Real-World Reinforcement Learning on a Generalist Robot Policy

Lawrence Qiu

December 18, 2025

Abstract

Developing robots that can follow commands and perform a wide variety of tasks represents significant research and commercial value. However, today's generalist robot policies, typically trained by imitating diverse human demonstrations, are limited by the speed and reliability at which they can complete tasks. Reinforcement learning (RL) techniques directly address these challenges but are difficult to apply to real-world policies for several reasons. From-scratch reinforcement learning methods are typically too sample inefficient to apply in the real world without severely limited action spaces. An alternative is to perform learning in simulation and then to transfer the policy to the real world, but recreating complex environments and stabilizing knowledge transfer is still an active area of research. For my final project, I aim to explore a third direction towards real-world RL: taking inspiration from the language domain, I utilize an existing pre-trained generalist robot model and fine-tune it on my own task before performing a gradient policy method on it to improve performance. With this technique, I show that reinforcement learning can be applied in real-world environments and result in practical improvements.

1 Introduction

Current robotic systems deployed in industrial and commercial use-cases are domain-specific, capable of handling only a single or a couple of tasks, with each task requiring its own programming and tuning. Humans, on the other hand, can perform a wide variety of tasks even without prior experience, intelligently learning and reasoning to navigate around barriers. Developing robotic systems that can exhibit this kind of versatility in behavior represents significant research and commercial value, enabling economic abundance as well as novel insights into how humans work.

The backbone of current generalist robotic systems are vision-language-action models, or VLAs. They are a relatively new class of models that combine the language-following capabilities of large language models (LLMs) with the robot control capabilities of robot policies. Some recently released VLAs include OpenVLA (2024) [1], Physical Intelligence's π_0 (2024) [2], and NVIDIA's GR00T N1 (2025) [3]. Typically, these models are trained by taking existing language models pre-trained on internet-scale data, such as Meta's Llama and Google's Gemma, and attaching an action head to generate robot behaviors. Modern VLAs have already demonstrated significant progress towards the generalist dream: Physical Intelligence's π_0 can fold shirts, bag groceries, and bus tables [2]. However, task completion speed and reliability are still far behind human counterparts, limiting applications in practice.

Reinforcement learning (RL) represents techniques that can address these challenges. By exploring and learning from the environment, agents can optimize their behavior and improve performance. However, RL is difficult to apply to real-world environments because of the inherent sample inefficiency of most techniques. Without severely limited action spaces that prevent the policy from learning dexterous and complex behaviors, RL algorithms typically take unreasonable amounts of time to converge. One method of getting around this limitation is to perform the RL in simulation and then to transfer the learned weights to the real world, but reproducing complex environments and enabling stable and consistent transfer is still an open research problem.

For my final project, I explore a different approach, inspired by methods in the language domain. Rather than initializing the model from scratch, language-domain reinforcement learning is typically done on an already strong baseline model, first pre-trained on internet-scale data before fine-tuned on task-specific text. The pretraining provides the model with a baseline understanding of human language, the fine-tuning

biases the model towards language traces that are conducive to the task, and the best options are finally amplified by the reinforcement learning step. Similarly, pretraining a robot policy on a wide variety of robot data is expected to allow the policy to distinguish reasonable action trajectories over random meaningless behaviors, the fine-tuning step should bias those trajectories towards the ones required for the tasks and robot embodiment at hand, and the reinforcement learning step refines and improves those behaviors. Table 1 compares these steps in the language and robot domains.

	Pre-training	**Fine-tuning**	**Reinforcement Learning**
Language Modeling	Supervised training on internet-scale language data. Examples: OpenAI GPT Series; Meta Llama Base	Supervised fine-tuning on specific task (usually chatbot conversation). Examples: OpenAI ChatGPT; Meta Llama Instruct	Reinforcement learning with correct answers rewarded. Examples: OpenAI o Series; DeepSeek R1
Robotics Control	Supervised training on thousands of hours of robot data across many robot embodiments. Examples: Physical Intelligence π_0 [2]; NVIDIA GR00T N1 [3]; OpenVLA [1]	Supervised fine-tuning for a specific robot across limited tasks. Examples: Fine-tuned versions of previous models	Reinforcement learning with successful tasks awarded. Examples: Limited simulation research

Table 1: Pre-training, fine-tuning, and reinforcement-learning stages for the language modeling domain and the robotics control domain, with select commercial and open-source examples.

In my final project, I implement these three stages on a real-world robot policy to demonstrate how reinforcement learning can be applied in the real world.

2 Method

2.1 Policy Model Selection

The specific robot policy model I went with for this project is π_0. π_0 is an open-source 3-billion-parameter VLA model created by the venture-backed San-Francisco startup Physical Intelligence [2]. The model consists of Google's PaliGemma vision-language model (VLM) [4] attached to a flow-matching-based action generator (see Figure 1). Inference is performed by submitting images from up to three cameras as well as a task description, from which the model generates a vector of joint angles for a specified horizon (up to a few seconds). The model is pre-trained on a proprietary 10,000-hour dataset collected from 7 robots performing 68 different tasks, augmented with a subset of the open-source Open X-Embodiment (OXE) dataset [5]. After training, the model can perform advanced tasks such as folding a shirt, clearing a table, and taking toast out of a toaster [2].

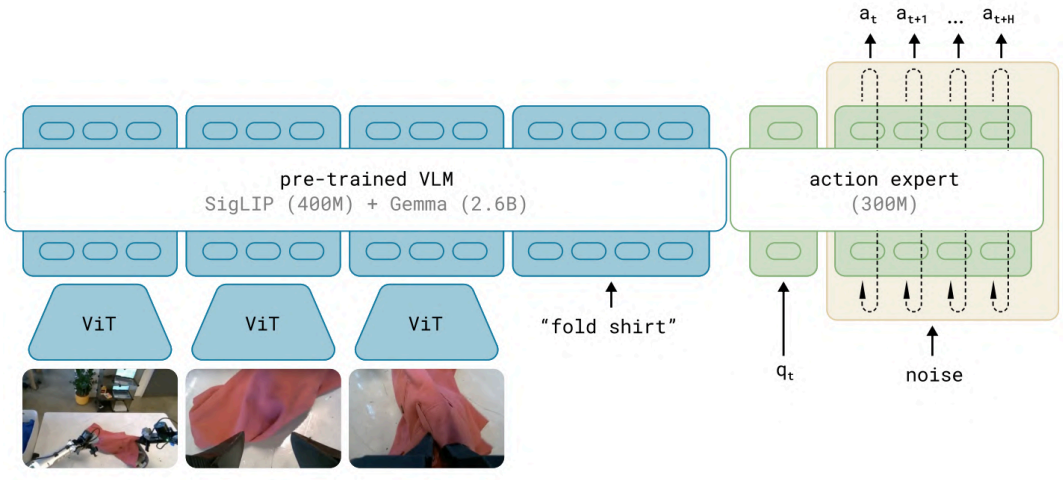


Figure 1: Architecture of the π_0 model. The model combines the PaliGemma VLM (SigLIP + Gemma) [4] with a flow-matching-based action generator. Figure taken from Black et al. [2].

There are several advantages towards using the π_0 model. Compared to older models like OpenVLA, the model is state-of-the-art and incorporates architectural improvements such as the flow-based action generator which enables high-frequency dexterous behaviors. The training set is large and high quality compared to alternative models, and there is extensive documentation detailing how it was created and trained. The most significant factor, however, is that both the model weights and the training and inference codebase are open source, enabling the architectural changes that are necessary for this project.

2.2 Infrastructure Details

However, working with a SOTA model also has some drawbacks. The most significant disadvantage is that the model is large: with over 3 billion parameters, training and inference requires significant hardware considerations. Most consumer hardware cannot perform inference of models this large at a reasonable speed, and nearly all do not have the necessary video memory to even load the model for training.

My solution to this problem is to perform the model training and inference remotely, on a server with the necessary capabilities. Specifically, I used the Tufts High-Performance Computing (HPC) cluster, which had a recent upgrade of 32 NVIDIA H200 GPUs. These top-of-the-line GPUs meet the requirements for training and inference, making this project possible.

However, using a remote server for compute adds additional complexity to the system architecture. Inference requires low-latency communication between the two systems and training requires high-throughput communication to transfer training data. To this end, I developed 'pickleio', a messaging library based on Python's pickle library that enables both synchronous low-latency messaging as well as asynchronous high-throughput messaging. Using this library, I achieved a round-trip latency is 100ms: 40ms for transfer to the server, 30ms for model inference, and 30ms for transfer back to the client. The network latency is poor primarily because a VPN is necessary to connect to the Tufts network, adding additional network hops. This latency is visible in the robot behavior as "stuttering," which can be seen in example recordings.

Additional codebase adjustments include enabling in-training-loop inference for online RL algorithms, adding support for the specific robot arm I decided to use, and creating a client side REPL for initiating training, inference, and teleoperation routines. These programming tasks are not particularly interesting in an academic sense but do require a significant amount of effort. The final codebase is open source at `https://github.com/LQ1234/cs138-code`.

2.3 Environment and Task

Most SOTA VLA experiments are performed on robot hardware that is out-of-reach for this project. For instance, the bill-of-materials for Stanford's ALOHA-2 robot adds up to more than $50,000 dollars [7].

Instead, I decided to build the SO-101 arm, an open-source 3D-printable 6 degree-of-freedom arm designed by The Robot Studio [8, 9]. Being 6 DOF, the arm is dexterous enough to perform many of the same behaviors as SOTA arms, but at a fraction of the price: each arm could be built for less than $150. Additionally, the arm has a large community supporting quality documentation and sample code.

The environment consists of one SO-101 arm mounted to a desk. There is a webcam mounted on the arm's wrist to provide close-up views, as well as another webcam providing a view of the entire desk. Task-related items are placed on the desk, such as wooden blocks. A photo of the environment is shown in Figure 2a, with a view of the camera in Figure 2b, and the teleoperation arm in Figure 2c.

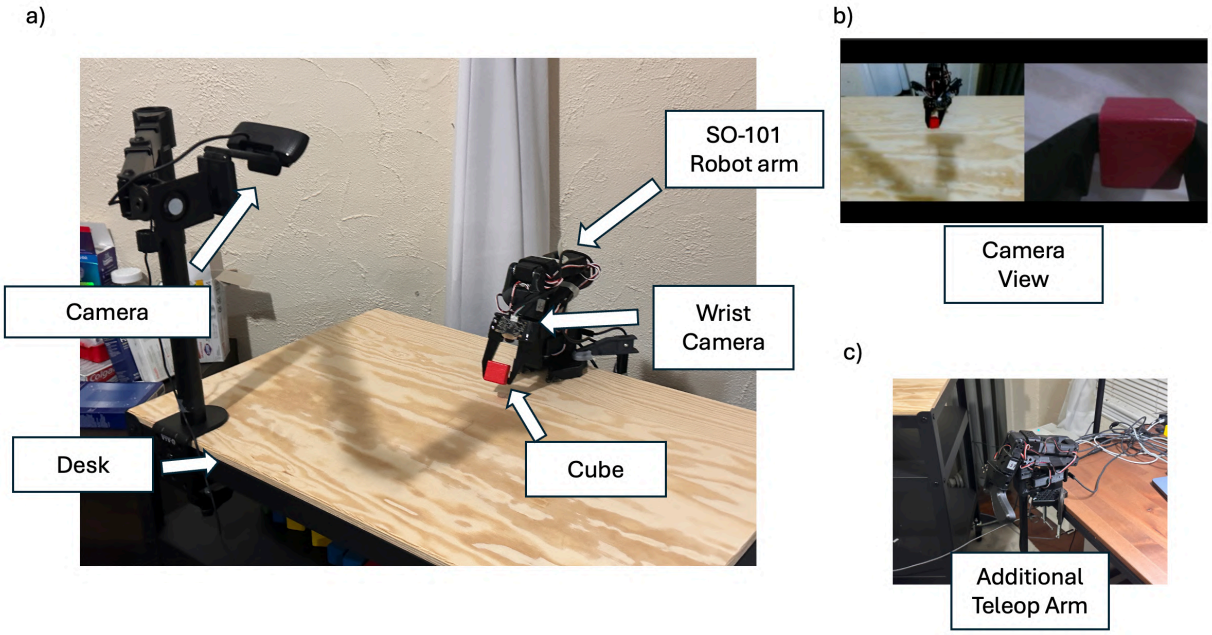


Figure 2: (a) Photo of the environment including desk, robot arm, both cameras, and a cube. (b) View from desk and wrist camera. (c) Additional teleoperation arm.

For this project, I ended up deciding on an extremely simple task: picking up a cube. There are several reasons why I ended up using such a simple task. Firstly, the policy is more likely to converge with fewer training samples, which is important given that demonstration collection is time consuming and I am performing this project individually with time constraints. Secondly, the episode length is short (typically 7–30 seconds), which means that reinforcement learning can be done in a reasonable amount of time. Thirdly, despite being a simple task, the behavior is still quite complex: The policy needs to learn to navigate out of the starting configuration to the gripping position, align the gripper to the block, close the gripper and pick it up, return to the starting position, and retry if the block fell out or wasn't successfully gripped. The main metric for the success of this task is whether the policy was able to pick up the block, and if so, the time it took to complete the task.

2.4 Reinforcement Learning Algorithm

The last consideration is which reinforcement learning algorithm to apply on the RL step. Given that the goal is to improve an existing policy, a policy gradient method is the most applicable here. One of the most common policy gradient methods is Proximal Policy Optimization (PPO), an algorithm proposed by OpenAI in 2017 [6]. In PPO, samples are collected from the environment, and then the policy is optimized (via gradient descent) to increase the likelihood of sampling actions that lead to high reward states. Specifically, a common clipped objective is:

$$L^{\text{CLIP}}(\theta) \;=\; \mathbb{E}_t\Big[\, \min\big(r_t(\theta)\,\hat{A}_t,\; \text{clip}(r_t(\theta),\, 1-\epsilon,\, 1+\epsilon)\,\hat{A}_t\big)\,\Big],$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the likelihood ratio and $\hat{A}_t$ is an advantage estimate [6].

There are two significant terms that differ this algorithm from simple weighted fine-tuning. First, the relative likelihood between the action in the new and the old model is clipped (usually between 0.8x and 1.2x) as to prevent the new model from diverging too much and causing instability. Second, the advantage A (the "weight" in weighted fine tuning) is calculated by subtracting the reward from value of the original state. This prevents poor actions that started from a good state from being favored over good actions that started from a poor state.

The disadvantage of PPO is that a separate value function needs to be trained. An alternative method, Group Relative Policy Optimization (GRPO) capitalizes from that fact that if all actions started from the same state, then the value function can be removed, and the advantage can be calculated by comparing returns directly. However, the episode can only consist of one interaction between the policy and the environment, but if the action is considered as an entire trajectory generated by the policy, this is possible. By removing references to the state, it no longer needs to be computed, reducing the complexity of the algorithm significantly.

Unfortunately, GRPO cannot be applied directly to π_0. This is because the action generator is a flow-based model, which generates samples not by sampling from a distribution output by the model but rather by repeatedly applying a function to noise. Therefore, it is nontrivial to calculate the relative probabilities of trajectories required by the PPO term. One alternative is to use π_0-FAST, which is a variant of π_0 that uses a more typical autoregressive tokenized action representation that does output direct probabilities [2]. Alternatively, if the learning rate is small enough, it could be assumed that the relative probability is always close to 1, in which the PPO clipping term can be removed. The resulting algorithm degenerates to a sort of single-interaction group-relative version of REINFORCE, where several episodes are collected from the environment, the returns are compared, and descent is performed to increase the likelihood of high reward trajectories.

The final method I used for the RL stage is shown in Algorithm 1. Rollouts are sampled from the policy, advantages are assigned, and the policy is optimized on the rollouts according to the advantages. The specific number of rollouts used was $n = 10$, with learning rate $\eta = 10^{-5}$. The way the advantages are assigned is discussed later.

Algorithm 1 Group-relative REINFORCE Algorithm (flow-matching)

Input: a flow-matching-based policy $T \sim \pi_\theta(z)$, where T represents an entire trajectory that results from several policy-environment interactions, θ represents model parameters, and z represents noise
Input: learning rate η
Initialize parameters θ
while NOT stopping criterion **do**
  for $i = 1$ to n **do**
    Generate a trajectory $T_i \sim \pi_\theta(z)$ in the environment
  end for
  Assign advantages A_i to each trajectory T_i based on relative performance
  Define loss:

$$L(\theta) = \frac{1}{n} \sum_i \left(A_i \cdot \mathbb{E}_{z \sim \mathcal{N}} \|\pi_\theta(z) - T_i\|^2 \right)$$

  Update $\theta \leftarrow \theta - \eta \cdot \text{Adam}(\nabla_\theta L(\theta))$
end while

3 Results and Discussion

3.1 Supervised Fine-Tuning

As π_0 is already pre-trained, the first step is to fine-tune it for the block-picking task. For the first training attempt, I manually collected 50 episodes and performed fine-tuning of the model with MSE loss using the Adam optimizer. The hyperparameters used were $\eta = 10^{-5}$ and no. epochs = 3, and the training curve is

shown in Figure 3. As the loss is MSE with the angles in each joint normalized to 0–100, the low loss values of < 1 would suggest that the model is extremely overfit. However, I found that continuing to train even after the model has converged based on the loss graph seems to improve the behavior of the policy during inference, including reducing jittering and "indecision" (the policy doing nothing).

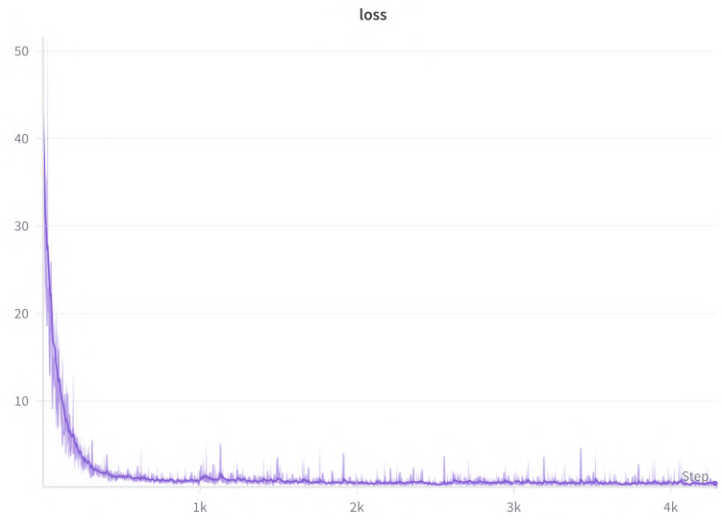


Figure 3: Training loss graph for the first SFT run, where the step (each consisting of 32 episode frames) is plotted against loss, the mean-squared-error against the training data. Loss decreases to below 1 within the first 1000 steps, suggesting that the model is already overfit, but continuing to train results in better inference behavior.

However, the performance of the policy on the task was quite poor. The policy would guide the arm towards the block and attempt to close the gripper, but often the arm would miss and instead the arm would "curl up," ending up in a state from which it would never recover. A representative rollout is shown in the supplemental videos as "failure_curled_up."

A state-space schematic visualizing why I think this problem is occurring is shown in Figure 4a. After training converges, the policy is guaranteed to point in the same direction as the example trajectories, but the policy can be divergent. This means that during inference, tiny errors from the starting state and noise from the environment accumulate, and rather than compensating for those errors, the policy magnifies them, resulting is a massively different end state such as the "curled up" configuration.

For the second attempt, I collected 50 more episodes. However, rather than starting at the same starting state for all of them, I collected episodes starting at different points along the target trajectory, with moderate errors (such as the arm angle or gripper alignment being off) that I would resolve during the episode. I then re-trained the policy (starting from fresh pre-trained weights) on the entire mixture of 100 episodes with the same hyperparameters as the first. The resulting policy is significantly better than the first attempt, successfully picking up the block most of the time. Figure 4b shows an illustrative state-space diagram explaining how those extra demonstrations improved the model: by initializing the episode in states that are near the target trajectory and then moving towards it, the policy is forced to become convergent.

One last experiment I performed before the RL step is to see what would happen if I initialized an episode in a state that the policy wasn't trained on. Specifically, I put the robot in a "snake-like" configuration before enabling policy inference. A recording of the rollout is available in the supplemental videos as "failure_snake." The policy fails to get the robot out of the snake-like configuration and instead flops around meaninglessly. This behavior makes sense: as shown in Figure 4c, this state was far from the states at which it would converge to the target trajectory, so instead the behavior of the policy is unknown.

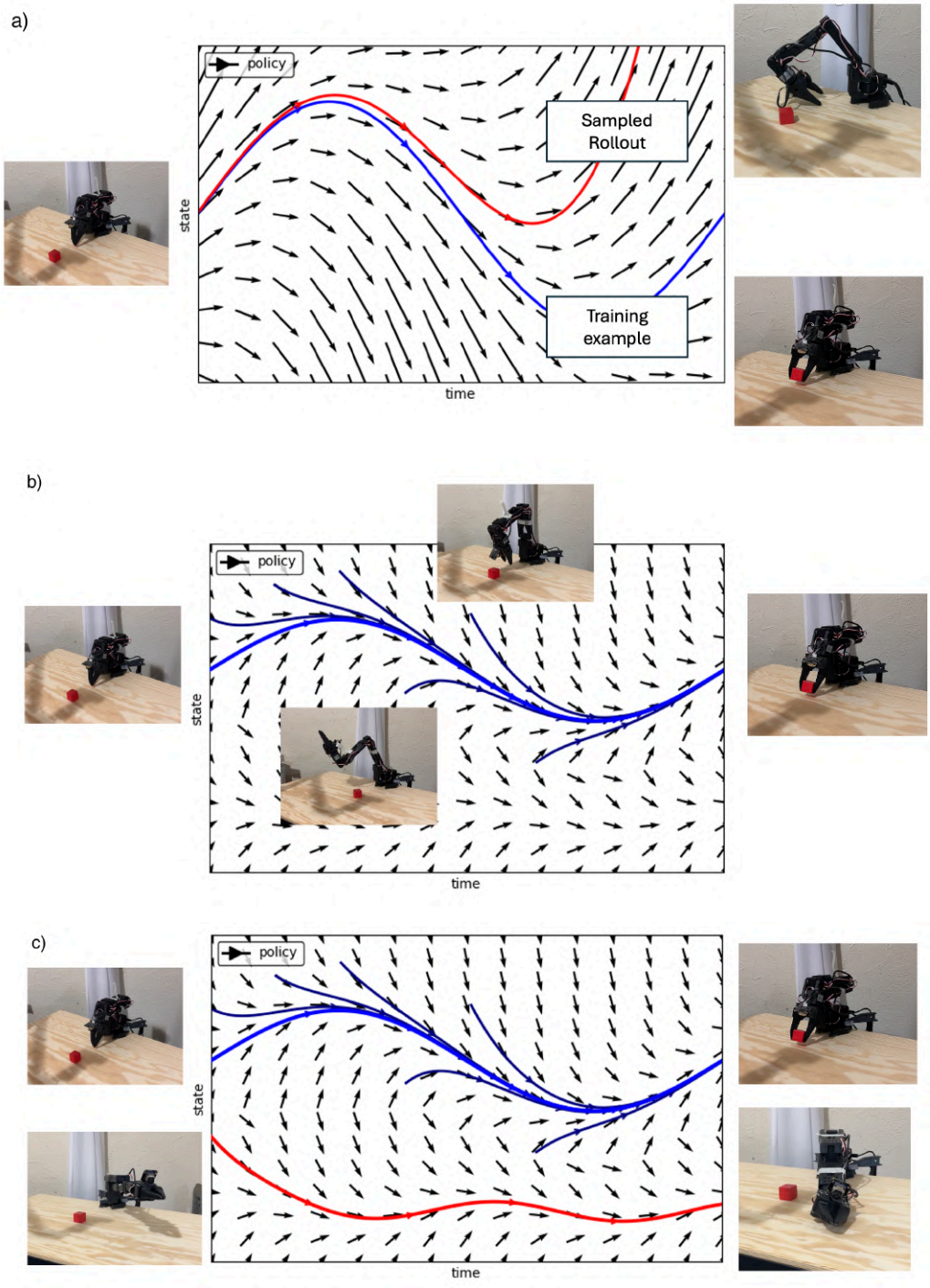


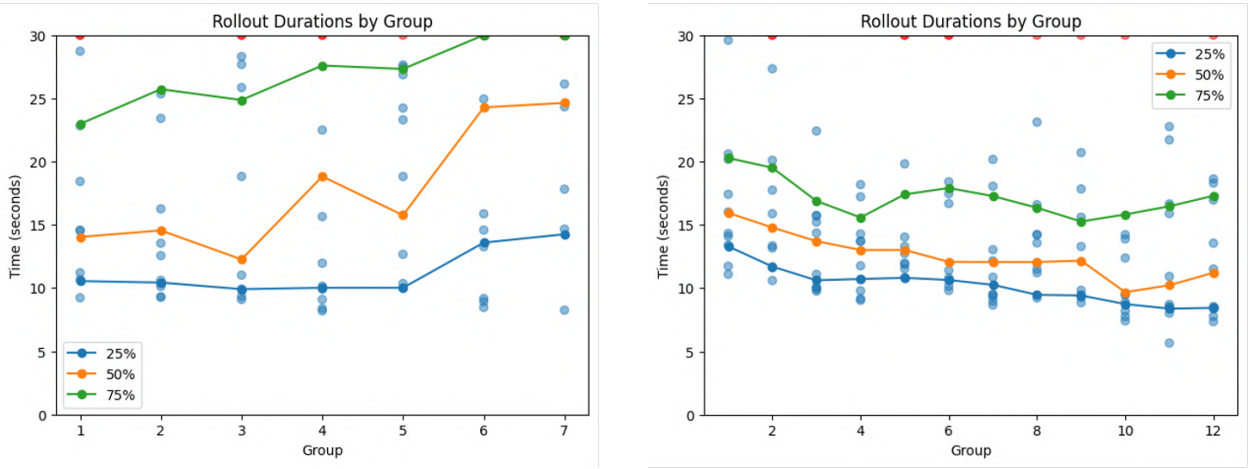
Figure 4: (a) State-space diagram showing why the first policy fails. The policy is divergent, causing small initial errors to amplify and result in a significantly different end state. Note that the diagram is purely illustrative. (b) Additional training samples that start near the target trajectory but with an error that is resolved during the episode forces the policy to be convergent. (c) Initial states that are far from the states that the model has seen during training might still fail to converge, such as the "snake" configuration.

3.2 Reinforcement Learning

Finally, with a strong base model, I began the reinforcement learning step by following the algorithm described previously (Algorithm 1). For the first attempt, I assigned advantages based on episode duration only, where:

$$A_i = \begin{cases} 1 & \text{if duration}(T_i) \leq \text{median}(\text{duration}(T_1), \ldots, \text{duration}(T_n)), \\ 0 & \text{otherwise.} \end{cases}$$

A strength of this method is that the advantage calculations can be performed automatically based on the episode duration, so the only step that needs to be manually done is the environment resetting. However, as seen in Figure 5a, the performance of the policy diverges. The reason that this happens is that the policy learns a risky gripping technique where the gripper holds the block at the very tip, which makes it likely for the robot to drop the block or fail to align the gripper in later episodes. As half of each group is always trained on, this behavior is reinforced even though it decreases performance.



(a) (b)

Figure 5: Rollout durations by group, where each rollout is scattered across group and duration. Rollouts that are longer than 30 seconds or never completed are clipped and shown in red. A smoothed line of the 25th, 50th, and 75th percentile is also shown, where each point corresponds to the percentile of the previous, current, and next groups. (a) Rollout durations for the first attempt. While possible improvement can be observed in the first three groups, the policy diverges, with the rollout duration increasing onwards. (b) Rollout durations for the second attempt. Here, policy improvement is visible, with the rollout times decreasing from the median of the first two groups of 15.2 seconds to 9.8 seconds, the median of the last two.

For the second RL attempt, I refined the method in which advantages would be assigned: advantage would only be given if the rollout duration was better than the current standard, if the gripping method was strong and stable, and if the robot completely returned to the starting position. However, this was not a hard-and-fast rule, and for rollouts where the robot did something desirable, I would assign advantage even its performance was worse. I continued to assign binary advantages for simplicity; however, the number of rollouts that were trained on now differs between groups.

The resulting graph of rollout duration against group is shown in Figure 5b. Although the durations against time is noisy, there is a definite trend towards lower rollout durations, and the median duration decreases from the first two of 15.2 seconds to 9.8 of the last two, a 35% decrease in duration.

A sample rollout from the final model is shown in the video "final_model_success." The robot moves to the block, picks it up, and returns to the starting position in only 7 seconds, making no mistakes. A more interesting rollout is shown in the video "final_model_multiple_attempts." Here, the model fails to pick up the block in the first attempt, but tries again, until succeeding in the end. This demonstrates some of the complex behavior that can be exhibited even for a simple task like picking up a block.

If I had continued to perform the group RL procedure, it is possible that the performance would continue improving. However, a major drawback of real-world live RL like here is the time that it takes to perform the RL process. 12 groups represent 120 different rollouts, each of which requires watching the robot perform and resetting the environment. Adding on the time required to wait for the policy to train, it took over an hour to perform the second experiment. A sped-up recording of a portion of the training process is shown in the video "training_100x."

4 Conclusion and Next Steps

In conclusion, I demonstrated that reinforcement learning can be performed in the real world on a generalist policy through the language-domain inspired process of using a pre-trained and fine-tuned policy before beginning the RL process. I selected Physical Intelligence's π_0 model, solved infrastructure issues and created a framework for remote inference, built a real-life RL environment using the SO-101, collected training samples for a block-picking task, resolved SFT difficulties according to my divergence theory, and finally created and performed the group RL algorithm, resulting in 35% decrease in task completion duration.

There are many directions to pursue for future work. For instance, while I use a policy capable of generalist behavior, I never use that capability in this project. One direction could be to train a policy to pick up and drop differently colored blocks, with commands such as "place the red block on the blue block." Another direction is to attempt to automate the reinforcement learning process to make it more scalable. For example, it is possible that a commercial off-the-shelf VLM could perform the advantage rating step, and the policy itself could be trained to reset the environment. A third direction could be to utilize the policy to its fullest potential with more complex tasks, such as folding laundry or cleaning up tables, which it can already do with other robot embodiments.

References

[1] M. J. Kim et al. *OpenVLA: An Open-Source Vision-Language-Action Model*. arXiv:2406.09246, 2024. https://arxiv.org/abs/2406.09246.

[2] K. Black et al. *π_0: A Vision-Language-Action Flow Model for General Robot Control*. arXiv:2410.24164, 2024. https://arxiv.org/abs/2410.24164.

[3] NVIDIA. *NVIDIA Announces Isaac GR00T N1 — the World's First Open Humanoid Robot Foundation Model*. Press release, Mar 18, 2025. https://nvidianews.nvidia.com/news/nvidia-isaac-gr00t-n1-open-humanoid-robot-foundation-model-simulation-frameworks.

[4] L. Beyer et al. *PaliGemma: A versatile 3B VLM for transfer*. arXiv:2407.07726, 2024. https://arxiv.org/abs/2407.07726.

[5] Open X-Embodiment Collaboration. *Open X-Embodiment: Robotic Learning Datasets and RT-X Models*. arXiv:2310.08864, 2023. https://arxiv.org/abs/2310.08864.

[6] J. Schulman et al. *Proximal Policy Optimization Algorithms*. arXiv:1707.06347, 2017. https://arxiv.org/abs/1707.06347.

[7] J. Aldaco et al. *ALOHA 2: An Enhanced Low-Cost Hardware for Bimanual Teleoperation*. arXiv:2405.02292, 2024. https://arxiv.org/abs/2405.02292.

[8] Hugging Face. *SO-101 Documentation (LeRobot)*. https://huggingface.co/docs/lerobot/en/so101.

[9] The Robot Studio. *SO-ARM100 / SO-101 repository*. https://github.com/TheRobotStudio/SO-ARM100.