Playground on a single A100 GPU.

4.4.1. Training Throughput with Batch Rendering

Figure 7 highlights the throughput of stepping two of our environments with pixel observations at different resolutions. By pairing MJX physics with Madrona batch rendering, our Cartpole and Franka environments unroll at roughly 403,000 and 37,000 steps per second respectively. Note that our Franka physics are over 20x more costly than Cartpole's, resulting in the lower sensitivity of FPS to image resolution.

Computationally, pixel-based policy training generally involves four main components: physics simulation, observation rendering, policy inference and policy updates. Figure 7 only encapsulates the former two and is not fully indicative of overall training throughput.

We find that in the context of a PPO training loop, physics, rendering, and inference together only comprise 9% and 43% of the Cartpole and Franka total training times, respectively, with most of the time spent updating the expensive CNN-based networks. Hence, compared to traditional on-policy training pipelines, we have shifted our bottleneck from collecting data to processing it. Training bottlenecks are further discussed in Appendix E.3 under Table 10 and Table 11. Further performance benchmarking and a rough comparison against prior simulators are in Appendix E.2.

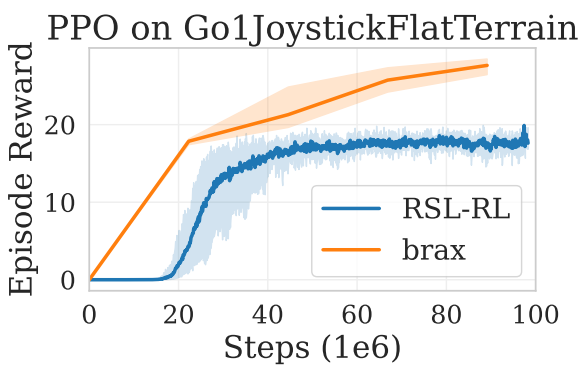

Figure 8 | Reward curves for PPO trained with RSL-RL and brax on an RTX-4090 GPU for 3 seeds each on the Unitree Go1.

4.4.2. RL Libraries

While MuJoCo Playground primarily uses a JAX-based physics simulator, practitioners are able to use both JAX and torch-based RL libraries for training RL agents. In Figure 8, we show reward curves for PPO agents trained using both Brax [14] and RSL-RL [32] implementations. Each corresponding RL library is trained with custom hyperparameters tailored to the corresponding PPO implementation. Both libraries are able to achieve successful rewards and gaits within similar wallclock times. All other results in this paper were obtained using the Brax PPO and SAC implementations.

5. Related Work

Physics simulation on GPU. The PhysX GPU implementation [34] has been heavily relied on for robotic sim-to-real workloads via IsaacGym [39] and more recently Isaac Lab [41]. The PhysX GPU implementation, however, is closed-source [34] and researchers lack the ability to extend the simulator for their specific tasks or workloads. Several GPU-based physics engines are open-source, such as MJX [43, 63], Brax [14], Warp [38], and Taichi [24]. Only a limited set of robot environments [52, 66] leverage these open-source counterparts, in contrast to the wide range of robotic sim-to-real results that were achieved with IsaacGym and Isaac Lab. Most recently, Genesis [15] provides a rigid-body implementation similar to MJX implemented using Taichi, that allows for dynamic constraints/contacts. However, sim-to-real results are still limited to a few locomotion policies.

Sim-to-real RL. A variety of locomotion and manipulation policies have successfully been deployed in the real world zero-shot [10, 11, 33, 36, 47, 57, 70]. We complement these results by demonstrating zero-shot sim-to-real on the Leap Hand, Unitree Go1, Berkeley Humanoid, Unitree G1, and Franka arm using MuJoCo rather than closed-source simulators. Similar to [39, 41], we provide code for environments and training.

Vision-based RL. State-of-the-art algorithms such as DrQ [67], RL from Augmented Data (RAD) [30], Dreamerv3 [18], TD-MPC2 [21], and EfficientZeroV2 [64] have pushed pixel-based RL performance over the years. Transferring these advances to the real world is appealing, as visual control loops offer precise positioning and robust behaviour in uncontrolled in-the-wild scenarios [19]. The limitation of training directly from pixel data is the large visual sim-to-real gap between simulation and reality, which is often overcome using domain randomization [62]. However, such training methods require exponentially more training samples. As as result, policies are typically trained with proprioceptive observations in simulation and subsequently distilled into vision-based policies offline [9, 10, 17], or trained with smaller exteroceptive observations [1, 40]. With Madrona, we are able to train vision-based policies directly in simulation without a distillation step using high-throughput batch rendering, similar to [41] and [60].

6. Limitations

MuJoCo Playground inherits the limitations of MJX due to constraints imposed by JAX. First, just-in-time (JIT) compilation can be slow (1-3 minutes on Playground's tasks). Second, computation time related to contacts does not scale like the number of *active* contacts in the scene, but like the number of *possible* contacts in the scene. This is due to JAX's requirement of static shapes at compile time. This limitation can be overcome by using more flexible frameworks like Warp [38]

and Taichi [15]. This upgrade is an active area of development. Finally we should note that the vision-based training using Madrona is still at an early stage.

7. Conclusion

MuJoCo Playground is a library built upon the open-source MuJoCo simulator and Madrona batch renderer with implementations across several reinforcement learning and robotics environments. We demonstrate policy training on various GPU topologies using JAX and pytorch-based reinforcement learning libraries. We also demonstrate sim-to-real deployment on several robotic tasks and embodiments, from locomotion to both dexterous and non-prehensile manipulation from proprioceptive state and from pixels. We look forward to seeing the community put this resource to use in advancing robotics research and its applications.

Acknowledgments

We thank Jimmy Wu, Kyle Stachowicz, Kenny Shaw, and Zhongyu Li for help with hardware. We thank Dongho Khang and Yunhao Cao for help with locomotion. We thank Rushrash Hari for help with hardware. We thank Luc Guy Rosenzweig, Brennan Shacklett and Kayvon Fatahalian for their extensive support in integrating the Madrona project into MJX. We thank Ankur Handa for help with manipulation. We thank Laura Smith and Philipp Wu for always being there to help with any problem and answer any question. We thank Lambda labs for sponsoring compute for the project. We thank Stone Tao for discussions on manipulation environments in MJX. We thank Erwin Coumans for introducing us to the Madrona team. We thank Kevin Bergamin and Michael Lutter for fruitful technical discussions and paper draft feedback. We thank Brent Yi for fruitful technical discussions and help with the website. We thank Lambda Labs for supporting this project with cloud compute credits.

This work is supported in part by The AI Institute. K. Sreenath has financial interest in Boston Dynamics AI Institute LLC. He and the company may benefit from the commercialization of the results of this research.

This work was supported in part by the ONR Science of Autonomy Program N000142212121 and the BAIR Industrial Consortium. Pieter Abbeel holds concurrent appointments as a Professor at UC Berkeley and as an Amazon Scholar. This paper describes work performed at UC Berkeley and is not associated with Amazon.

References

[1] A. Agarwal, A. Kumar, J. Malik, and D. Pathak. Legged locomotion in challenging terrains using egocentric vision. In *Conference on robot learning*, pages 403–415. PMLR, 2023.

[2] V. K. Albert H. Li, Preston Culbertson and A. D. Ames. Drop: Dexterous reorientation via online planning. *arXiv preprint arXiv:2409.14562*, 2024. Available at: https://arxiv.org/abs/2409.14562.

[3] J. ALOHA 2 Team, Aldaco, T. Armstrong, R. Baruch, J. Bingham, S. Chan, K. Draper, D. Dwibedi, C. Finn, P. Florence, S. Goodrich, et al. Aloha 2: An enhanced low-cost hardware for bimanual teleoperation. *arXiv preprint arXiv:2405.02292*, 2024.

[4] M. Andrychowicz, A. Raichuk, P. Stańczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski, et al. What matters in on-policy reinforcement learning? a large-scale empirical study. *arXiv preprint arXiv:2006.05990*, 2020.

[5] 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.

[6] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula,

A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

[7] K. Caluwaerts, A. Iscen, J. C. Kew, W. Yu, T. Zhang, D. Freeman, K.-H. Lee, L. Lee, S. Saliceti, V. Zhuang, et al. Barkour: Benchmarking animal-level agility with quadruped robots. *arXiv preprint arXiv:2305.14654*, 2023.

[8] Y. Chebotar, A. Handa, V. Makoviychuk, M. Macklin, J. Issac, N. Ratliff, and D. Fox. Closing the sim-to-real loop: Adapting simulation randomization with real world experience. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8973–8979. IEEE, 2019.

[9] T. Chen, J. Xu, and P. Agrawal. A system for general in-hand object re-orientation. In *Conference on Robot Learning*, pages 297–307. PMLR, 2022.

[10] X. Cheng, K. Shi, A. Agarwal, and D. Pathak. Extreme parkour with legged robots. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 11443–11450. IEEE, 2024.

[11] Y. Cho, J. Han, Y. Cho, and B. Kim. Corn: Contact-based object representation for nonprehensile manipulation of general unseen objects. *arXiv preprint arXiv:2403.10760*, 2024.

[12] O. R. developers. Onnx runtime. https://onnxruntime.ai/, 2021. Version: x.y.z.

[13] T. Flayols, A. Del Prete, P. Wensing, A. Mifsud, M. Benallegue, and O. Stasse. Experimental evaluation of simple estimators for humanoid robots. In *2017 IEEE-RAS 17th International Conference on Humanoid Robotics (Humanoids)*, pages 889–895, 2017. doi: 10.1109/HUMANOIDS.2017.8246977.

[14] C. D. Freeman, E. Frey, A. Raichuk, S. Girgin, I. Mordatch, and O. Bachem. Brax-a differentiable physics engine for large scale rigid body simulation, 2021. *URL http://github.com/google/brax*, 6, 2021.

[15] Genesis-Authors. Genesis: A universal and generative physics engine for robotics and beyond, December 2024. URL https://github.com/Genesis-Embodied-AI/Genesis.

[16] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[17] T. Haarnoja, B. Moran, G. Lever, S. H. Huang, D. Tirumala, J. Humplik, M. Wulfmeier, S. Tunyasuvunakool, N. Y. Siegel, R. Hafner, et al. Learning agile soccer skills for a bipedal robot with deep reinforcement learning. *Science Robotics*, 9 (89):eadi8022, 2024.

[18] D. Hafner, J. Pasukonis, J. Ba, and T. Lillicrap. Mastering diverse domains through world models. *arXiv preprint arXiv:2301.04104*, 2023.

[19] M. Haiderbhai, R. Gondokaryono, A. Wu, and L. A. Kahrs. Sim2real rope cutting with a surgical robot using vision-based reinforcement learning. *Transactions on Automation Science and Engineering*, 2024. doi: 10.1109/TASE.2024.3410297.

[20] A. Handa, A. Allshire, V. Makoviychuk, A. Petrenko, R. Singh, J. Liu, D. Makoviichuk, K. Van Wyk, A. Zhurkevich, B. Sundaralingam, et al. Dextreme: Transfer of agile in-hand manipulation from simulation to reality. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5977–5984. IEEE, 2023.

[21] N. Hansen, H. Su, and X. Wang. Td-mpc2: Scalable, robust world models for continuous control, 2024.

[22] Y. He and S. Liu. Analytical inverse kinematics for franka emika panda – a geometrical solver for 7-dof manipulators with un-

conventional design. In *2021 9th International Conference on Control, Mechatronics and Automation (ICCMA)*, pages 194–199, 2021. doi: 10.1109/ICCMA54375.2021.9646185.

[23] S. Holt, T. Davchev, D. Tirumala, B. Moran, Y. Lin, A. Laurens, A. Iscen, E. Frey, M. Wulfmeier, F. Romano, and N. Heess. Evolving control: Evolved high frequency control for continuous control tasks. In *CoRL Workshop on Safe and Robust Robot Learning for Operation in the Real World*, 2024. URL https://openreview.net/forum?id=gzADUWLD9X.

[24] Y. Hu, L. Anderson, T.-M. Li, Q. Sun, N. Carr, J. Ragan-Kelley, and F. Durand. Difftaichi: Differentiable programming for physical simulation. *arXiv preprint arXiv:1910.00935*, 2019.

[25] J. Ibarz, J. Tan, C. Finn, M. Kalakrishnan, P. Pastor, and S. Levine. How to train your robot with deep reinforcement learning: lessons we have learned. *The International Journal of Robotics Research*, 40(4-5):698–721, 2021.

[26] G. Ji, J. Mun, H. Kim, and J. Hwangbo. Concurrent training of a control policy and a state estimator for dynamic and robust legged locomotion. *IEEE Robotics and Automation Letters*, 7(2):4630–4637, Apr. 2022. ISSN 2377-3774. doi: 10.1109/lra.2022.3151396. URL http://dx.doi.org/10.1109/LRA.2022.3151396.

[27] E. Kaufmann, L. Bauersfeld, A. Loquercio, M. Müller, V. Koltun, and D. Scaramuzza. Champion-level drone racing using deep reinforcement learning. *Nature*, 620(7976):982–987, 2023.

[28] J. Kober, J. A. Bagnell, and J. Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[29] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ international conference on intelligent robots and systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. Ieee, 2004.

[30] M. Laskin, K. Lee, A. Stooke, L. Pinto, P. Abbeel, and A. Srinivas. Reinforcement learning with augmented data. arXiv:2004.14990.

[31] J. Lee, J. Hwangbo, and M. Hutter. Robust recovery controller for a quadrupedal robot using deep reinforcement learning. *arXiv preprint arXiv:1901.07517*, 2019.

[32] leggedrobotics. rsl_rl: Fast and simple implementation of rl algorithms, designed to run fully on gpu. https://github.com/leggedrobotics/rsl_rl, 2023. Accessed: January 10, 2025.

[33] Z. Li, X. B. Peng, P. Abbeel, S. Levine, G. Berseth, and K. Sreenath. Reinforcement learning for versatile, dynamic, and robust bipedal locomotion control. *The International Journal of Robotics Research*, page 02783649241285161, 2024.

[34] J. Liang, V. Makoviychuk, A. Handa, N. Chentanez, M. Macklin, and D. Fox. Gpu-accelerated robotic simulation for distributed reinforcement learning. In *Conference on Robot Learning*, pages 270–282. PMLR, 2018.

[35] Q. Liao, B. Zhang, X. Huang, X. Huang, Z. Li, and K. Sreenath. Berkeley humanoid: A research platform for learning-based control. *arXiv preprint arXiv:2407.21781*, 2024.

[36] J. Long, J. Ren, M. Shi, Z. Wang, T. Huang, P. Luo, and J. Pang. Learning humanoid locomotion with perceptive internal model. *arXiv preprint arXiv:2411.14386*, 2024.

[37] Y. J. Ma, W. Liang, G. Wang, D.-A. Huang, O. Bastani, D. Jayaraman, Y. Zhu, L. Fan, and A. Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*, 2023.

[38] M. Macklin. Warp: A high-performance python framework for gpu simulation and graphics. In *NVIDIA GPU Technology Conference (GTC)*, 2022.

[39] V. Makoviychuk, L. Wawrzyniak, Y. Guo, M. Lu, K. Storey, M. Macklin, D. Hoeller, N. Rudin, A. Allshire, A. Handa, et al. Isaac gym: High performance gpu-based physics simulation for robot learning. *arXiv preprint arXiv:2108.10470*, 2021.

[40] T. Miki, J. Lee, J. Hwangbo, L. Wellhausen, V. Koltun, and M. Hutter. Learning robust perceptive locomotion for quadrupedal robots in the wild. *Science robotics*, 7(62): eabk2822, 2022.

[41] M. Mittal, C. Yu, Q. Yu, J. Liu, N. Rudin, D. Hoeller, J. L. Yuan, R. Singh, Y. Guo, H. Mazhar, et al. Orbit: A unified simulation framework for interactive robot learning environments. *IEEE Robotics and Automation Letters*, 8(6):3740–3747, 2023.

[42] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518 (7540):529–533, 2015. doi: 10.1038/nature14236. URL https://doi.org/10.1038/nature14236.

[43] MuJoCo XLA Authors. MuJoCo XLA (MJX). https://mujoco.readthedocs.io/en/stable/mjx.html. Accessed: December 16, 2024.

[44] S. Niekum and I. I. Saito. ar_track_alvar, 2016. URL https://github.com/ros-perception/ar_track_alvar.

[45] A. Petrenko, A. Allshire, G. State, A. Handa, and V. Makoviychuk. Dexpbt: Scaling up dexterous manipulation for hand-arm systems with population based training. *RSS*, 2023.

[46] L. Pinto, M. Andrychowicz, P. Welinder, W. Zaremba, and P. Abbeel. Asymmetric actor critic for image-based robot learning. *RSS*, 2018.

[47] I. Radosavovic, S. Kamat, T. Darrell, and J. Malik. Learning humanoid locomotion over challenging terrain. *arXiv preprint arXiv:2410.03654*, 2024.

[48] P. Ramachandran, B. Zoph, and Q. V. Le. Searching for activation functions, 2017. URL https://arxiv.org/abs/1710.05941.

[49] L. G. Rosenzweig, B. Shacklett, W. Xia, and K. Fatahalian. High-throughput batch rendering for embodied ai. 2024.

[50] N. Rudin, D. Hoeller, P. Reist, and M. Hutter. Learning to walk in minutes using massively parallel deep reinforcement learning. In *Conference on Robot Learning*, pages 91–100. PMLR, 2022.

[51] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[52] C. Sferrazza, D.-M. Huang, X. Lin, Y. Lee, and P. Abbeel. Humanoidbench: Simulated humanoid benchmark for whole-body locomotion and manipulation. *Robotics: Science and Systems (RSS)*, 2024.

[53] B. Shacklett, L. G. Rosenzweig, Z. Xie, B. Sarkar, A. Szot, E. Wijmans, V. Koltun, D. Batra, and K. Fatahalian. An extensible, data-oriented architecture for high-performance, many-world simulation. *ACM Transactions on Graphics (TOG)*, 42(4):1–13, 2023.

[54] B. Shacklett, L. G. Rosenzweig, Z. Xie, B. Sarkar, A. Szot, E. Wijmans, V. Koltun, D. Batra, and K. Fatahalian. An extensible, data-oriented architecture for high-performance, many-world simulation. *ACM Trans. Graph.*, 42(4), 2023.

[55] Y. Shao, Y. Jin, X. Liu, W. He, H. Wang, and W. Yang. Learning free gait transition

for quadruped robots via phase-guided controller. *IEEE Robotics and Automation Letters*, 7(2):1230–1237, 2021.

[56] K. Shaw, A. Agarwal, and D. Pathak. Leap hand: Low-cost, efficient, and anthropomorphic hand for robot learning. *Robotics: Science and Systems (RSS)*, 2023.

[57] R. Singh, A. Allshire, A. Handa, N. Ratliff, and K. Van Wyk. Dextrah-rgb: Visuomotor policies to grasp anything with dexterous hands. *arXiv preprint arXiv:2412.01791*, 2024.

[58] L. Smith, J. C. Kew, X. B. Peng, S. Ha, J. Tan, and S. Levine. Legged robots that keep on learning: Fine-tuning locomotion policies in the real world. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 1593–1599. IEEE, 2022.

[59] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. *arXiv preprint arXiv:1804.10332*, 2018.

[60] S. Tao, F. Xiang, A. Shukla, Y. Qin, X. Hinrichsen, X. Yuan, C. Bao, X. Lin, Y. Liu, T.-k. Chan, et al. Maniskill3: Gpu parallelized robotics simulation and rendering for generalizable embodied ai. *arXiv preprint arXiv:2410.00425*, 2024.

[61] Y. Tassa, Y. Doron, A. Muldal, T. Erez, Y. Li, D. d. L. Casas, D. Budden, A. Abdolmaleki, J. Merel, A. Lefrancq, et al. Deepmind control suite. *arXiv preprint arXiv:1801.00690*, 2018.

[62] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30, 2017.

[63] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pages 5026–5033. IEEE, 2012.

[64] S. Wang, S. Liu, W. Ye, J. You, and Y. Gao. Efficientzero v2: Mastering discrete and continuous control with limited data. *arXiv preprint arXiv:2403.00564*, 2024.

[65] Y. Wang, Q. Wang, S. Shi, X. He, Z. Tang, K. Zhao, and X. Chu. Benchmarking the performance and energy efficiency of ai accelerators for ai training. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 744–751. IEEE, 2020.

[66] H. Xue, C. Pan, Z. Yi, G. Qu, and G. Shi. Full-order sampling-based mpc for torque-level locomotion control via diffusion-style annealing. *arXiv preprint arXiv:2409.15610*, 2024.

[67] D. Yarats, R. Fergus, A. Lazaric, and L. Pinto. Mastering visual continuous control: Improved data-augmented reinforcement learning. *arXiv preprint arXiv:2107.09645*, 2021.

[68] K. Zakka, Y. Tassa, and MuJoCo Menagerie Contributors. MuJoCo Menagerie: A collection of high-quality simulation models for MuJoCo, 2022. URL http://github.com/google-deepmind/mujoco_menagerie.

[69] W. Zhao, J. P. Queralta, and T. Westerlund. Sim-to-real transfer in deep reinforcement learning for robotics: a survey. In *2020 IEEE symposium series on computational intelligence (SSCI)*, pages 737–744. IEEE, 2020.

[70] Z. Zhuang, Z. Fu, J. Wang, C. Atkeson, S. Schwertfeger, C. Finn, and H. Zhao. Robot parkour learning. *arXiv preprint arXiv:2309.05665*, 2023.

Appendices

Table of Contents

A. Author Contributions

A.1. Real World Experiments

Locomotion

- Qiayuan Liao and Kevin Zakka set up and iterated on all Go1 and Berkeley Humanoid tasks and real-world deployments.
- Qiayuan Liao, Kevin Zakka, and Carmelo Sferrazza set up, iterated on and deployed policies on the G1.

Manipulation

- Kevin Zakka and Arthur Allshire built and set up deployment on the open-source LEAP hand. Arthur implemented vision-based cube state estimation, enabling sim-to-real transfer.
- Kevin Zakka and Qiayuan Liao iterated on in-hand reorientation results.
- Mustafa Haiderbhai and Jing Yuan Luo designed the visual Panda Pick Cube experiments and deployed them on real hardware.
- Samuel Holt and Baruch Tabanpour designed the non-prehensile manipulation task and deployed it on real hardware. Samuel Holt built and setup deployment for both the robot controller in C++ and vision-based block state estimation pipeline for the Franka-Robotiq arm, calibration, and iterated the task and setup to enable reliable sim-to-real transfer, through accurate tracking and latency optimization in the real setup.

A.2. RL and Simulation

Locomotion

- Baruch Tabanpour implemented the first Barkour joystick task and helped replicate the Barkour results on the Go1.
- Kevin Zakka implemented various joystick and fall-recovery tasks for multiple platforms, including Unitree Go1, G1, Berkeley Humanoid, and others.
- Carmelo Sferrazza iterated on tasks for Berkeley Humanoid, H1, and G1, achieving the first implementation and walking gaits for G1.

Manipulation

- Baruch Tabanpour implemented the first open-source manipulation environment: Franka Pick Cube.
- Kevin Zakka implemented the LEAP hand rotation and re-orientation tasks, with contributions from Baruch Tabanpour for basic reward structure and sweeps.
- Arthur Allshire significantly improved the LEAP hand task results.
- Samuel Holt implemented the first version of the Panda Push Cube task including domain randomization, which was refactored and improved from contributions from Baruch Tabanpour for improved rewards, sweeps and domain randomization.
- Jing Yuan Luo implemented the Open Cabinet task and adapted Pick Cube for orientation targets.
- Erik Frey implemented the first version of the ALOHA task, with tuning and finalization by Baruch and Andrew.

A.2.1. DM Control Suite

- Kevin Zakka ported over the DM Control Suite tasks and tuned the first RL baseline using SAC.
- Baruch Tabanpour added the PPO baseline and ran hyperparameter sweeps for both RL baselines.

- Jing Yuan Luo implemented and tuned the Vision Cartpole Task.

PyTorch Integration

- Arthur Allshire led and implemented the integration of Playground with PyTorch and rsl_rl.

A.3. Miscellaneous

- Baruch Tabanpour advised and improved simulation speed for both manipulation and locomotion tasks.
- Jing Yuan Luo worked on paper figures and writing.
- Samuel Holt contributed the figure plotting code.

A.4. Vision

- Mustafa Haiderbhai worked on Madrona-MJX feature development.
- Mustafa Haiderbhai and Jing Yuan Luo worked on Madrona-MJX integration into Playground.
- Jing Yuan Luo benchmarked Madrona-MJX.
- Erik Frey and Baruch Tabanpour contributed to early versions of Madrona-MJX.

A.5. Software Infrastructure

- Baruch Tabanpour and Kevin Zakka led the development and release of MuJoCo Playground.
- Baruch Tabanpour advised on brax and MuJoCo MJX, enabling upstream changes.
- Kevin Zakka added the asymmetric actor-critic implementation to the brax PPO codebase.
- Jing Yuan Luo added vision support to the brax PPO codebase.
- Kevin Zakka wrote MJX-to-MuJoCo sim2sim deployment code, with contributions from Carmelo Sferrazza for the joystick interface.
- Kevin Zakka and Jing Yuan Luo developed the Brax Flax to ONNX conversion script.

A.6. Project Management

- Kevin Zakka and Baruch Tabanpour conceived and led the project.
- Baruch Tabanpour led the paper writing effort.
- Carmelo Sferrazza, Erik Frey, Yuval Tassa, and Pieter Abbeel advised and guided the project.
- Carmelo Sferrazza and Yuval Tassa contributed significantly to paper writing.
- Pieter Abbeel, Erik Frey, Koushil Sreenath and Yuval Tassa provided resource support and feedback.

B. DM Control Suite

B.1. Environments

In Table 3, we show the environments from DM Control Suite ([61]) that were re-implemented in MuJoCo Playground. Certain XMLs were modified for performance and are shown in the table.

B.2. RL Training Results

For all DM Control Suite environments ported to MuJoCo Playground, we train both PPO [51] and SAC [16] using the RL implementations in [14] and we report reward curves below. In Figure 9 we report environment steps versus reward and in Figure 10 we report wallclock time versus reward. All environments are run across 5 seeds on a single A100 GPU.

B.3. RL Training Throughput

We report training throughput on all DM Control Suite environments in Table 4 by dividing the number of environment steps by wallclock time, as reported in Appendix B.2, for each RL algorithm.

Env	MJX	XML Modifications
acrobot-swingup	✓	iterations=2, ls_iterations=4
acrobot-swingup_sparse	✓	
ball_in_cup-catch	✓	iterations=1, ls_iterations=4
cartpole-balance	✓	iterations=1, ls_iterations=4
cartpole-balance_sparse	✓	
cartpole-swingup	✓	
cartpole-swingup_sparse	✓	
cheetah-run	✓	iterations=4, ls_iterations=8, max_contact_points=6, max_geom_pairs=4
finger-spin	✓	iterations=2, ls_iterations=8, max_contact_points=4, max_geom_pairs=2, removed cylinder collision
finger_turn_easy	✓	
finger_turn_hard	✓	
fish-upright	✓	iterations=2, ls_iterations=6, disabled contacts
fish-swim	✓	
hopper-stand	✓	iterations=4, ls_iterations=8, max_contact_points=6, max_geom_pairs=2
hopper-hop	✓	
humanoid-stand	✓	timestep=0.005, max_contact_points=8, max_geom_pairs=8
humanoid-walk	✓	
humanoid-run	✓	
pendulum-swingup	✓	timestep=0.01, iterations=4, ls_iterations=8
point_mass-easy	✓	iterations=1, ls_iterations=4
reacher-easy	✓	timestep=0.005, iterations=1, ls_iterations=6
reacher-hard	✓	
swimmer-swimmer6	✓	timestep=0.003, iterations=4, ls_iterations=8, con-type/conaffinity set to 0
swimmer-swimmer15	×	
walker-stand	✓	timestep=0.005, iterations=2, ls_iterations=5, max_contact_points=4, max_geom_pairs=4
walker-walk	✓	
walker-run	✓	
manipulator-bring_ball	×	
manipulator-bring_peg	×	
manipulator-insert_ball	×	
manipulator-insert_peg	×	
dog-stand	×	
dog-walk	×	
dog-trot	×	
dog-run	×	
dog-fetch	×	

Table 3 | DM Control Suite Environments ported to MJX. Where specified, XML modifications were made to the solver iterations, line search iterations, as well as contact custom parameters for MJX.

Env	PPO Steps per Second	SAC Steps Per Second
AcrobotSwingup	752092 ± 11562	30661 ± 244
AcrobotSwingupSparse	750597 ± 4640	30624 ± 210
BallInCup	235899 ± 565	15492 ± 283
CartpoleBalance	718626 ± 6894	30891 ± 168
CartpoleBalanceSparse	721061 ± 14135	31031 ± 183
CartpoleSwingup	728088 ± 12503	30870 ± 207
CartpoleSwingupSparse	718355 ± 10189	31061 ± 226
CheetahRun	435162 ± 12183	18819 ± 202
FingerSpin	246791 ± 1763	16475 ± 153
FingerTurnEasy	245255 ± 4561	16086 ± 112
FingerTurnHard	245421 ± 4278	16084 ± 69
FishSwim	183750 ± 1773	11591 ± 55
HopperHop	201313 ± 2833	12098 ± 166
HopperStand	201517 ± 3227	12008 ± 255
HumanoidRun	91617 ± 1019	5886 ± 62
HumanoidStand	91927 ± 1004	5893 ± 17
HumanoidWalk	91563 ± 1150	5842 ± 51
PendulumSwingup	724126 ± 21524	32836 ± 178
PointMass	730775 ± 3608	31710 ± 148
ReacherEasy	520021 ± 9637	24888 ± 149
ReacherHard	523441 ± 8012	24874 ± 156
SwimmerSwimmer6	167259 ± 2377	10012 ± 79
WalkerRun	141581 ± 831	6069 ± 48
WalkerStand	140360 ± 1762	6085 ± 29
WalkerWalk	139818 ± 1267	6098 ± 30

Table 4 | Training throughput is displayed for all the DM Control Suite environments on an A100 GPU device across 5 seeds using brax PPO and the RL hyperparameters in Appendix Appendix F. We report the 95th percentile confidence interval.

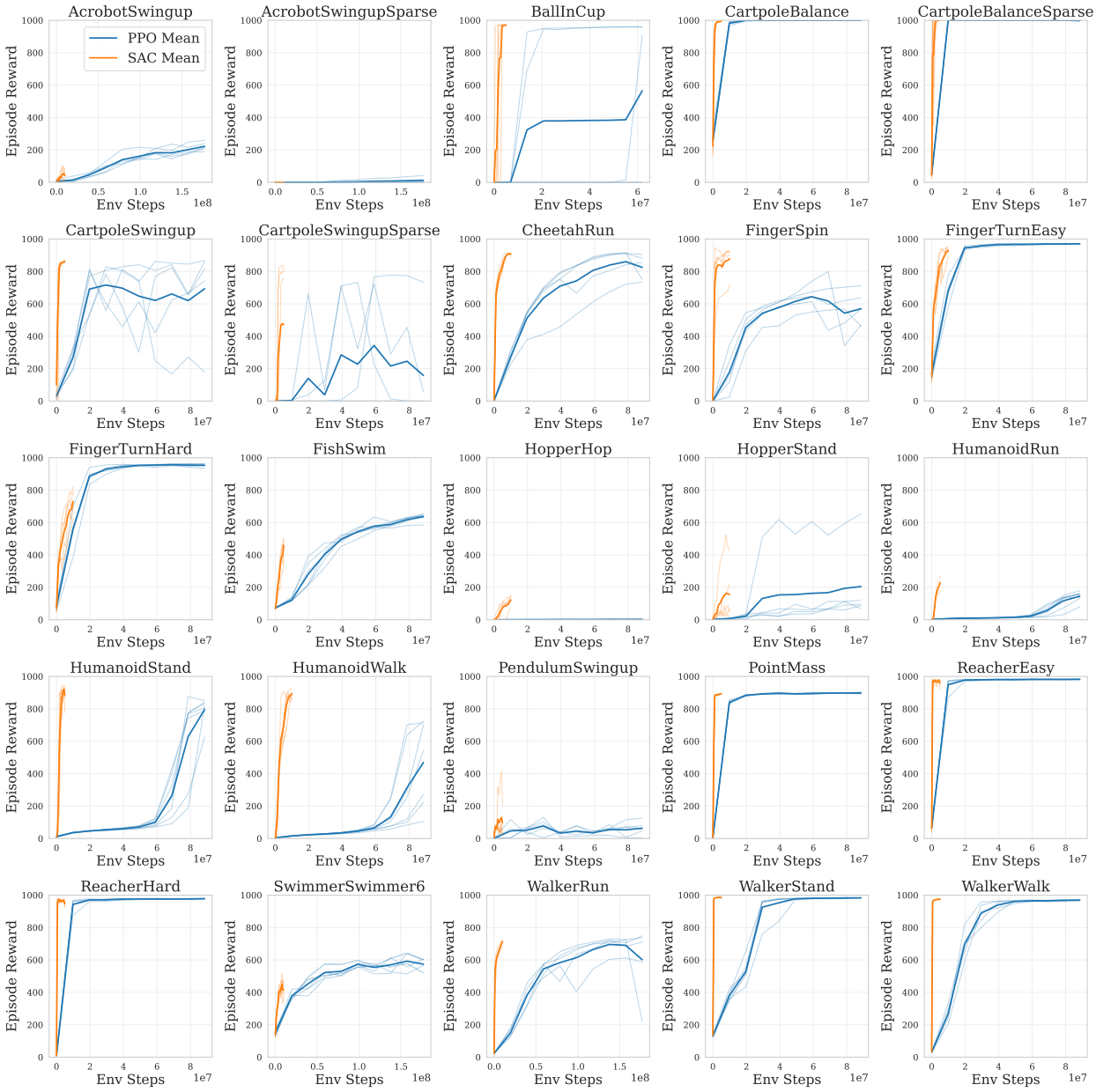

Figure 9 | Reward vs environment steps for PPO and SAC on the full DM Control Suite environments in MuJoCo Playground. We run PPO for 60M steps, with a few selected environments running on 100M steps. SAC runs for 5M steps. All settings are run with 5 seeds on a single A100 GPU device.

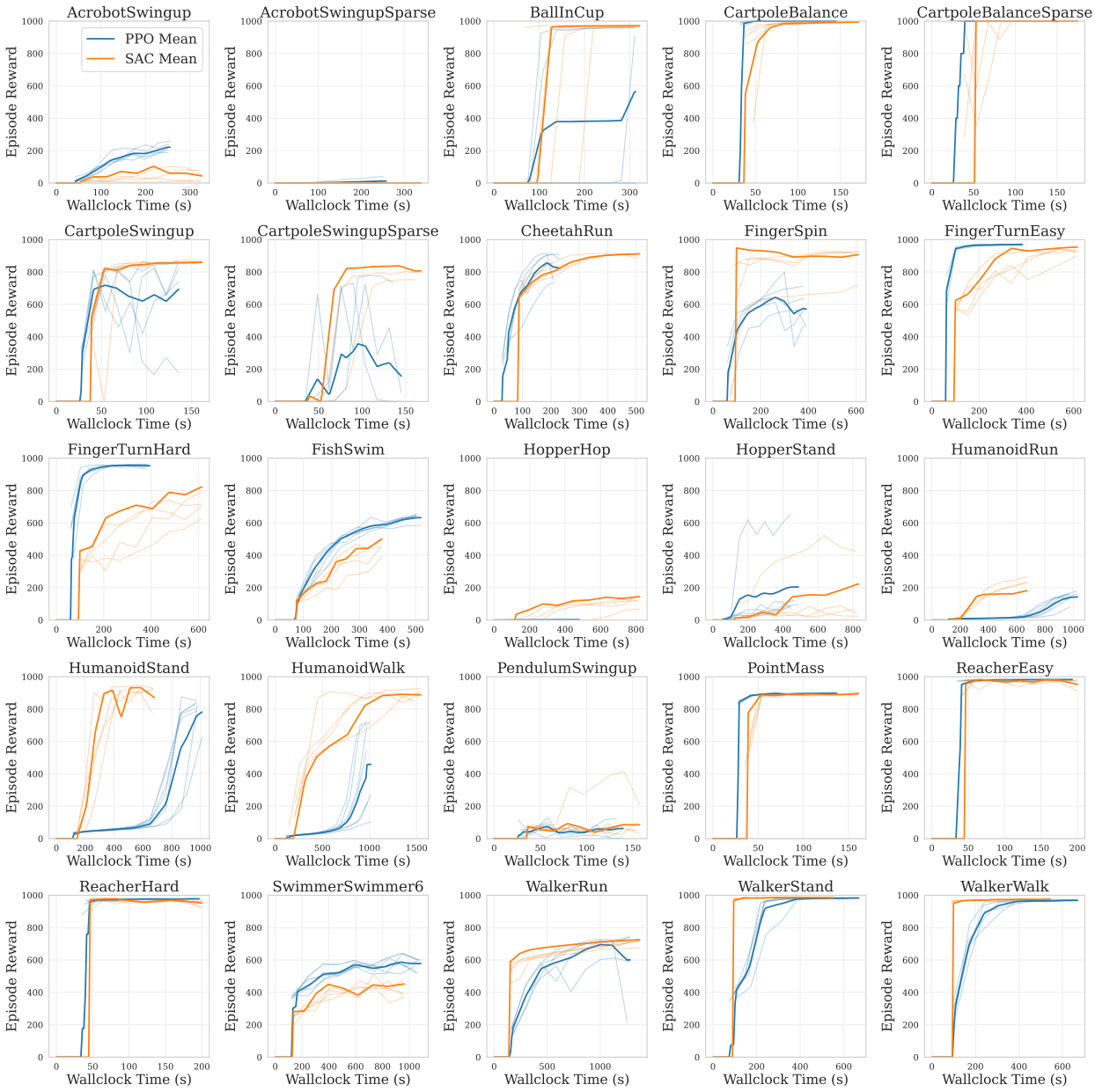

Figure 10 | Reward vs wallclock time for PPO and SAC on the full DM Control Suite environments in MuJoCo Playground. All settings are run with 5 seeds on a single A100 GPU device.

C. Locomotion

C.1. Environment

In Table 5 we show all the locomotion environments available in MuJoCo Playground, broken down by robot platform and available controller.

Robot	Type	Environment
Google Barkour	Quadruped	JoystickFlatTerrain, JoystickRoughTerrain
Berkeley Humanoid	Biped	Joystick
Unitree G1	Biped	Joystick
Unitree Go1	Quadruped	JoystickFlatTerrain, JoystickRoughTerrain, Getup, Handstand, Footstand
Unitree H1	Biped	InplaceGaitTracking, JoystickGaitTracking
OP3	Biped	Joystick
Boston Dynamics Spot	Quadruped	JoystickFlatTerrain, JoystickGaitTracking, Getup

Table 5 | Locomotion environments implemented in MuJoCo Playground by robot platform.

C.2. RL Training Details

C.2.1. Observation and Action

We use a unified observation space across all locomotion environments:

(a) Gravity projected in the body frame,
(b) Base linear and angular velocity,
(c) Joint positions and velocities,
(d) Previous action,
(e) (Optional) User command for joystick-based tasks.

For humanoid locomotion tasks, a phase variable [55] is introduced to shape the gait. This phase variable cycles between $-\pi$ and π for each foot, representing the gait phase. To capture this information effectively, the cos and sin of the phase variable for each foot are included in the observation space. This representation provides a continuous and smooth encoding of the phase, enabling the policy to synchronize its actions with the desired gait cycle.

The action space is defined differently depending on the task. For joystick tasks, we use an *absolute* joint position with a default offset:

$$q_{\text{des},t} = q_{\text{default}} + k_a\, a_t,$$

where k_a is the action scale. For all other tasks, we use a *relative* joint position:

$$q_{\text{des},t} = q_{\text{des},t-1} + k_a\, a_t.$$

The desired joint position is mapped to torque via a PD controller:

$$\tau = k_p\big(q_{\text{des}} - q\big) \;-\; k_d\, \dot{q}, \tag{1}$$

where k_p and k_d are the proportional and derivative gains, respectively.

C.2.2. Domain Randomization

To reduce the sim-to-real gap, we randomize several parameters during training:

- **Sensor noise:** All sensor readings are corrupted with noise.
- **Dynamic properties:** Physical parameters that are difficult to measure precisely (e.g., link center-of-mass, reflected inertia, joint calibration offsets).
- **Task uncertainties:** Ground friction and payload mass are varied randomly.

C.2.3. Reward and Termination

Table 6 | Reward Functions

Reward	**Expression**
Linear Velocity Tracking	$r_v = k_v \exp\left(-\|cmd_{v,xy} - v_{xy}\|^2/\sigma_v\right)$
Angular Velocity Tracking	$r_\omega = k_\omega \exp\left(-\|cmd_{\omega,z} - \omega_z\|^2/\sigma_\omega\right)$
Feet Airtime	$r_{\text{air}} = \text{clip}\left((T_{\text{air}} - T_{\min}) \cdot C_{\text{contact}}, 0, T_{\max} - T_{\min}\right)$
Feet Clearance	$r_{\text{clear}} = k_{\text{clear}} \cdot \|p_{f,z} - p_{f,z}^{\text{des}}\|^2 \cdot \|v_{f,xy}\|^{0.5}$
Feet Phase	$r_{\text{phase}} = k_{\text{phase}} \cdot \exp\left(-\|p_{f,z} - r_z(\phi)\|^2/\sigma_{\text{phase}}\right)$
Feet Slip	$r_{\text{slip}} = k_{\text{slip}} \cdot \|C_{f,i} \cdot v_{f,xy}\|^2$
Orientation	$r_{\text{ori}} = k_{\text{ori}} \cdot \|\phi_{\text{body,xy}}\|^2$
Joint Torque	$r_\tau = k_\tau \cdot \|\tau\|^2$
Joint Position	$r_q = k_q \cdot \|q - q_{\text{nominal}}\|^2$
Action Rate	$r_{\text{rate}} = k_{\text{rate}} \cdot \|a_t - a_{t-1}\|^2$
Energy Consumption	$r_{\text{energy}} = k_{\text{energy}} \cdot \|\dot{q} \cdot \tau\|$
Pose Deviation	$r_{\text{pose}} = k_{\text{pose}} \cdot \exp\left(-\|q - q_{\text{default}}\|^2\right)$
Termination (Penalty)	$r_{\text{termination}} = k_{\text{termination}} \cdot \text{done}$
Stand Still (Penalty)	$r_{\text{standstill}} = k_{\text{standstill}} \cdot \|cmd_{v,xy}\|$
Linear Velocity in Z (Penalty)	$r_{\text{lin\_z}} = k_{\text{lin\_z}} \cdot \|v_z\|^2$
Angular Velocity in XY (Penalty)	$r_{\text{ang\_xy}} = k_{\text{ang\_xy}} \cdot \|\omega_{x,y}\|^2$

In Table 6, $cmd_{v,xy}$ and $cmd_{\omega,z}$ represent the commanded linear velocity in the xy-plane and angular velocity around the z-axis, respectively. v_{xy} and ω_z are the actual linear and angular velocities. T_s and T_a represent the time of the last touchdown and takeoff of the feet. $p_{f,z}$ and $p_{f,z}^{\text{des}}$ denote the actual and desired foot heights, while $v_{f,xy}$ is the horizontal foot velocity. τ is the torque, q is the joint position, and $\dot{q}$ is the joint velocity.

The total reward r_{total} is calculated as the weighted sum of all the reward terms:

$$r_{\text{total}} = \sum_i w_i r_i,$$

Finally, the total reward is clipped to ensure it remains non-negative.

Termination: For joystick-controlled policies, we use a reduced collision model (only the feet) and terminate the episode if the robot inverts (e.g., ends up upside down). For other tasks, we employ the full collision model approximated using geometric primitives.

C.2.4. Network Architecture

We employ an asymmetric actor–critic [46] setup, in which the policy network (actor) and the value network (critic) receive different observation inputs. The policy network is fed with the aforementioned

observations, while the value network additionally receives uncorrupted versions of these signals and extra sensor readings such as contact forces, perturbation forces, and joint torques.

Both the policy and value networks use a three-layer multilayer perceptron (MLP) with hidden sizes of 512, 256, and 128. Each hidden layer uses the Swish [48] activation function. A full set of hyper-parameters is available in Appendix F.

C.2.5. Finetuning

Joystick policy.

1. Train for 100 M timesteps with a command range of $\{1.5, 0.8, 1.2\}$.
2. Finetune for 50 M timesteps with a command range of $\{1.5, 0.8, 2\pi\}$.
3. Finetune on rough terrain for 100 M timesteps.

Getup policy.

1. Train with a power termination cutoff of 400 W.
2. Finetune with a joint velocity cost.

Handstand and footstand policies.

1. Finetune with a joint acceleration and energy cost.
2. Progressively reduce the power termination budget from 400 W to 200 W.

Finally, all policies are trained on flat terrain for 200 M timesteps, then finetuned on rough terrain for 100 M timesteps. The rough terrain is modeled as a heightfield generated from Perlin noise.

C.3. RL Training Results

For all locomotion environments implemented in MuJoCo Playground, we train with PPO using the RL implementation from [14] and we report reward curves below. In Figure 11 we report environment steps versus reward and in Figure 12 we report wallclock time versus reward. All environments are run across 5 seeds on a single A100 GPU.

C.4. RL Training Throughput

In Table 7 we show training throughput for all locomotion envs. In Figure 13 we show training throughput of the Go1JoystickFlatTerrain environment. Different devices and topologies do not make material difference in training wallclock time, since the environment is quite simple with limited contacts between the feet and the floor.

Env	PPO Steps per Second
BarkourJoystick	385920 ± 2162
BerkeleyHumanoidJoystickFlatTerrain	120145 ± 484
BerkeleyHumanoidJoystickRoughTerrain	30393 ± 44
G1Joystick	106093 ± 131
Go1Footstand	204578 ± 906
Go1Getup	96173 ± 230
Go1Handstand	204416 ± 738
Go1JoystickFlatTerrain	417451 ± 2955
Go1JoystickRoughTerrain	291060 ± 727
H1InplaceGaitTracking	289372 ± 1498
H1JoystickGaitTracking	291018 ± 1111
Op3Joystick	198910 ± 406
SpotFlatTerrainJoystick	404931 ± 2710
SpotGetup	266792 ± 1038
SpotJoystickGaitTracking	407572 ± 4091

Table 7 | Training throughput is displayed for all the Locomotion environments on an A100 GPU device across 5 seeds using brax PPO and the RL hyperparameters in Appendix F. We report the 95th percentile confidence interval.

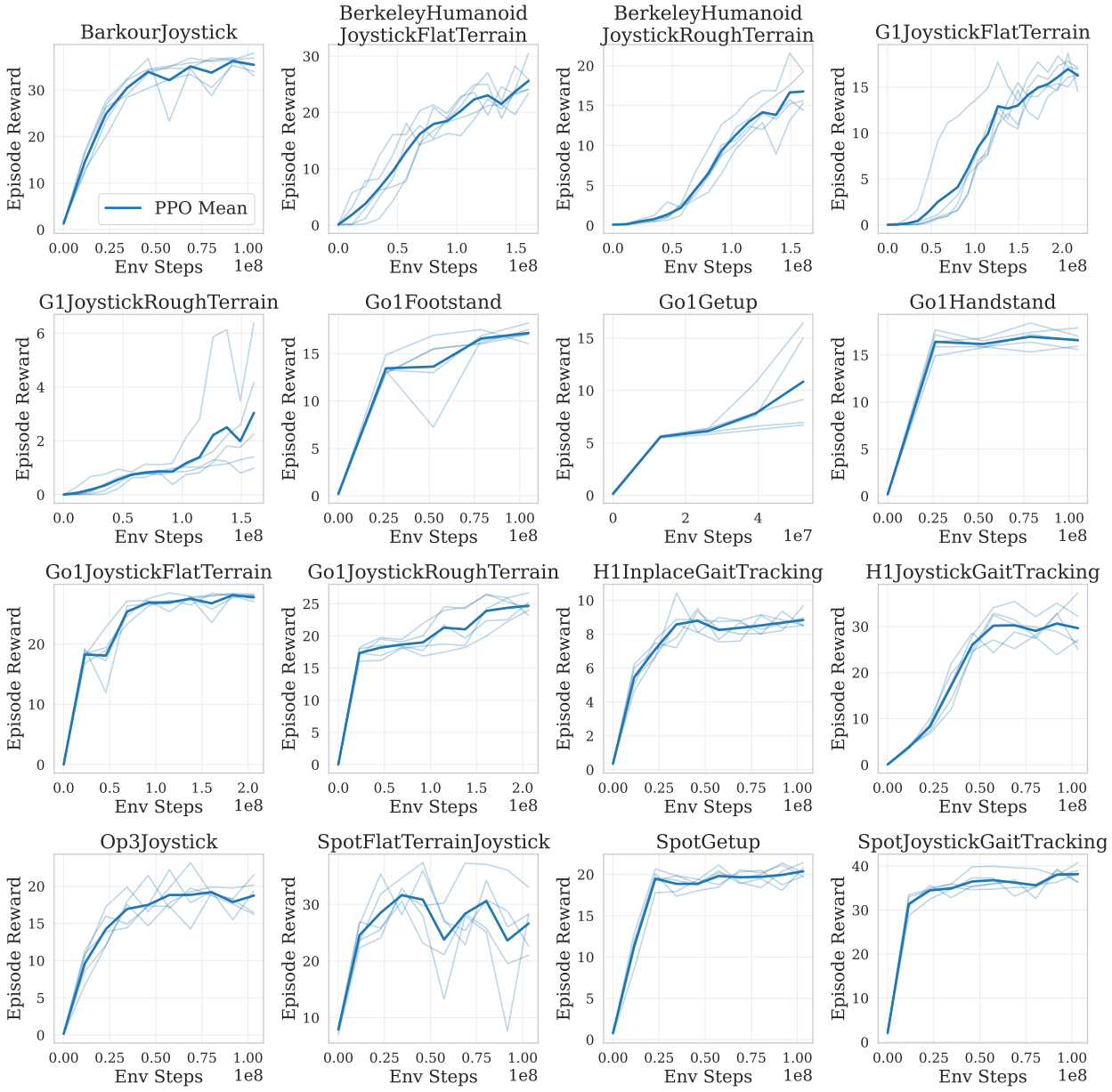

Figure 11 | Reward vs environment steps for Brax PPO. All settings are run with 5 seeds on a single A100 GPU device.

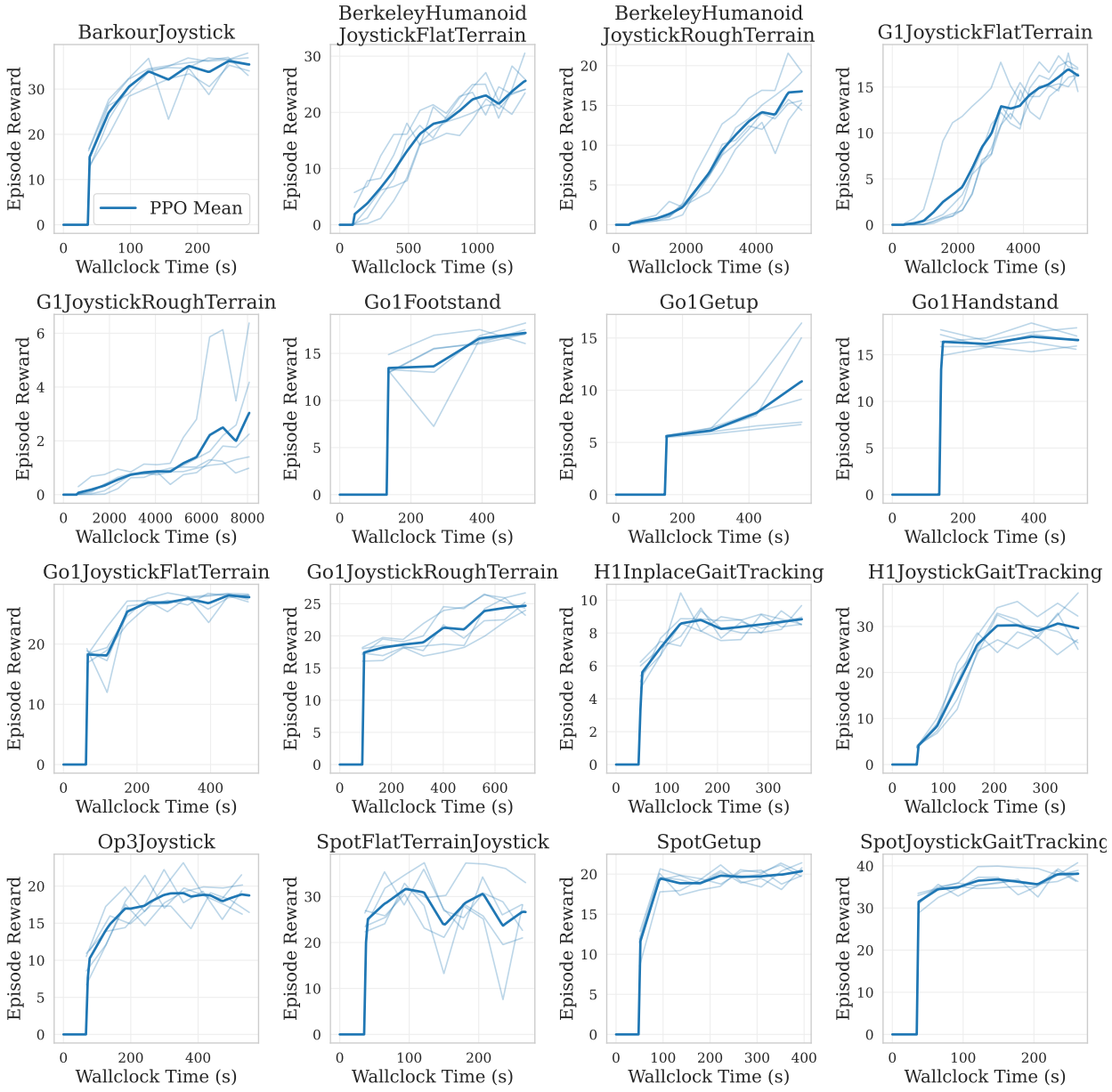

Figure 12 | Reward vs wallclock time for Brax PPO. All settings are run with 5 seeds on a single A100 GPU device. Notice that the initial flat region measures the compilation time for the training + environment code.

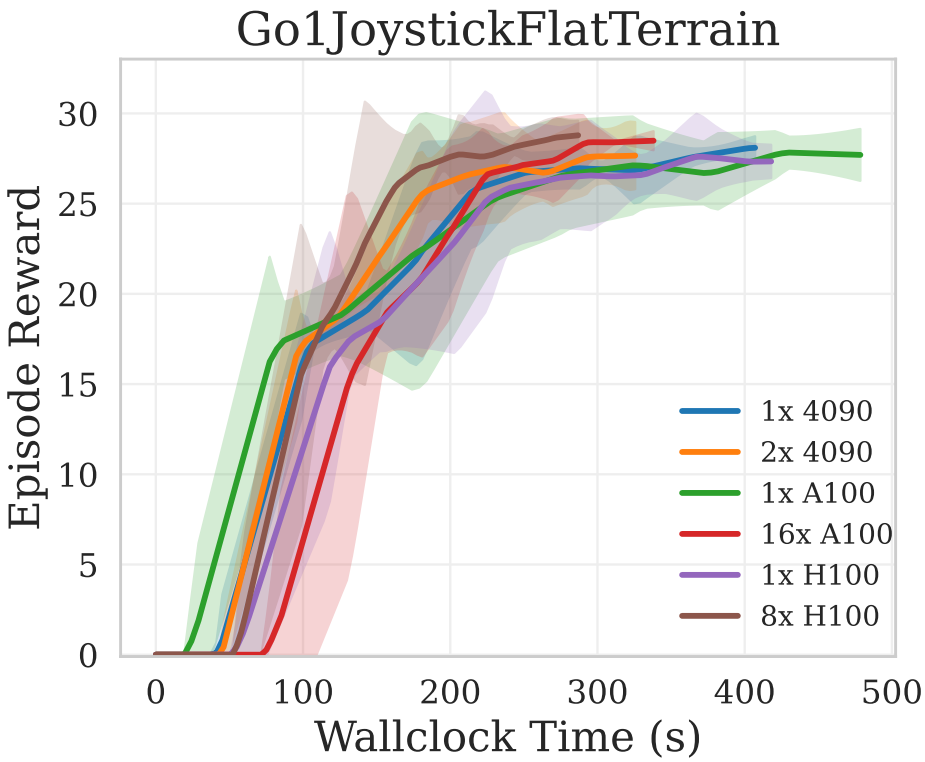

Figure 13 | Training wallclock time for Go1JoystickFlatTerrain on different GPU devices and topologies.

C.5. Real-world Setup

All locomotion deployments are based on `ros2-control` and are written in `C++` with real-time guarantees. The Unitree SDK1, Unitree SDK2, and the Berkeley Humanoid EtherCAT master are each wrapped as abstract sensor and actuator hardware interfaces. These same interfaces are also used in Gazebo [29] to facilitate sim-to-sim verification.

Different RL policies can be loaded and executed within the same process—whether operating on physical hardware or in simulation—by receiving sensor readings and issuing control commands via the hardware interface. Each policy model is inferenced at 50 Hz using ONNX Runtime [12], alongside a model-based estimator. In addition, a separate model-based estimator [13] runs at the hardware interface's maximal communication frequency (500–2000 Hz), providing linear velocity observations and other diagnostic information.

D. Manipulation

D.1. Environments

Robot	Environment
Aloha	SinglePegInsertion
Franka Emika Panda	PickCube, PickCubeOrientation, PickCubeCartesian, OpenCabinet
Franka Emika Panda, Robotiq Gripper	PushCube
Leap Hand	Reorient, RotateZAxis

Table 8 | Manipulation environments implemented in MuJoCo Playground by robot platform.

D.2. RL Training Results

For all manipulation environments implemented in MuJoCo Playground, we train with PPO using the RL implementation from [14] and we report reward curves below. In Figure 14 we report environment steps versus reward and in Figure 15 we report wallclock time versus reward. All environments are run across 5 seeds on a single A100 GPU.

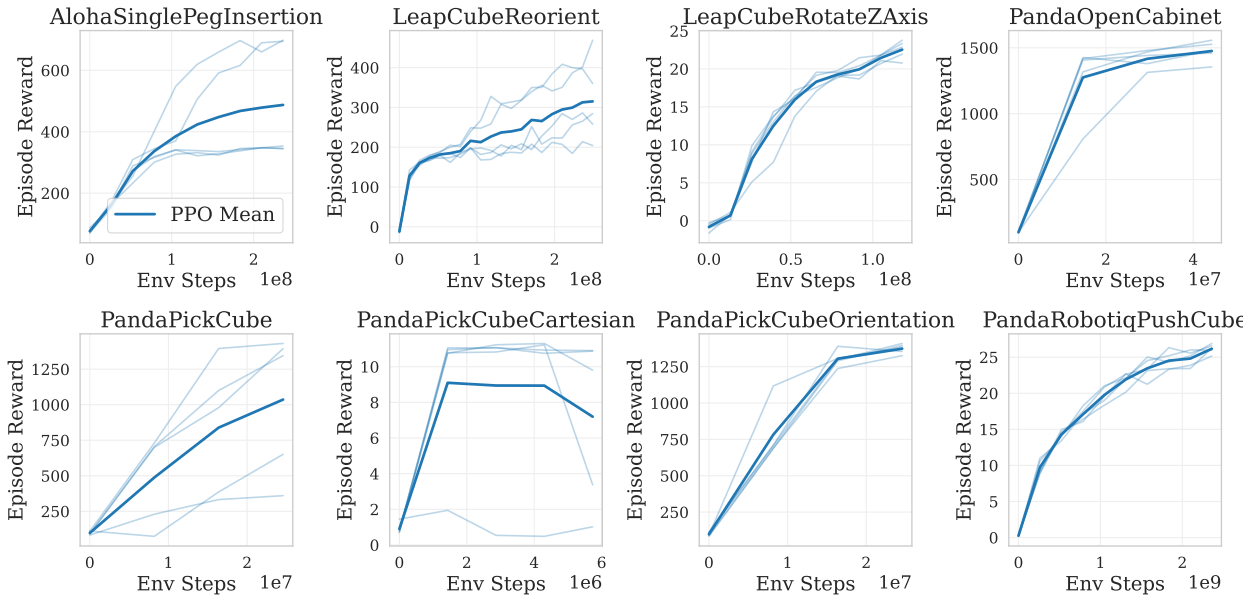

Figure 14 | Reward vs environment steps for brax PPO. All settings are run with 5 seeds on a single A100 GPU device.

D.3. RL Training Throughput

We show RL training throughput for all manipulation environments below in Table 9. In Figure 16 we show reward versus wallclock time on different GPU devices and topologies for the LeapCubeReorient environment.

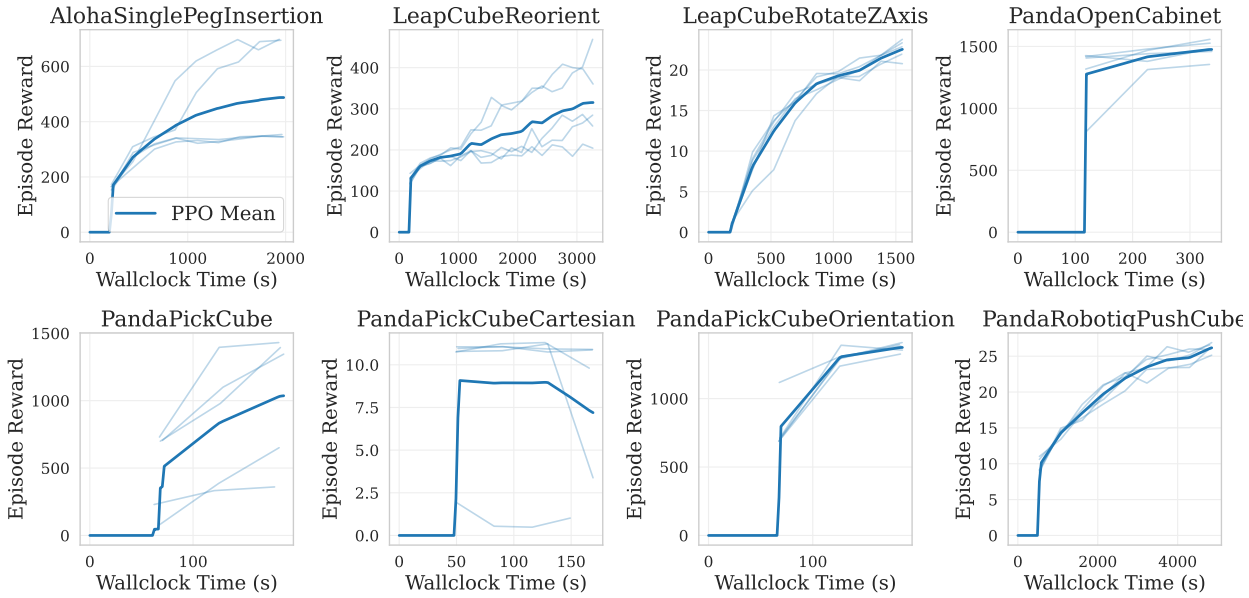

Figure 15 | Reward vs wallclock time for brax PPO. All settings are run with 5 seeds on a single A100 GPU device. Notice that the initial flat region measures the compilation time for the training + environment code.

Env	PPO Steps per Second
AlohaSinglePegInsertion	121119 ± 2159
LeapCubeReorient	76354 ± 143
LeapCubeRotateZAxis	76602 ± 179
PandaOpenCabinet	136007 ± 1553
PandaPickCube	140386 ± 1707
PandaPickCubeCartesian	38015 ± 5302
PandaPickCubeOrientation	140429 ± 1604
PandaRobotiqPushCube	487341 ± 4346

Table 9 | Training throughput is displayed for all the Manipulation environments on an A100 GPU device across 5 seeds using brax PPO and the RL hyperparameters in Appendix F. We report the 95th percentile confidence interval.

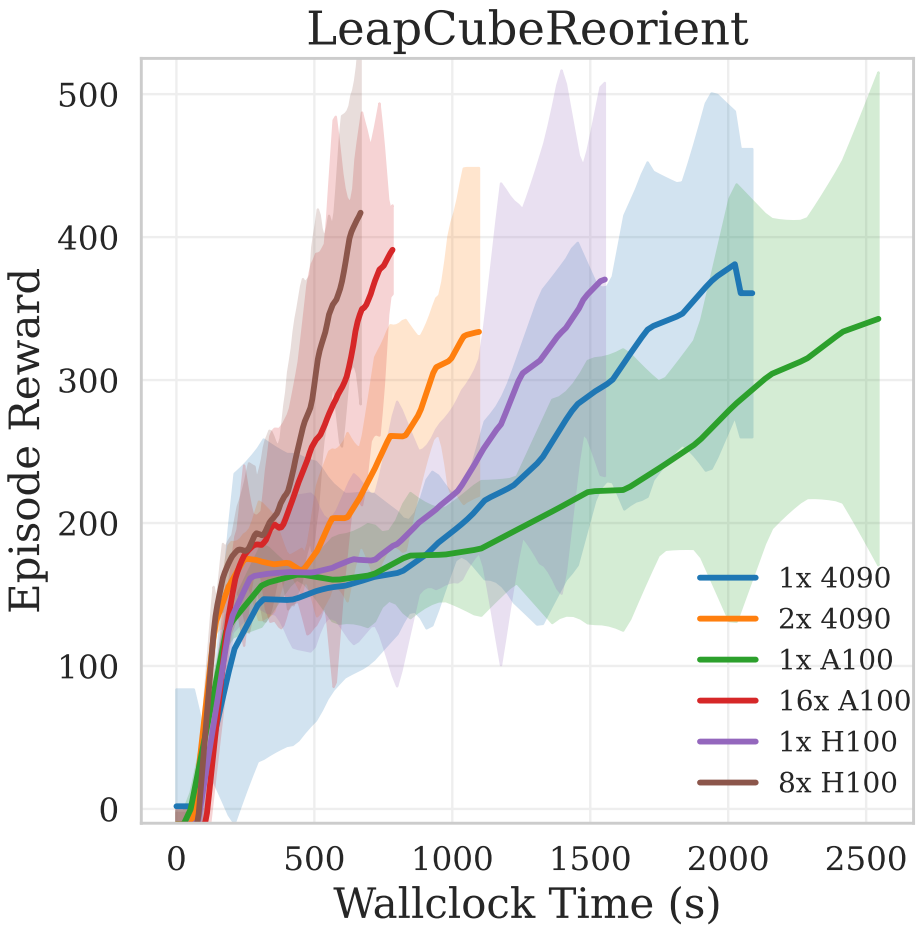

Figure 16 | Training wallclock time for LeapHandReorient on different GPU devices and topologies.

D.4. Real-world Cube Reorientation with a Leap Hand

In this section, we present the technical details of our real-world cube reorientation task using the LEAP Hand, covering the simulation environment, training process, hardware interface, and camera-based object pose estimation.

D.4.1. Simulation Environment

The in-hand reorientation environment is designed to sequentially re-orient a cube within the palm of a robotic hand, without dropping the cube. The cube is initialized randomly above the palm of the hand. The policy then receives the joint angle measurements, estimated cube pose, and previous action. Upon reaching a target orientation within a 0.4 rad tolerance, a new orientation is sampled and the success counter is incremented. We continue re-sampling new target orientations until the cube is dropped or the hand becomes stuck for over 30 s. To avoid trivial adjustments, new orientations are sampled at least 90° away from the previous goal (as in [2, 20]). The cube must reach a target orientation within 0.1 rad (as opposed to 0.4 rad in the real-world setup).

Policy Inputs and Actions. As in locomotion environments, we use an asymmetric actor–critic setup, in which the policy network (actor) and the value network (critic) receive different observation inputs. The policy network is fed observations as outlined below, while the value network additionally receives uncorrupted robot pose, robot velocity, fingertip positions, cube pose, cube velocity, and perturbation forces.

- **Observations** (a) noisy estimates of the robot pose, (b) desired robot pose relative to the current robot pose, (c) noisy estimates of the cube pose, and (d) the previous commanded joint positions.
- **Actions** 16 relative joint positions.

Training Setup. To promote sim-to-real transfer, we apply domain randomization on friction, cube mass, joint offsets, motor friction, reflected inertia, and PD gains, as well as link masses and sensor noise. We also add 2 cm positional and 0.1 rad rotational noise to the cube pose. We conduct two main training phases. During the first 200 M steps, we train without random pose injection and torque limits. We then perform a 100 M-step fine-tuning stage in which we introduce random pose "injections" with a 0.1 probability to mimic "freak-out" moments in real pose estimation (e.g., due to occlusions) and impose torque limits to match the real hardware.

D.4.2. System Identification and Domain Randomization

The original simulation environment for sim-to-real transfer provided by the LEAP Hand [56] does not include robust system identification and instead relies heavily on manual parameter tuning. To improve both the performance and transparency of the system, we performed system identification on the DYNAMIXEL servo actuator used in the hand.

The armature inertia (i.e., rotor inertia reflected through the gearbox) for each joint is: $I_a = k_g^2 I_r$, where $k_g = 288.35$ is the gear ratio from the supplier's data sheet, and $I_r = \frac{1}{2} m_r r_r^2 = 1.7 \times 10^{-8}\,\mathrm{kg\,m^2}$ is the rotor inertia. To obtain I_r, we assumed a uniform mass distribution of the rotor, based on physical disassembly and measurements of the rotor mass ($m_r = 2.0 \times 10^{-3}$ kg) and radius ($r_r = 4.12 \times 10^{-3}$ m).

Because accurately modeling and measuring friction losses is difficult, we set 10% of the maximum torque as the nominal friction value, and employed heavy domain randomization to account for uncertainties.

For training, the servo actuator was controlled using a PD mapping similar to the locomotion setup

in (1). However, during real-world deployment, the control law running on the servo actuator is: $i = k_{\mathrm{p}}^{\mathrm{m}}\left(\theta_{\mathrm{des}}^{\mathrm{m}} - \theta^{\mathrm{m}}\right) - k_{\mathrm{d}}^{\mathrm{m}}\dot{\theta}^{\mathrm{m}}$, where i is the motor current command, and $k_{\mathrm{p}}^{\mathrm{m}}, k_{\mathrm{d}}^{\mathrm{m}}, \theta_{\mathrm{des}}^{\mathrm{m}}, \theta^{\mathrm{m}}$ are expressed in units different from those used in training. To reconcile these discrepancies, we assume $\tau = k_{\mathrm{t}} i$ and carefully compute the mappings based on the motor specifications provided in the data sheet.

Unlike the locomotion setup, the DYNAMIXEL actuator does not perform true current control (i.e., no direct motor current feedback). As a result, the above PD controller may deviate from the ideal behavior. To mitigate this mismatch, we introduce randomization in k_{p} and k_{d} parameters during training.

D.4.3. Real Robot Setup

We deploy the learned policy on the hand using its open-source software, with the following modifications:

- **Control Frequency.** We reduce the policy control frequency from 150 Hz to 20 Hz in both simulation and real-world deployment, due to jitter issues with the low-level USB driver at higher frequencies.
- **System Identification.** We use the same torque (current) limit, stiffness, and damping parameters in training, guided by the system identification results described above.

D.4.4. Vision-based Pose Estimator

We use the vision-based cube pose estimator from [20] in order to solve for the pose of the cube, although any equivalalent method of obtaining the SE(3) camera-to-object transformation would work. Given the local-space 3D coordinates of the cube and the 2D keypoints from the pose estimator, we solve for the camera-to-world transformation. Using camera-to-hand intrinsics calibration, we can then find the hand-to-cube transformation which is used as input to the policy. We run the cube pose estimator at 15FPS. During manipulation, we observe some small jitters or missed detections, but generally it is stable. Despite having access to three cameras on the physical hardware setup, we elect to use only one for simplicity.

D.5. Real-world Non-prehensile Block Reorientation with a Franka-Robotiq Arm

In this section, we provide technical details for our block reorientation task on a real Franka Emika Panda robot with a Robotiq gripper, including the simulation environment, training process, robot hardware interface, and camera-based object pose estimation. Our approach enables reliably learning and deploying a policy for non-prehensile manipulation of a yoga block, requiring only a brief training time in simulation while allowing zero-shot transfer to the real robot.

D.5.1. Simulation Environment

The simulation environment (Figure 17) is designed to reorient a rectangular yoga block within a tabletop workspace region. The block is initialized at a random position and orientation subject to workspace bounds, and is then pushed, slid, or tapped to a desired goal pose at the center of the workspace. The policy uses 7D torque control signals for the robot arm and a fixed, closed Robotiq gripper. We include a simple termination condition when the block leaves the workspace or the end-effector violates safety constraints (e.g., collides with walls or floors).

Key Features.

- **High-frequency torque control at 200 Hz.** Each simulation step is advanced at a high frequency to match the targeted real-world controller rate.
- **Curriculum learning.** We randomize initial joint positions, block poses, latencies in actions and observations, and other environment factors. A progressive curriculum increases the difficulty by gradually expanding the block's displacement and orientation range.
- **Observation Delay.** Both actions and observations are delayed by random amounts at each episode step to approximate real hardware latencies.
- **Reward Shaping.** Shaped rewards encourage the robot to (i) stay near a nominal joint configuration, (ii) minimize velocities, (iii) keep the end-effector near the block, (iv) push the block toward the goal, and (v) orient the block to the desired angle.

Policy Inputs and Actions. As shown in the environment code:

- **Observations** (a) noisy estimates of the block pose, (b) current and recent robot joint positions and velocities, (c) the estimated end-effector pose, and (d) the target block pose.
- **Actions** are 7D torque commands applied at the robot's joints. A constant action for the gripper (fingers closed) is appended for technical reasons in MuJoCo but remains fixed at a configured grasp.

Simulated Environment Details.

- **Gravity Compensation and Torque Bounds.** We configure the MuJoCo model to match the real robot's gravity compensation mode. Torque bounds are set to 8 Nm per joint in simulation, reflecting the approximate safe torque limit on real hardware.
- **Collision Geometry.** The environment enforces collisions with floor, walls, and the block. The Robotiq gripper is held fixed but included for contact modeling.
- **Delayed Observations and Actions.** We adopt random delays (between 1 and 3 steps for actions, and 6 to 12 steps for observations) to emulate real system communication latencies and sensor delay, following best practices in sim-to-real transfer.

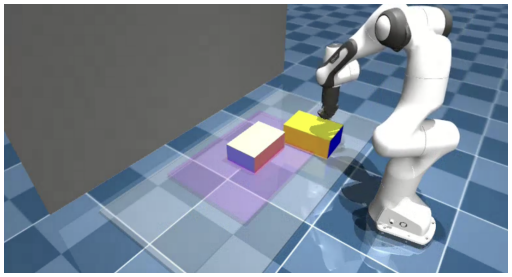

Figure 17 | Example MuJoCo scene of our block reorientation environment. The block is pushed toward the center.

Training Setup. We train the policy on a 16x NVIDIA A100 GPU set-up for ten minutes of wall-clock time, with 3000 steps per episode (with action repeat set to 4, effectively running 750 policy decisions per episode). During training, the block's pose, robot states, and delays are heavily randomized. The final policy was selected from a checkpoint that achieved the highest success rate in the simulator.

D.5.2. Real Robot Setup

We deployed the final trained policy on a Franka Emika Panda manipulator equipped with a Robotiq 2F-85 gripper and an integrated force torque sensor (Robotiq FT-300). Figure 18 illustrates the hardware platform used for our experiments.

Direct Torque Control with Franka FCI. We interface with the robot via the Franka Control Interface (FCI) and send torque commands at 200 Hz:

- **Gravity Compensation.** The Panda is configured to compensate for the arm's own weight. The policy torques therefore focus on regulating the contact interactions with the block, making the system compliant.
- **Bypassing Low-level PID Gains.** We avoid additional position or velocity tracking by sending raw joint torques. This significantly reduces the overhead of tuning any gain schedules and allows the learned policy to directly control contact forces.
- **Safety Considerations.** We define software torque limits and monitor the robot's built-in safety stops and collision detection thresholds. In practice, the learned policy operates well within these limits to gently push the block.

Control and Communication Pipeline. We use a lightweight C++/ROS node that relays torque commands to the Franka FCI at 200 Hz:

- **Policy Node in Python.** Our Python node loads the final trained policy (JIT-compiled for inference speed). At each 5 ms tick, it receives the robot's current joint positions, velocities, and the estimated block pose from ROS topics.
- **Torque Message Publication.** The Python node computes a new 7D torque vector and publishes it as a ROS message to the C++ node. This node directly invokes the FCI's real-time interface to set joint torques.
- **Timing Synchronization.** We maintain a fixed 200 Hz loop, matching the simulator's update frequency. This avoids aliasing or missed steps and ensures that delays in the real system resemble the random delays already modeled in simulation.

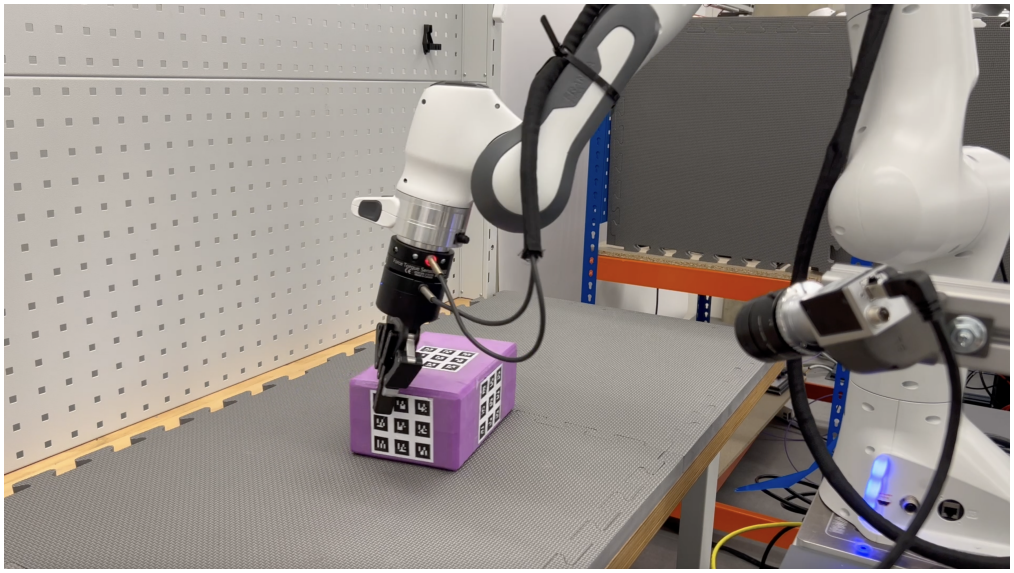

Figure 18 | Real Franka Emika Panda robot with a Robotiq gripper, pushing the yoga block to the goal region.

D.5.3. Camera-based Block Pose Estimation

The policy requires an estimate of the block's 6D pose (position and orientation). We implement a multi-camera setup with four commodity RGB cameras:

- **Intrinsics and Extrinsics.** Each camera is calibrated via OpenCV's standard calibration procedure. We record images of a checkerboard pattern from various viewpoints to obtain precise intrinsic parameters (focal length, principal point) and extrinsic transformations.
- **AR Tag Tracking.** We attach an Alvar [44] fiducial marker to each face of the yoga block. Each camera runs the Alvar pose estimation pipeline. The final block pose is computed as the uniform average of valid detections.
- **Placement Recommendations.** To improve coverage and reduce occlusions, we place two cameras at a lower height (approximately 40 cm above the table) and two cameras overhead (around 80 cm), all aimed toward the center of the workspace, inline with the base of the arm. This diversity of vantage points helps maintain robust tracking, even as the block is manipulated.
- **ROS Integration.** Each camera node publishes pose estimates (with timestamps). A central ROS node fuses these estimates and broadcasts the block pose as a `geometry_msgs/PoseStamped` message at about 30–60 Hz.

Summary. With this environment and training protocol, policies learned in simulation (under domain randomization and fast torque-control loops) exhibit a robust ability to transfer zero-shot to real hardware. Additionally, we encountered several limitations with the policy and workspace. For example, the policy sometimes pushed the block outside the robot's workspace, making it impossible for the robot to reach it. We also observed that early versions of the policy moved the block too quickly, exceeding the robot's force limit and causing it to pause. To address this, we introduced torque penalties, enabling the robot to maintain similar behavior while minimizing force. In summary we found that minimal engineering overhead was needed to align the MuJoCo-based environment with the real robot's dynamic properties, underscoring the effectiveness of torque-based sim-to-real strategies with MuJoCo Playground.

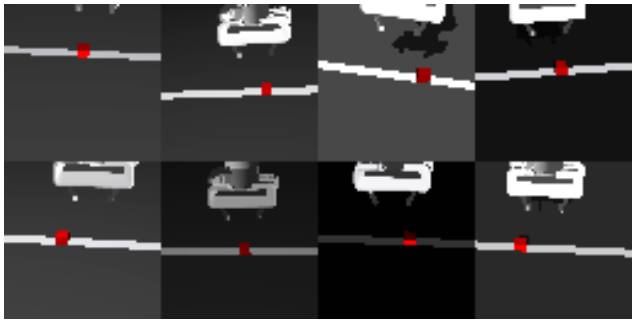

Figure 19 | Policy inputs across domain randomized environments (64x64 pixels each) used while training the deployed PandaPickCubeCartesian agent. Lighting conditions, colors, brightness and camera pose are all randomized.

D.6. Real-world Franka PickCube from Pixels

To highlight the sim-to-real viability of our pixel-based environment, we highlight a robust real-world transfer using the Franka PickCube task.

Task Description. In our PickCube task, the goal of the robot is to move and grasp a 2x2x3 cm upright cube and return it to a fixed target position. To enable robust deployment with a single RGB camera, we limit both the object randomization and the robot's action space to a fixed Y-Z plane. We set the target in simulation to be (x, 0.0, 20.0), where x is set such that both the cube and the gripper initialize are in the same plane. Success is defined in simulation as lifting the object to a target height of 17 cm, and in real experiments as a stable grasp followed by lifting the object at least 10 cm above the table. The object's starting position is randomized along the Y-axis within a 20 cm range centered around 0. Because we train with randomized camera pose and a black background, we lay white tape over the range of possible cube starting positions to allow the memory-less policy to gauge its progress from the grasping site to the target height.

Training. We use a similar reward shaping scheme as [45], using sparse rewards to encourage lifting the cube and bringing it close to the target position and dense rewards to guide the policy search in between. To simplify reward tuning, the dense reward terms only consider progress: $r_t = \text{clip}\left(\sum_i r_{t,i} - \max(r_1, r_2, ..., r_{t-1}), 0\right)$. This helps to emphasize the sparse terms during training. To improve sample efficiency, we terminate the policy upon completion. We train with randomized lighting conditions, colors, brightness and camera pose for robust real-world transfer as shown in Figure 19. Similar to the non-prehensile task in Section D.5.1, we adopt a random delay of 0 to 5 steps for the gripper action, as the real system has a small delay before the grippers begin to close. With an environment step of 50 ms, this results in the agent learning to adapt to a action delay of up to 0.25 s. We find the resulting conservative grasp behaviour to be important for sim-to-real transfer. We disable all except the pair-wise collisions between the gripper fingers and cube to increase simulation throughput.

Agent. Both the agent and critic networks comprise of a standard lightweight CNN architecture [42] followed by two hidden dense layers with size 256. Each channel of the input RGB image is individually normalised per sample by subtracting its mean and dividing it by its standard deviation. The policy network outputs a 3 value action from a single RGB camera looking down towards the gripper (Figure 20). The first two actions are Cartesian increments in the Y and Z directions that is subsequently solved by an inverse kinematics controller [22]. X movement is ignored so that the gripper is restricted to the vertical plane of the block. We discretize the third action dimension to command a closed gripper when the policy value is below zero and an open position when greater than or equal to zero. All values are outputted in the range -1 to 1.

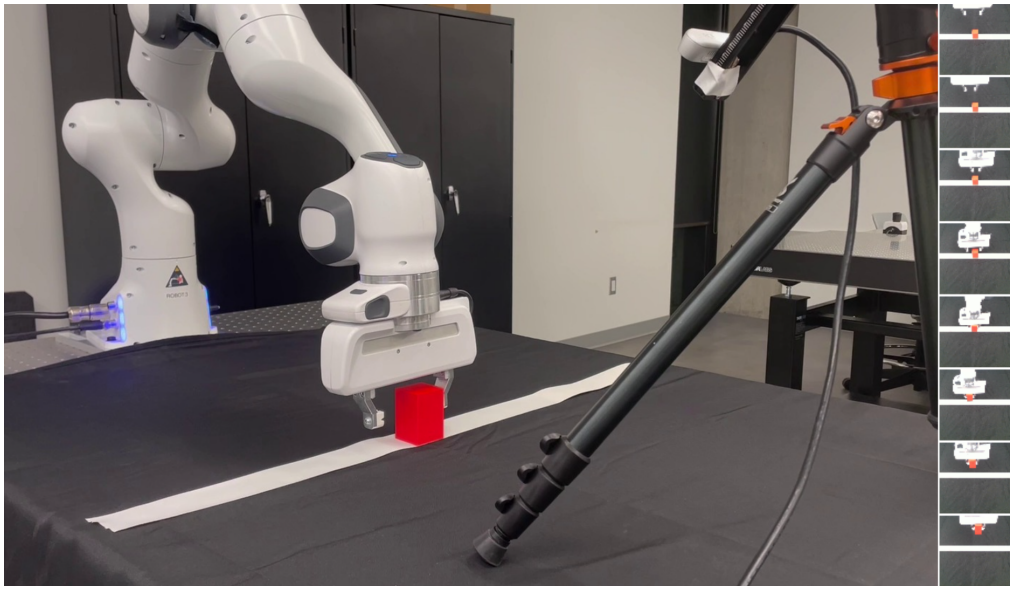

Figure 20 | **Left.** Franka Research robot with a Realsense camera capturing input images. **Right.** Policy inputs from one embodied rollout.

Hardware. We train our deployed policies within ten minutes on a single consumer-grade RTX 4090 GPU paired with a i9-14900KF processor. See Appendix E.1 for training curves for the PandaPickCubeCartesian environment. We deploy on a Franka Research arm with an Intel D435 Realsense camera, using an RTX 3090 GPU for policy inference running at 15Hz.

Control and Communication Pipeline. We use a C++/ROS stack to execute our vision-based policies in real life. Camera images are square-cropped and down-sampled to 64x64 pixels before being passed to a lightweight C++ ONNX ROS Node for inference to produce a Cartesian increment and gripper command. This command is passed to a C++ ROS Node that computes joint commands using the same IK solution as used for training. These commands are output to a final ROS Node that wraps the Franka Control Interface (FCI) to control the robot joints. The control loop runs at 15 Hz, set by the incoming camera stream. We find that sim2real performance drastically improves from roughly calibrating the Cartesian increment scale in our physical setup to the one that the policy was trained on.

E. Madrona Rendering Environments

E.1. RL Training Results

MuJoCo Playground showcases two pixel observation environments using batch rendering; CartPoleBalance and PandaPickCubeCartesian. These two environments include complete training examples using brax PPO. We show the PPO training curves for both environments in Figures 21 and 22 across 5 seeds.

CartPoleBalance is adjusted for pixel-based observations by decreasing the control frequency such that more simulated experience can be factored into training with less policy updates, and by re-adjusting the RL training hyperparameters as necessary. The observations are of dimension 64x64x3 and consist of the current and previous two rendered observations, collapsed into grayscale then transposed for CNN inference. PandaPickCubeCartesian is derived from PandaPickCube. Our changes for faster and more stable pixel-based training are described in Appendix D.6.

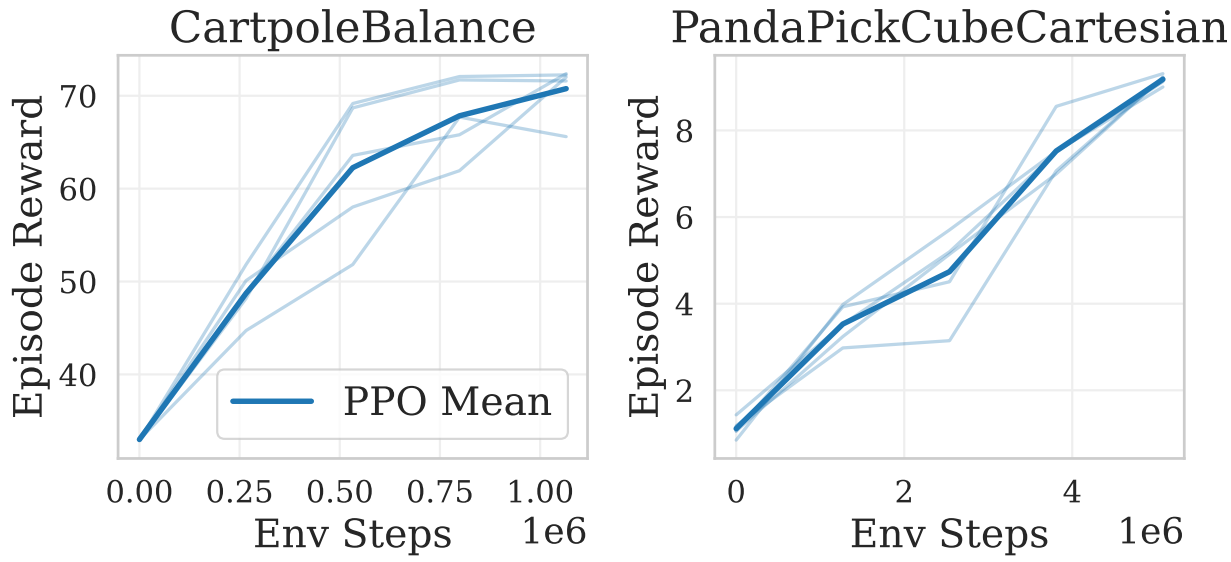

Figure 21 | Reward vs environment steps for brax PPO. All settings are run with 5 seeds on a single RTX 4090 GPU.

E.2. Performance Benchmarking

In this section we benchmark the througput of Madrona MJX GPU batch rendering. For reference, we plot our results alongside those from IsaacLab [41] and Maniskill3 [60]; data for IsaacLab and Maniskill3 is obtained from [60]. This is only a rough comparison as we only take steps to ensure similar hardware and timestep size, as a fully controlled performance benchmark is difficult due to the inherent differences between simulators. Our goal in these comparisons is to only highlight that our batch rendering is competitive with other state-of-the-art simulators that also include high-throughput rendering.

The y-axis of Figure 23 measures the rate of generating environment transitions (s_t, o_t, r_t, s_{t+1}) with random actions, comprising the basic data unit of most on and off-policy training algorithms. The first subplot measures Cartpole simulation with computationally trivial state-based observations. The next three plots show the cost of generating transitions where o_t involves rendering with increasing

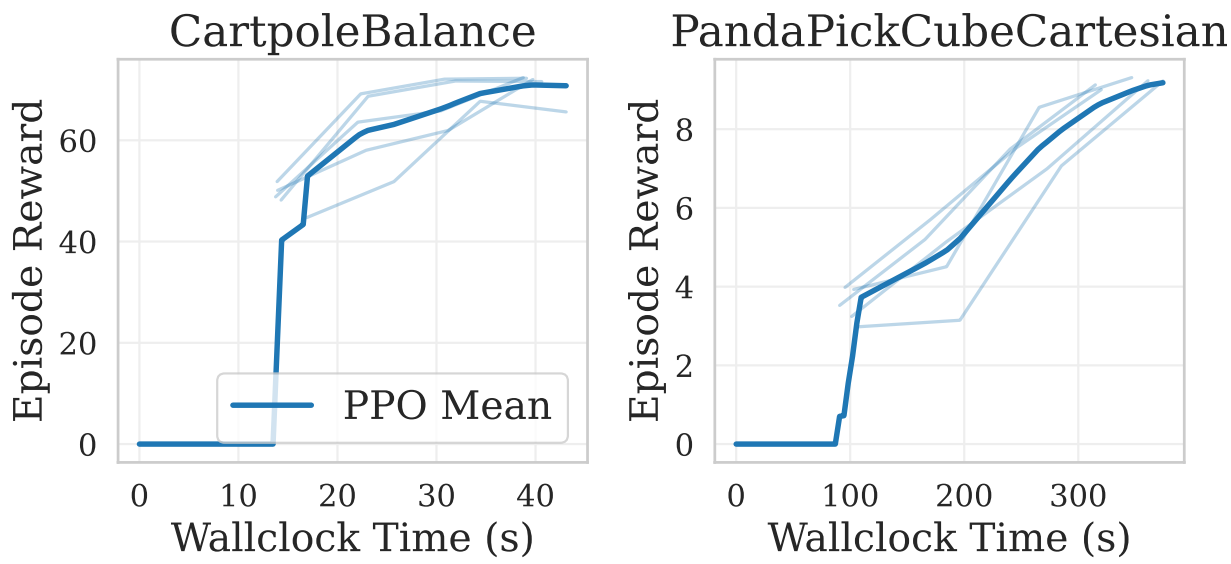

Figure 22 | Reward vs wallclock time for brax PPO. All settings are run with 5 seeds on a single RTX 4090 GPU.

resolution.

Figure 24 evaluates how much of our throughput increase is due to MJX's faster physics step. For each bar, the dark area shows the cost of the physics step and the non-overlapping light area shows the cost of generating the pixel observation. Note that lower values are better, as we display the inverse of frequency. While MJX's faster physics simulation indeed benefits throughput at lower image resolutions, Madrona's rendering speed improvements appear to be the primary driver of the measured speed-ups.

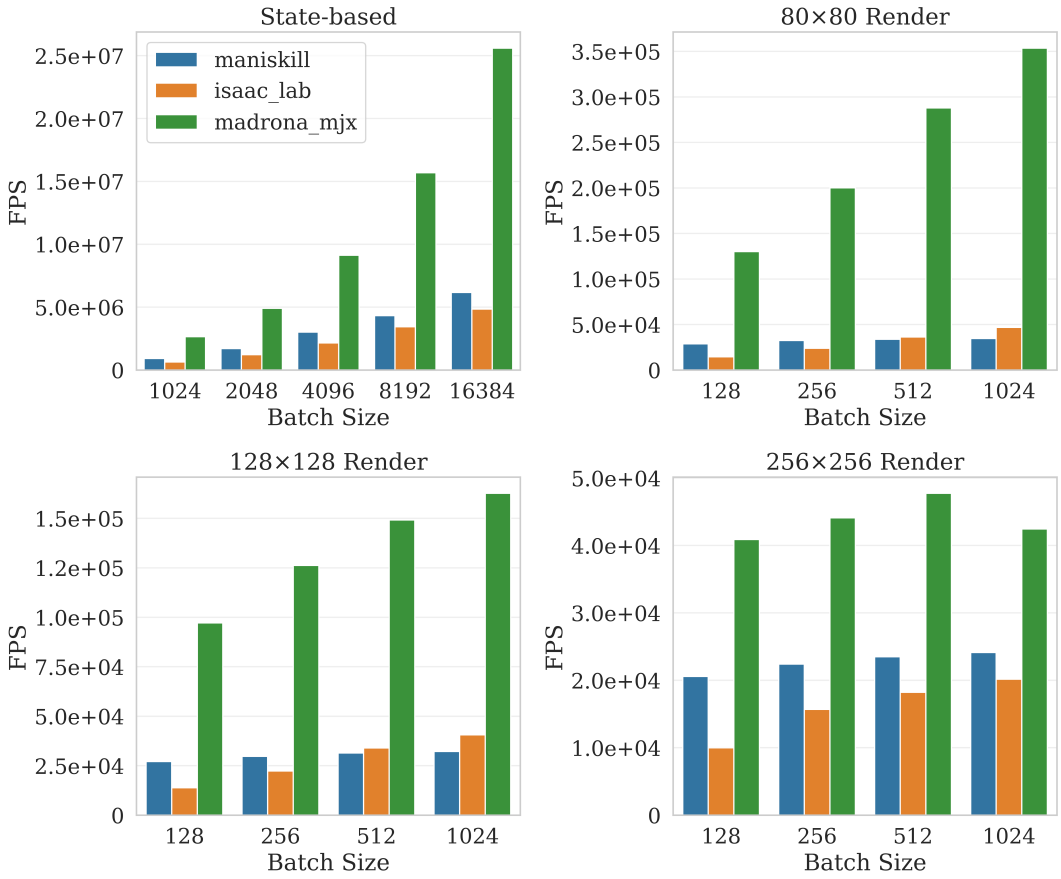

Figure 23 | Comparison of raw environment-stepping throughput with prior simulators for CartpoleBalance with state-based and pixel observations of varying sizes.

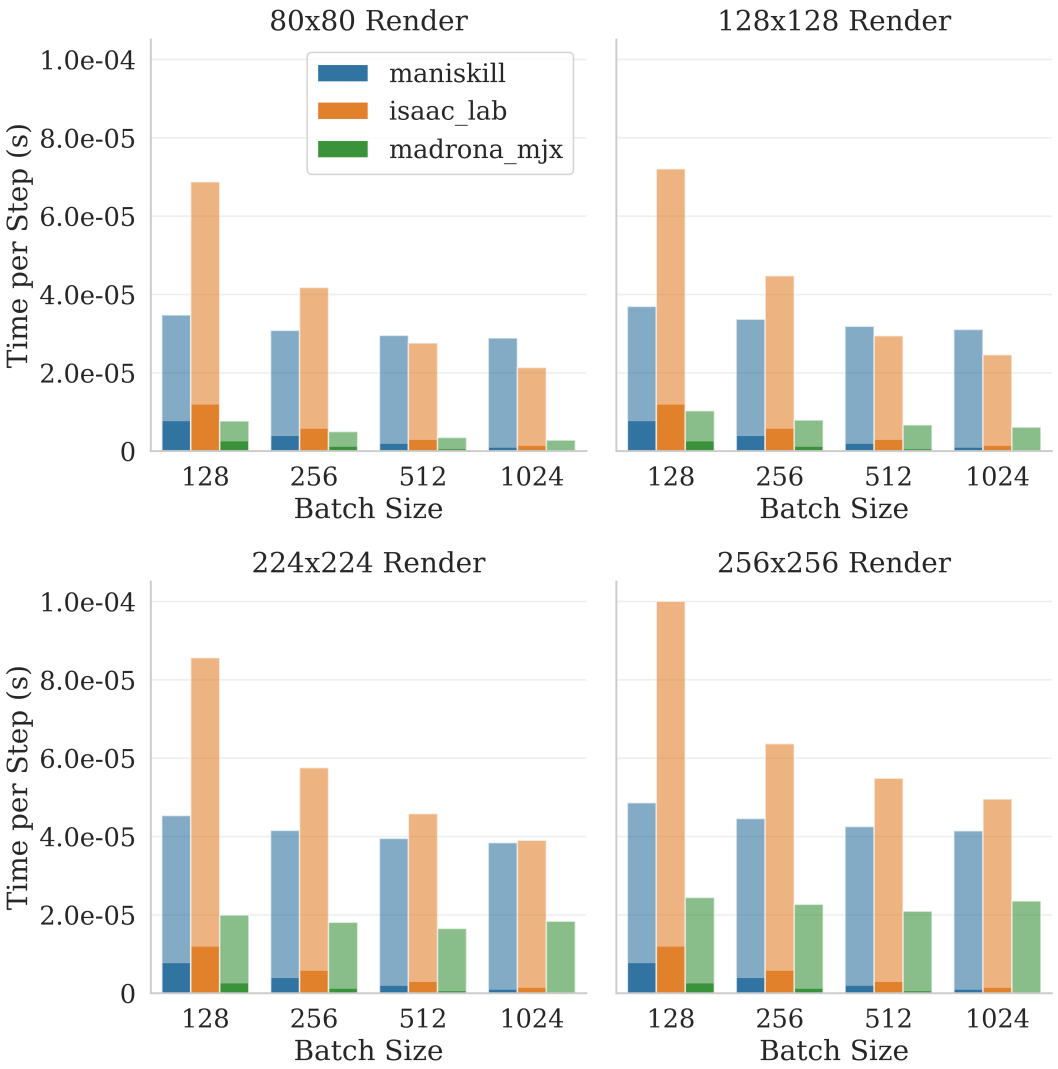

Figure 24 | Time-cost breakdown of unrolling physics simulation and rendering for CartpoleBalance with pixel observations. *Lower is better.* Per-step rendering time is stacked without overlap over physics simulation time.

E.3. Bottlenecks in Pixels-based Training

	Env Step	with Pixels	and Inference	and Training
CartpoleBalance				
FPS	1.37×10^6	4.03×10^5	3.41×10^5	3.13×10^4
Time/Env Step (s)	7.30×10^{-7}	2.48×10^{-6}	2.93×10^{-6}	3.20×10^{-5}
PandaPickCubeCartesian				
FPS	6.40×10^4	3.69×10^4	3.60×10^4	1.56×10^4
Time/Env Step (s)	1.56×10^{-5}	2.71×10^{-5}	2.78×10^{-5}	6.39×10^{-5}

Table 10 | Raw throughput of our two pixel-based environments in various settings. *Env step*: stepping the physics with random actions. *with Pixels*: Same, with the overhead of rendering pixel-based observations. *with Inference*: random actions are replaced with policy inference. *and Training*: the speed of PPO training. Results averaged over 5 runs on an RTX4090.

	Physics	Rendering	Inference	Policy Update
CartpoleBalance				
Time/Env Step (s)	7.30×10^{-7}	1.75×10^{-6}	4.49×10^{-7}	2.91×10^{-5}
Fraction	0.02	0.06	0.01	0.91
PandaPickCubeCartesian				
Time/Env Step (s)	1.56×10^{-5}	1.15×10^{-5}	6.45×10^{-7}	3.62×10^{-5}
Fraction	0.24	0.18	0.01	0.57

Table 11 | Breakdown of total training time by component for CartpoleBalance and PandaPickCube-Cartesian tasks, derived from Table 10. Results averaged over 5 runs on an RTX4090.

Table 11 isolates the contributions of policy rollout (physics simulation, rendering, inference) and policy update to the overall cost per step in a training loop. We amortize the cost of policy update per policy rollout step, setting $t_4 = t_{training} + t_{inference} + t_{rendering} + t_{envstep}$, where t_4 corresponds to *Time/Env Step* in the last column of Table 10. Working in reverse order through the table, we isolate each component. For example, $t_{training} = t_4 - t_3$ corresponds to the Policy Update *Time/Env Step*.

We see that in both of our provided pixel-based environments, the training speed bottleneck is shifted from rendering to policy updates. This is especially true for the Cartpole, as the policy and value architectures includes convolutions determined by the size of the input image regardless of robot and task complexity. The expensive architecture coupled with the trivial embodiment, shift over 90% of the training burden to network updates. For the Franka Panda environment, we buffer more of the computation into the physics by training with a lower control frequency. At 20 Hz control with a 5ms physics timestep, the policy makes only one decision per 10 simulator sub-steps. Similar to Cartpole, rendering is less of a bottleneck than the cost of processing the resultant images via convolutional-based network architectures.

F. Reinforcement Learning Hyper-parameters

In this section, we report the hyper-parameters used to train RL policies for all environments in MuJoCo Playground.

F.1. DM Control Suite

Hyperparameter	Default Value	Environment-Specific Modifications
num_timesteps	60,000,000	AcrobotSwingup, Swimmer, WalkerRun: 100,000,000
num_evals	10	
reward_scaling	10.0	
normalize_observations	True	
action_repeat	1	PendulumSwingUp: 4
unroll_length	30	
num_minibatches	32	
num_updates_per_batch	16	PendulumSwingUp: 4
discounting	0.995	BallInCup: 0.95, FingerSpin: 0.95
learning_rate	1e-3	
entropy_cost	1e-2	
num_envs	2048	
batch_size	1024	

Table 12 | Brax PPO hyperparameters.

Hyperparameter	Default Value
madrona_backend	True
wrap_env	False
num_timesteps	1,000,000
num_evals	5
reward_scaling	0.1
normalize_observations	True
action_repeat	1
unroll_length	10
num_minibatches	8
num_updates_per_batch	8
discounting	0.97
learning_rate	5e-4
entropy_cost	5e-3
num_envs	1024
num_eval_envs	1024
batch_size	256

Table 13 | Brax PPO hyperparameters for vision-based environments.

Hyperparameter	Default Value	Environment-Specific Modifications
num_timesteps	5,000,000	Acrobot, Swimmer, Finger, Hopper, CheetahRun, HumanoidWalk, PendulumSwingUp, WalkerRun: 10,000,000
num_evals	10	
reward_scaling	1.0	
normalize_observations	True	
action_repeat	1	PendulumSwingUp: 4
discounting	0.99	
learning_rate	1e-3	
num_envs	128	
batch_size	512	
grad_updates_per_step	8	
max_replay_size	1048576 * 4	
min_replay_size	8192	
network_factory.q_network_layer_norm	True	

Table 14 | Brax SAC hyperparameters.

F.2. Locomotion

Hyperparameter	Default Value
num_timesteps	100,000,000
num_evals	10
reward_scaling	1.0
normalize_observations	True
action_repeat	1
unroll_length	20
num_minibatches	32
num_updates_per_batch	4
discounting	0.97
learning_rate	3e-4
entropy_cost	1e-2
num_envs	8192
batch_size	256
max_grad_norm	1.0
policy_hidden_layer_sizes	(128, 128, 128, 128)
policy_obs_key	"state"
value_obs_key	"state"

Table 15 | Default Brax PPO hyperparameters.

Hyperparameter	Value
num_timesteps	200,000,000
num_evals	10
num_resets_per_eval	1
policy_hidden_layer_sizes	(512, 256, 128)
value_hidden_layer_sizes	(512, 256, 128)
value_obs_key	"privileged_state"

Table 16 | Brax PPO hyperparameters specific to Go1JoystickFlatTerrain and Go1JoystickRoughTerrain.

Hyperparameter	Value
num_timesteps	100,000,000
num_evals	5
policy_hidden_layer_sizes	(512, 256, 128)
value_hidden_layer_sizes	(512, 256, 128)
value_obs_key	"privileged_state"

Table 17 | Brax PPO hyperparameters specific to Go1Handstand and Go1Footstand.

Hyperparameter	Value
num_timesteps	200,000,000
num_evals	10
discounting	0.95
policy_hidden_layer_sizes	(512, 256, 128)
value_hidden_layer_sizes	(512, 256, 128)
value_obs_key	"privileged_state"

Table 18 | Brax PPO hyperparameters specific to Go1Backflip.

Hyperparameter	Value
num_timesteps	50,000,000
num_evals	5
policy_hidden_layer_sizes	(512, 256, 128)
value_hidden_layer_sizes	(512, 256, 128)
value_obs_key	"privileged_state"

Table 19 | Brax PPO hyperparameters specific to Go1Getup.

Hyperparameter	Value
num_timesteps	400,000,000
num_evals	16
num_resets_per_eval	1
reward_scaling	0.1
unroll_length	32
num_updates_per_batch	5
discounting	0.98
learning_rate	1e-4
entropy_cost	0
num_envs	32768
batch_size	1024
clipping_epsilon	0.2
policy_hidden_layer_sizes	(512, 256, 64)
value_hidden_layer_sizes	(256, 256, 256, 256)
value_obs_key	"privileged_state"

Table 20 | Brax PPO hyperparameters specific to G1Joystick.

Hyperparameter	Value
num_timesteps	100,000,000
num_evals	10
num_resets_per_eval	1
clipping_epsilon	0.2
discounting	0.99
learning_rate	1e-4
entropy_cost	0.005
policy_hidden_layer_sizes	(512, 256, 128)
value_hidden_layer_sizes	(512, 256, 128)
value_obs_key	"privileged_state"

Table 21 | Brax PPO hyperparameters specific to Berkeley Humanoid.

F.3. Manipulation

Hyperparameter	Default Value
normalize_observations	True
reward_scaling	1.0
policy_hidden_layer_sizes	(32, 32, 32, 32)
policy_obs_key	"state"
value_obs_key	"state"

Table 22 | Default Brax PPO hyperparameters.

Hyperparameter	Value
num_timesteps	150,000,000
num_evals	10
unroll_length	40
num_minibatches	32
num_updates_per_batch	8
discounting	0.97
learning_rate	3e-4
entropy_cost	1e-2
num_envs	1024
batch_size	512
policy_hidden_layer_sizes	(256, 256, 256, 256)

Table 23 | Brax PPO hyperparameters for AlohaSinglePegInsertion.

Hyperparameter	Value
num_timesteps	40,000,000
num_evals	4
unroll_length	10
num_minibatches	32
num_updates_per_batch	8
discounting	0.97
learning_rate	1e-3
entropy_cost	2e-2
num_envs	2048
batch_size	512
policy_hidden_layer_sizes	(32, 32, 32, 32)
num_resets_per_eval	1

Table 24 | Brax PPO hyperparameters for PandaOpenCabinet.

Hyperparameter	Value
num_timesteps	5,000,000
num_evals	5
unroll_length	10
num_minibatches	8
num_updates_per_batch	8
discounting	0.97
learning_rate	5.0e-4
entropy_cost	7.5e-3
num_envs	1024
batch_size	256
reward_scaling	0.1
policy_hidden_layer_sizes	(256, 256)
num_resets_per_eval	1
max_grad_norm	1.0

Table 25 | Brax PPO hyperparameters for PandaPickCubeCartesian.

Hyperparameter	Value
num_timesteps	20,000,000
num_evals	4
unroll_length	10
num_minibatches	32
num_updates_per_batch	8
discounting	0.97
learning_rate	1e-3
entropy_cost	2e-2
num_envs	2048
batch_size	512
policy_hidden_layer_sizes	(32, 32, 32, 32)

Table 26 | Brax PPO hyperparameters for PandaPickCube.

Hyperparameter	Value
num_timesteps	2,000,000,000
num_evals	10
unroll_length	100
num_minibatches	32
num_updates_per_batch	8
discounting	0.994
learning_rate	6e-4
entropy_cost	1e-2
num_envs	8192
batch_size	512
num_resets_per_eval	1
num_eval_envs	32
policy_hidden_layer_sizes	(64, 64, 64, 64)

Table 27 | Brax PPO hyperparameters for PandaRobotiqPushCube.

Hyperparameter	Value
num_timesteps	100,000,000
num_evals	10
num_minibatches	32
unroll_length	40
num_updates_per_batch	4
discounting	0.97
learning_rate	3e-4
entropy_cost	1e-2
num_envs	8192
batch_size	256
num_resets_per_eval	1
policy_hidden_layer_sizes	(512, 256, 128)
value_hidden_layer_sizes	(512, 256, 128)
policy_obs_key	"state"
value_obs_key	"privileged_state"

Table 28 | Brax PPO hyperparameters for LeapCubeRotateZAxis).

Hyperparameter	Value
num_timesteps	100,000,000
num_evals	20
num_minibatches	32
unroll_length	40
num_updates_per_batch	4
discounting	0.99
learning_rate	3e-4
entropy_cost	1e-2
num_envs	8192
batch_size	256
num_resets_per_eval	1
policy_hidden_layer_sizes	(512, 256, 128)
value_hidden_layer_sizes	(512, 256, 128)
policy_obs_key	"state"
value_obs_key	"privileged_state"

Table 29 | Brax PPO hyperparameters for LeapCubeReorient.

Hyperparameter	Value
madrona_backend	True
wrap_env	False
normalize_observations	True
reward_scaling	1.0
policy_hidden_layer_sizes	(32, 32, 32, 32)
num_timesteps	5,000,000
num_evals	5
unroll_length	10
num_minibatches	8
num_updates_per_batch	8
discounting	0.97
learning_rate	5.0e-4
entropy_cost	7.5e-3
num_envs	1024
batch_size	256
reward_scaling	0.1
num_resets_per_eval	1

Table 30 | Brax PPO hyperparameters for vision-based PandaPickCubeCartesian.