Developing a Simulation Platform for the Benchmarking of Generalist Robot Policies

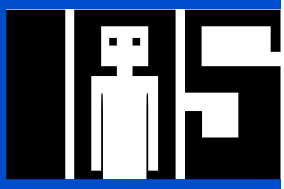
Entwicklung einer Simulationsumgebung für das Benchmarking von Generalist Robot Policies

Master thesis in the field of Information Systems by Tristan Jacobs
Date of submission: April 28, 2025

1. Review: Prof. Jan Peters
2. Review: Theo Gruner
3. Review: Daniel Palenicek
4. Review: Tim Schneider
5. Review: Maximilian Tölle

Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT



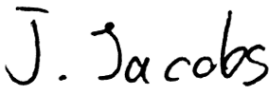
Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, Tristan Jacobs, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB der TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.



Darmstadt, 28. April 2025

Tristan Jacobs

Abstract

We present RoboXim, a novel simulation platform developed using the Unity Game Engine [48] to investigate the generalization capabilities of generalist robot policies. The simulation platform facilitates the generation of demonstration datasets and the evaluation of robot policies. Using our platform, we test whether generalist models such as Octo [28], RT-1-X [9], and OpenVLA [23] can perform tasks zero shot in our environment and find that they are generally not capable of doing so. Therefore, we fine-tune models to our environment in order to evaluate generalization across dimensions such as visuals, skills, and objects. Through extensive experiments with the Octo model, we demonstrate that while fine-tuning enables high success rates for specific tasks, performance varies significantly with diverse objects and skills, particularly those requiring precise manipulation. In addition, we propose a benchmark that can measure generalization and knowledge transfer across multiple dimensions. Evaluating Octo with this benchmark reveals that Octo can transfer knowledge across dimensions effectively.

Zusammenfassung

Wir stellen die Simulationsumgebung RoboXim vor, die mit der Unity Game Engine [48] entwickelt wurde, um die Generalisierungsfähigkeiten von *generalist robot policies* zu untersuchen. Die Simulationsplattform ermöglicht die Erzeugung von Demonstrationsdatensätzen und die Evaluation von *policies*. Mithilfe unserer Plattform testen wir, ob generalistische Modelle wie Octo [28], RT-1-X [9] und OpenVLA [23] in unserer Umgebung Aufgaben direkt ohne dediziertes *fine-tuning* ausführen können und stellen fest, dass sie dazu im Allgemeinen nicht in der Lage sind. Aus diesem Grund *fine-tunen* wir die Modelle auf unsere Umgebung, um anschließend die Generalisierung für Dimensionen wie Bildmaterial, Fähigkeiten und Objekte zu bewerten. In umfangreichen Experimenten mit dem Octo-Modell zeigen wir, dass das *fine-tuning* zwar hohe Erfolgsquoten bei bestimmten Aufgaben ermöglicht, die Leistung jedoch bei verschiedenen Objekten und Fähigkeiten, insbesondere bei solchen, die eine präzise Manipulation erfordern, erheblich schwankt. Des Weiteren stellen wir eine Benchmark vor, die Generalisierung und den Wissenstransfer über mehrere Dimensionen hinweg messen kann. Die Bewertung von Octo mit dieser Benchmark zeigt, dass Octo Wissen effektiv über mehrere Dimensionen hinweg übertragen kann.

Contents

Figures and Tables

List of Figures

List of Tables

1 Introduction

The advancement of large-scale models in domains like vision and natural language processing, has inspired new possibilities for robotic control. Typically, robot policies are trained with a dataset of demonstrations specific to a narrowly defined task. However recently, a number of so-called generalist robot policies have been released that aim to control robots for a wide range of task settings [9, 23, 28]. These models are often derived from large-scale vision-language models (VLMs) by fine-tuning them to interpret their outputs as robotic actions instead of a natural language sequence. Employing generalist models is desirable because collecting robotic training data for a task is resource intensive since it is often manually generated by human teleoperation. According to the original authors, many of the generalist models which we present in more detail in chapter 3 show generalization capabilities across different skills, objects, and even various robot embodiments [9]. While some of the generalist models even exhibit zero shot capabilities in environments seen in their training [5, 6, 28], an important aspect of such a model is that it can be effectively fine-tuned to a new task with only a few demonstrations [28].

The main goal of this work is to benchmark and conduct an in-depth investigation of the generalization capabilities of generalist robot policies. This is done by utilizing our own custom simulation platform that is specifically developed to evaluate these policies. The generalization and transfer of knowledge is investigated across various dimensions, such as vision, different skills, and objects. Our research questions are (1) whether the generalist models can run in our new simulation environment zero shot, (2) whether we can fine-tune these models to our environment, (3) to what extent knowledge is transferred from the pretraining, and (4) how well the model can generalize and transfer knowledge across dimensions. To this end, we contribute a simulation platform that can be used for both the collection of demonstrations for fine-tuning, and the evaluation of robot policies. We also develop and demonstrate a benchmark to measure and compare the generalization capabilities of a model.

2 Foundations

This chapter contains some preliminaries for our work. First, we provide background information on generalist robot policies in section 2.1. Then, we present the Unity Game Engine [48] that is used to implement our simulation platform in section 2.2.

2.1 Generalist Robot Policies

We describe the architecture of generalist robot policies in detail in chapter 4 where we present a number of existing models. In the following sections we provide basic definitions in the field of robot learning and some background on imitation learning.

2.1.1 Environment

A robot exists and learns in an environment. For the definition of an environment and of a task (see section 2.1.2) we adopt the definitions of an existing taxonomy called STAR-Gen [15]. An environment is a tuple

$$E = (\mathcal{S}, \mathcal{O}, \mathcal{A}, \mathcal{L}, f_o, f_t),$$

where $\mathcal{S}$ is the state space, $\mathcal{O}$ is the observation space, $\mathcal{A}$ is the action space, $\mathcal{L}$ is the space of language instructions, $f_o : \mathcal{S} \to \mathcal{O}$ is the observation function, and $f_t : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the transition function. $\mathcal{O}$ consists of images, e.g., of the workspace or images from a camera attached to the robot end effector. $\mathcal{A}$ consists of robot actions, e.g., delta poses for the end effector [15].

2.1.2 Task

The definition of a task varies in the literature. In this work, we mostly use the definition from STAR-Gen [15]. The definition of a task may vary in chapter 4 where we present related works that use their own definitions. A task space T is defined for a given environment E. A task is a tuple $\tau \in T$

$$\tau = (p_\tau(s_0), l_\tau, R_\tau),$$

where $p_\tau(s_0)$ is an initial state distribution for E, $l_\tau \in \mathcal{L}$ is a language instruction, and $R_\tau : (\mathcal{S} \times \mathcal{A})^* \to \{0, 1\}$ is a success function that maps a sequence of states and actions to either 0 for failure or 1 for success [15].

2.1.3 Policy

A policy $\pi(a \mid o^n, l)$ is a function that takes in $n \geq 1$ observations, and a language instruction. It outputs an action distribution. An expert policy π_E is a policy that leads to the success function R_τ yielding 1 after a number of actions produced by π_E are applied to the environment E [15].

2.1.4 Robot Learning

In the context of our work, the goal in robot learning is to learn a policy to solve a task with a language instruction and image observations. At each timestep t the policy π is provided with a language instruction l and image observations o_t. We control the robot by sampling an action a_t from the action distribution $\pi(\cdot \mid l, o_t)$. This process ends when a termination condition is reached, e.g., the task is solved successfully or a maximum number of steps is reached. The complete sequence of interactions $l, \{(o_i, a_i)\}_{i=0}^{T}$, from the initial step $t = 0$ to the final step T, is called an episode. When an episode is completed successfully, that is, the robot performed the language instruction, the agent receives a reward of 1. Otherwise, the reward is 0. The objective is to learn a policy that maximizes the average reward for a given environment and task distribution [5].

2.1.5 Imitation Learning

Imitation learning is a machine learning paradigm in which an agent learns to perform tasks by mimicking the behavior of an expert demonstrator. A policy π is trained with a dataset $\mathcal{D} = \{(l^n, \{(o_t^n, a_t^n)\}_{t=0}^{T^n})\}_{n=0}^{N}$ of successful episodes [30, 58, 20]. A common approach to learn π is behavioral cloning [30], where the agent learns a mapping from states to actions via supervised learning. Imitation learning is particularly effective in domains like robotics due to the difficulty of specifying reward functions [16].

2.2 Unity

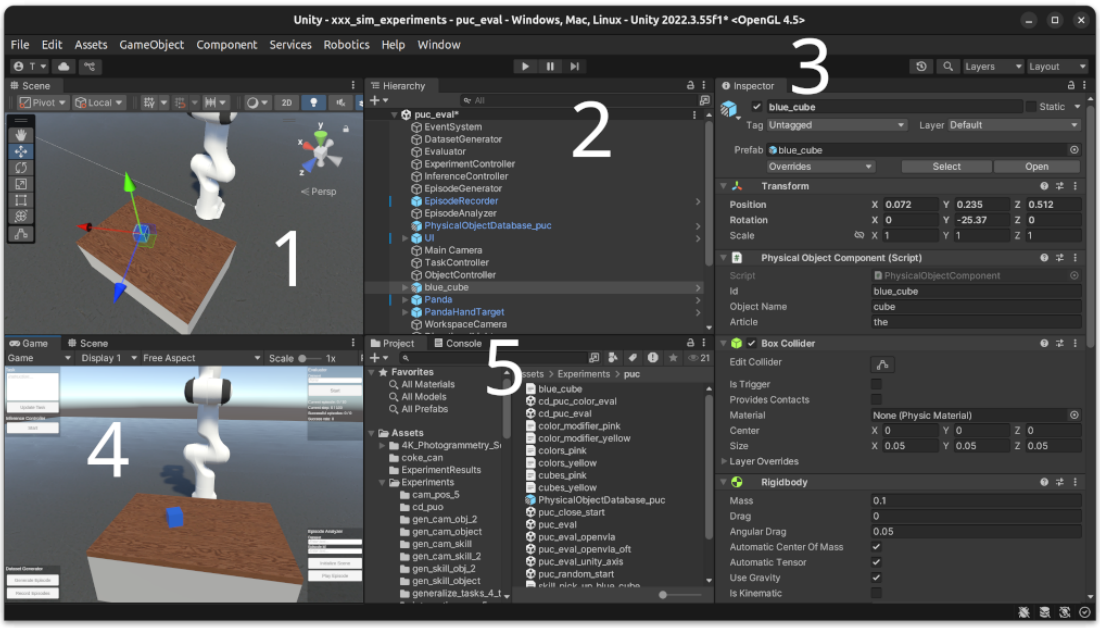


Figure 2.1: A screenshot of the Unity editor showing the scene view (1), the hierarchy (2), the inspector(3), the game view (4), and the project files (5).

In this section we provide an overview of the Unity Game Engine [48] that is used to implement our simulation platform. Unity is an established game engine that features a user friendly editor. It provides a robust feature set to create a robotics simulation,

including rendering, a physics simulation powered by PhysX, and scripting. A screenshot of the Unity editor is shown in figure 2.1. In the top left is the interactive scene view marked with the (1). A scene is a 3D space used to organize objects and other elements of the simulation. In Unity, objects that exist in a scene are called `GameObjects` and are listed in the hierarchy view (2). Each `GameObject` can have components that determine its function or behavior. All `GameObjects` have the `Transform` component because it determines the `GameObject`'s position and rotation in the 3D scene. Other components are optional. Some components are provided by Unity, such as a `Renderer` to render objects in the 3D scene, and a `Rigidbody` to enable the physics simulation. Users can also author their own components using C# to implement custom functionalities. When a `GameObject` is selected in the hierarchy, the inspector view (3) shows all the attached components of this `GameObject`. A component can expose variables that can be configured directly in the inspector. The game view (4) shows a preview of the camera that renders to the screen when we run the simulation. We can enter play mode directly in the editor by clicking on the play button at the top. This allows us to instantly test out new configurations or even adjust variables while in play mode from the inspector. Window (5) shows all project files such as 3D objects, textures and scripts. The implementation of our simulation platform called RoboXim is detailed in section 4.2. We use the Unity scenes to visually configure each of our experiments. In addition, we make use of a feature called `Prefabs`. We can create a `Prefab` by dragging a `GameObject` from the hierarchy into the project view. This saves all the property values of the attached components and syncs them in between scenes if an object was instantiated from this prefab. For example, if we change something on the Panda robot, we can propagate this change to all experiments, saving a lot of configuration time.

3 Related Work

In this chapter we present a selection of generalist robot policies in section 3.1. Then, we describe existing benchmarks and simulators in section 3.2. In section 3.3 we introduce a taxonomy designed for the evaluation of generalist policies.

3.1 Generalist Robot Policies

Generalist robot policies are inspired by the success of large and general models used in domains like vision and natural language processing. These models often use a Transformer architecture first introduced in 2017 [51]. RT-1 stands for Robotics Transformer 1 and is an architecture that encodes camera images, language instructions, and end-effector actions into token representations that can then be used by a Transformer [5]. Specifically, RT-1 takes a history of 6 images and tokenizes them by flattening the output of an ImageNet pretrained EfficientNet-B3 [43] model into 81 tokens. Universal Sentence Encoder [7] is used to produce a language embedding of the instruction. The image tokenizer is conditioned on this embedding, promoting the early extraction of task-relevant image features. Before the tokens are passed on to the Transformer, only 8 important tokens of the 81 tokens are selected with TokenLearner [40]. This is done to increase the inference speed of the Transformer and, therefore, make RT-1 viable for real-time control. Finally, the Transformer predicts action tokens consisting of seven tokens for arm movement (x, y, z, roll, pitch, yaw, gripper opening), three tokens for base movement (x, y, yaw), and one token for switching between arm movement, base movement, or terminating the episode. Each action token represents one of 256 bins that the action dimensions have been discretized into [5].
RT-1 is trained on approximately 130k demonstrations of a robot performing various tasks in a kitchen environment. The authors define a task as a verb surrounded by one or more nouns, where the verb is also referred to as skill, and the nouns correspond to

various objects in the scene. Eight different skills combined with several objects result in a total number of approximately 700 tasks. When RT-1 is evaluated on tasks that were present in the training data, a success rate of 97% is reached. In order to test generalization capabilities, RT-1 is also evaluated for robustness to distractor objects, changing backgrounds, and unseen tasks, i.e., certain combinations of skills and objects that were withheld from the training data. The authors report success rates of 83% for distractor robustness, 59% for new backgrounds, and 76% for unseen tasks. In addition, the authors examine the generalization for a combination of the mentioned dimensions by defining three levels. The first level is a new kitchen environment, the second level introduces unseen distractor objects, and the third level includes new task objects or objects in unseen locations. Success rates of 88%, 75%, and 50% are reported for levels 1-3. In conclusion, the authors show that RT-1 can successfully learn a large number of tasks and is capable of generalizing to new tasks, distractors, and backgrounds. The RT-1 code is open source and trained checkpoints are publicly available [5].

Another model called RT-2 builds on RT-1, but instead of just training on robotic trajectory demonstrations a large VLM pretrained on internet-scale data is co-fine-tuned with robotic demonstrations [6]. As in RT-1 the actions for the robot are encoded as tokens and therefore can be treated by the VLM in the same way as natural language tokens. Such models are referred to as vision-language-action models (VLAs). There are two instances of RT-2 that differ in the choice of the pretrained VLM. One is built on PaLI-X [8] the other on PaLM-E [11]. The action space of the robot consists of six variables for the end-effector pose, one for the gripper, and one for terminating the episode. Each continuous variable is discretized into 256 bins. In order to fine-tune the VLMs to output robot actions, 256 existing tokens of a VLM are reserved to represent the bins. For PaLI-X [8] the tokens representing integers from 1-256 are used. In the case of PaLM-E [11] such a token representation does not exist, therefore the 256 least used tokens are repurposed instead. To ensure that a model only produces valid robot actions, only the mentioned 256 tokens are sampled for its output vocabulary [6].
The robotic demonstrations used in the co-fine-tuning are from the RT-1 dataset. The authors also use the same notion of tasks and skills as in RT-1. RT-2 is evaluated for seen tasks, unseen objects, unseen backgrounds, and unseen environments. While the performance for seen tasks is equal to RT-1, the success rate for the unseen experiments approximately doubles compared to RT-1 indicating that knowledge from the Internet-scale pretraining is transferred and results in better generalization. RT-2 is also tested for what the authors call *emergent capabilities*. These are capabilities not present in the robotic training data that are supposed to emerge from the pretraining, i.e., semantic and visual concepts of objects and their relations. The authors define the following three

categories: *Symbolic understanding* evaluates the transfer of semantic knowledge from the pretraining. Instructions include objects or relations that were not present in the robotic data. The second category *reasoning* requires the VLM to perform the following examples of reasoning: visual reasoning (e.g., "move the apple to cup with same color"), math (e.g., "move X near the sum of two plus one"), and multilingual understanding, i.e., executing an instruction given in another language. The third category is called *human recognition* and evaluates tasks related to humans (e.g., move the coke can to the person with glasses). Both RT-2 variants outperform RT-1 in all categories. On average, the PaLI-X [8] variant achieves more than three times the success rate of RT-1. The PaLM-E [11] variant performs slightly worse than PaLI-X [8] especially in the category *symbol understanding*. While the results show that transfer of knowledge from pretraining is possible, the authors find that the robot is not able to perform new skills or motions that were not present in the robotic fine-tuning data [6].

One challenge in training large general models in robotics is the availability of data. While VLMs can be trained with data scraped from the Internet, collecting robotic demonstrations is more time-consuming and often requires teleoperation of a robot by a human. For example, collecting the training data for RT-1 took 17 months with the use of 13 robots [5]. Open X-Embodiment is a collaborative effort by multiple institutions to provide a large number of diverse robotic datasets [9]. At the time of its release, the Open X-embodiment dataset is made up of 60 datasets from 34 robotic research labs. It contains 1M+ real robot trajectories from 22 different robot embodiments. There are 311 different scenes, 5228 objects, and 527 skills, although most trajectories depict a skill from the pick-place family [9].
Multiple X-embodiment datasets are then used to train the RT-1 and RT-2 models which are called RT-1-X and RT-2-X. An RT-1-X checkpoint is publicly available. For a number of X-embodiment datasets, these RT-X models are then compared to an RT-1 variant that is trained only on the respective dataset. The evaluation is split between small- and large-scale datasets. For small datasets, only RT-1-X is considered. It outperforms the RT-1 variant on average by 50%, implying that domains with limited data availability benefit from X-embodiment co-training. In the case of large datasets, RT-1-X achieves lower success rates than the RT-1 variant. However, RT-2-X can outperform the RT-1 variant. The authors conclude that in the large-scale data domain, a sufficiently large model architecture is needed to benefit from the X-embodiment co-training. In another experiment, the transfer of knowledge between different robot embodiments is studied. RT-2-X is able to execute additional skills that were only seen in other robot embodiments [9].

Another model trained on the Open X-embodiment data is Octo [28]. It is open-source

and checkpoints of two versions, Octo-Small (27M parameters) and Octo-Base (93M parameters) are available. Octo introduces a more flexible architecture that allows for flexibility in task definitions, observations, and action spaces. The task can be specified by a language instruction or by a goal image. In the observation space, a third person camera and a wrist camera are supported. Actions can be predicted for end effector or joint control. This design is supposed to make Octo usable in many robot applications and enables efficient fine-tuning for different robot setups [28].
The architecture of Octo has three core components: *Input tokenizers*, a *transformer backbone*, and *action readout heads*. First, *input tokenizers* are used to create tokens from the language instruction or goal image, and the observation images. For language inputs, a t5-base [32] model is used. Images are processed with a shallow convolution stack and split into a sequence of flattened patches [10]. Then, the produced tokens are passed to the *transformer backbone*. The Octo architecture introduces special *readout tokens* that only attend to observation and task tokens, but are not attended by any of the observation and task tokens. The *action readout head* then uses a diffusion process to produce actions from the *readout tokens* [28].
To evaluate Octo, two types of experiments are performed: zero-shot evaluation and fine-tuning evaluation. For the zero-shot evaluation, Octo is tested in four settings from the pretraining data. Only seen tasks are considered, but lighting conditions and object placement varies slightly from the original scenes. In these settings, Octo outperforms RT-1-X and achieves a similar success rate to RT-2-X. However, only 10 episodes are executed per experiment, which may cause inaccurate results. For the fine-tuning evaluation, Octo is tested in six settings not present in the pretraining data. Approximately 100 demonstrations are collected per setting. While Octo is fine-tuned separately for each setting, the same hyperparameter config is used across all runs. In the evaluation Octo achieves an average success rate of 72% clearly outperforming a baseline model trained from scratch. This shows that the pretrained Octo model is useful for fine-tuning to new settings and that the provided default config is a good starting point for fine-tuning applications. The authors do not further explore generalization capabilities in the sense of unseen tasks and objects. Instead, generalization is primarily understood as being able to execute many tasks with different robot embodiments, where all these tasks, embodiments, and scenes were in the pretraining or fine-tuning data [28].

OpenVLA is a 7B-parameter model similar to RT-2-X [23]. It is also trained with the Open X-embodiment data [9], but in contrast to RT-2 and RT-2-X it is open-source and pretrained checkpoints are available. OpenVLA builds on the Prismatic-7B VLM [22]. Instead of a single visual encoder, Prismatic uses two encoders in parallel and concatenates their outputs which is beneficial for spatial reasoning [22]. The two encoders are the SigLIP

[57] and DinoV2 [29] models. As its large language model (LLM) backbone, Prismatic uses the Llama 2 model [47]. In order to turn the Prismatic VLM into a VLA, many of the techniques presented in RT-2 [6] are used. The continuous robot actions are discretized and represented with 256 action tokens. During fine-tuning the 256 least used tokens in the Llama tokenizer's vocabulary are overwritten with these action tokens [23].
To evaluate OpenVLA, two types of experiments are conducted in settings from the training data. For the first type, the "Google robot" from the RT-1 and RT-2 evaluations [5, 6] is used, and performance for in-distribution and out-of-distribution tasks is evaluated. OpenVLA is compared to RT-1-X [9], RT-2-X [9], and Octo [28]. While RT-2-X achieves a similar success rate to OpenVLA, RT-1-X and Octo are significantly outperformed for both in-distribution and out-of-distribution tasks. The other experiment type uses the WidowX robot from the BridgeData V2 evaluations [52]. Here, four generalization types are defined that are called *visual* (unseen backgrounds, distractor objects, and the appearances of objects), *motion* (unseen object poses), *physical* (unseen object sizes and shapes), and *semantic* (unseen target objects, instructions, and knowledge from the Internet). OpenVLA mostly achieves higher success rates than the other models for all generalization types, except for *semantic* generalization where RT-2-X is slightly better [23].

3.2 Benchmarks and Simulators

Generalization capabilities are often evaluated to some extent directly in the work that introduces a new model. However, these evaluations are not standardized and often vary widely in the generalization dimensions that are looked at and the methods used for evaluation. Because of this, several benchmarks have been proposed.

RLBench is a simulated learning environment and benchmark for robot learning [19]. It is based on the robot simulation framework V-REP [36], and PyRep [18] which is a toolkit that brings a number of improvements to V-REP, including more realistic rendering and increased speed. RLBench features a single scene with the Franka Emika Panda arm. It is possible to retrieve robot proprioceptive data and visual data from a workspace and gripper camera. The authors introduce a concept of tasks, variations, and episodes. For example, a task could be "Pick up <object>" and variations of this task could be "Pick up the cube" or "Pick up the apple". For a variation, it is then possible to produce an infinite number of episodes in which object positions are changed. Demonstrations of these episodes can be generated by using the Open Motion Planning Library [42]. RLBench features 100 unique tasks out of the box. The authors propose a few-shot challenge to

evaluate general policies. The 100 tasks are split into 90 tasks for train data and 10 tasks for test data. While a policy can be trained arbitrarily with the train data, at test time it is only provided with 1, 5, or 20 demonstrations of the test tasks. Success rates are then measured for the 1-shot, 5-shot, and 20-shot cases [19].

The Colosseum is a benchmark designed specifically to evaluate generalization in robotic learning [31]. It is a modification of RLBench and uses a selection of 20 tasks out of the 100 implemented in RLBench. The authors define 14 *perturbation factors* that are applied to the tasks. *Perturbation factors* are grouped into the categories *manipulation object perturbation*, *receiver object perturbation*, *background perturbation*, and *physical perturbation*. The first two perturbation categories change the color, texture, or size of the respective objects. *Background perturbation* affects the light color, texture and color of scene objects, distractor objects, or the camera pose. *Physical perturbation* modifies physical properties of objects, like friction and mass. All of the *perturbation factors* are used in *The Colosseum* Challenge which consists of four phases. First, a training dataset with 100 demonstrations per task is generated without any perturbations. Then, a model is trained with this dataset. In the third phase, this model is then evaluated for each *perturbation factor* with a fixed set of 25 episodes per *perturbation factor*. The last phase consists of ranking the models based on the change in success rate for each *perturbation factor*. It is shown that success rates measured in simulation correlates with success rates measured in a real setup [31].

Meta-World is a benchmark that focuses on reinforcement learning in robotics [56]. It is based on OpenAI Gym [4] and offers 50 manipulation tasks implemented with the MuJoCo physics engine [46]. The following two categories of evaluation are employed: In the category *Multi-Task* a policy is evaluated for its ability to perform a certain number of tasks. There are versions for 1, 10, and 50 tasks. Parameters like object position and goal position are not randomized. For the other category *Meta-Learning*, five tasks are hold out to test a policy. The training is done with either 10, or 45 tasks and initial object and goal positions are randomized. Although Meta-World tests generalization in robotic manipulation, it is not applicable for the models we investigate in this work because it is not vision-based [56].

Robosuite is a simulation framework for robot learning [59]. It is built on the MuJoCo physics engine [46] and provides two APIs for setting up and running the simulation. The *Modeling* API can be used to define the simulation model that consists of a robot model, object models, and the workspace. Out of the box, there are 10 robots available. Robosuite provides a controller for each of the robots that can turn actions from various action spaces into the low-level torque commands used in MuJoCo. Objects can be loaded via MuJoCo's

native MJCF format or procedurally generated from geometric primitives. The second API is called the *Simulation* API and is used to interface with the MuJoCo physics engine. It accepts actions from a policy or an I/O device, and provides observations and rewards from the simulation. In terms of observations, cameras and proprioceptive measurements are supported. Robosuite also provides a number of pre-made benchmark environments. These include basic tasks such as block lifting, block stacking, pick and place, nut assembly, door opening, table wiping, and some two arm related tasks [59].

LIBERO is a manipulation benchmark based on Robosuite [59, 25]. Its main features are a task generation system and four task suites for benchmarking. The task generation pipeline consists of three steps. First, a large-scale dataset of language descriptions of human activities is processed to generate a set of task templates. These templates are then used in combination with objects available in the simulator to create task descriptions, e.g., the template "Open..." could be used to create the task "Open the drawer". The second step consists of creating an initial state distribution. A suitable scene for the task is selected, initial object placements are created, and the state of an object is generated, e.g., open/closed. In the third step, a task goal is specified based on the task description and the initial state. The goal consists of a conjunction of predicates that can be evaluated and terminate an episode when all predicates are true. The four task suites *LIBERO-Spatial*, *LIBERO-Object*, *LIBERO-Goal*, and *LIBERO-100* offer a fixed set of a total number of 130 tasks. The first three task suites all contain 10 tasks. *LIBERO-Spatial* is used to evaluate the knowledge of spatial information. While the goal and objects are the same across the tasks in this category, the robot has to manipulate one of two identical objects that differ only in their spatial relationship to other objects. Tasks in *LIBERO-Object* include different object types, and *LIBERO-Goal* employs the same objects in fixed spatial relations, but introduces multiple task goals. *LIBERO-100* is a set of 100 tasks from all categories described above [25].

ManiSkill is a robotics platform that supports GPU parallelization for simulation and photo-realistic rendering [44]. It is based on the SAPIEN simulator [55]. There are 12 premade environment types, including table-top manipulation and room-scale scenes. Out of the box, more than 20 robots are provided with controller implementations. ManiSkill introduces an object-oriented API for creating robotic environments and task building. Additionally, tools are included for trajectory replay, action space conversion, and domain randomization, e.g., to randomize camera poses. The GPU parallelization can be used to run multiple environments at the same time and is especially useful for collecting a lot of visual data. According to the authors, GPU memory usage is 2-3x lower than in other simulators such as IsaacLab [27, 44].

IsaacLab which is built on the IsaacSim platform is similar to ManiSkill [27]. It also provides GPU parallelization and enables photo-realistic rendering. The framework allows for custom task creation and is designed specifically for research areas in robot learning such as reinforcement learning and imitation learning. Similarily to ManiSkill, 16 robot platforms with their respective controllers are supported out of the box. IsaacLab also provides 20 benchmark tasks [27].

For our simulation platform RoboXim, we choose the Unity game engine [48]. While Unity is not specifically designed for robotic simulation, plugins are available that provide the necessary functionality such as importing robot URDF files [50] and communicating with ROS [37, 38, 35]. Unity can render realistic visuals, has an accurate, built-in physics engine, and gives the user full authority over how to configure the simulation [21]. Google DeepMind uses Unity for research in reinforcement learning due to its versatility and toolset to build diverse environments [53]. Their recently released generalist SIMA agent, which is similar to the VLA models described in section 3.1 is also evaluated in environments built in Unity [45]. While the DeepMind examples are not focused on robotics, several robotic simulators have been developed with Unity, including DoorGym [49], ThreeDWorld [14], and ManipulaTHOR [12].
In table 3.1 we compare simulators and benchmarks in regard to features that we require for our in-depth analysis and benchmarking of generalist robot policies. The task generation feature in the table refers to a system that can automatically generate tasks with language instructions. In most of the mentioned simulators it is only possible to manually add new tasks. Photo-realism is considered important because generalist models are mostly trained on real data [9]. In general, ManiSkill and IsaacLab are probably good choices for running and evaluating generalist policies. One issue with IsaacLab is that it consumes a lot of VRAM while generalist models usually require a lot of VRAM too. Therefore, on typical consumer GPUs it may become unfeasible to run IsaacLab and the model at the same time. When running headless and for a simple environment, RoboXim typically requires less than 200MB of VRAM with two cameras rendering images at a 256x256 resolution. However, the most important advantage of Unity is its versatility, user friendly programming experience and editor. Although ManiSkill offers an interactive GUI it can not be used for building environments and configuring every detail of the experiment like the Unity editor. Our RoboXim platform is described in detail in section 4.2.

Table 3.1: Comparison of features across different robotics simulators and benchmarks.

Feature	**RoboXim**	ManiSkill3	IsaacLab	RLBench	Colosseum	Meta-World	Robosuite	LIBERO
Photo-Realism	✓	✓	✓	✗	✗	✗	✗	✗
Editor GUI	✓	✗	✓	✗	✗	✗	✗	✗
Demonstration Generation	✓	✓	✓	✓	✓	✗	✓	✗
Task Generation	✓	✗	✗	✗	✗	✗	✗	✓
VLA Inference	✓	✓	✗	✗	✗	✗	✗	✗
Robot Embodiments	1	20	16	1	1	1	10	1
Min VRAM	200MB	~2GB[1]	8GB[2]	n/a	n/a	n/a	n/a	n/a

3.3 Taxonomy for Evaluating Generalist Robot Policies

Most of the works on generalist policies and benchmarks described in section 3.1 and 3.2 evaluate certain aspects of generalization. However, the methods and the particular categories of generalization vary and are not standardized. STAR-Gen is an attempt to establish a comprehensive and systematic taxonomy for generalization in robotics [15]. STAR-Gen defines generalization as perturbations from a base task. The perturbations are categorized into the modalities "visual" (visual input changes), "semantic" (language input changes), and "behavioral" (action output changes). Combinations of the mentioned modalities are also considered *categories*, e.g., a perturbation of the initial object pose

[1] Estimate based on a very simple environment without textures where 1.7GB of VRAM usage was measured [44].
[2] Taken from the Isaac Sim requirements [17].

belongs to the *category* "visual + behavioral", because both the input images and the required action outputs change. To further formalize STAR-Gen, the authors introduce *factors* which group together perturbations that affect tasks in a similar way. For example the *factor* "lighting" could encompass perturbations like light intensity and light color. In addition, the authors define *axes* which group together similar *factors* with common modalities. Therefore, each *axis* is part of a specific *category*. The axes are subjectively designed by the authors, e.g., the *axis* "Image Augmentations" consists of visual *factors* such as "lighting", or "image blur" and belongs to the *category* "visual". The introduced concepts of *factors*, *axes*, and *categories* form a hierarchy that allows to evaluate generalization at multiple granularities, i.e., *generalization to a factor*, *generalization to an axis*, and *generalization to a category* [15].
STAR-Gen is demonstrated by using it to benchmark several VLA models, including OpenVLA [23], MiniVLA [2], and a reimplementation of π_0 [34]. The evaluation is based on the Bridge V2 dataset [52] and four base tasks that are similar to existing tasks in the Bridge V2 data are selected. Initially, the VLA models are only trained on the Bridge V2 data. In addition, they are then co-fine-tuned with a few demonstrations of the base tasks. This ensures that the base tasks are in-domain and it also allows for flexibility in choosing the base tasks. During the evaluation, these base tasks are perturbed to create 55 task variations that cover 13 *axes* across five *categories*. In general, the success rate is found to be low for most of the tested *axes*. Especially in the "semantic" category, performance is low for all models. In conclusion, STAR-Gen is a useful taxonomy for a detailed analysis of generalization across various axes and categories, if generalization is understood as perturbations to a base task. We will further discuss this prerequisite and point to some issues when we formulate our own methods to evaluate generalization in chapter 4 [15].

4 Methods

In this chapter we present our methods to investigate generalist robot policies. First, we discuss how to evaluate generalist policies in section 4.1. Then, we provide a detailed description of the implementation of our simulation environment RoboXim in section 4.2.

4.1 Evaluation of Generalist Robot Policies

In chapter 3 we have presented numerous works that evaluate and benchmark generalization in various ways. While we do not fully adopt a specific approach, our method is influenced by previous works as discussed below. In order to structure our investigation into the generalization capabilities of models, we split the evaluation into the following four categories:

1) Zero Shot
2) Fine-Tuning and In-Domain Generalization
3) Cross-Domain Generalization
4) Out-of-Domain Generalization

The categories are explained in more detail in the following sections. The *Experiments* chapter 5 is structured around the four categories and contains various experiments for each category. In section 4.1.4 we also present our design for a benchmark.

4.1.1 Zero Shot

Zero shot capabilities are reported for most models by their original authors, but only for environments from the training data. We want to find out if any meaningful behavior can be observed in our new simulation environment. Therefore, we test openly available checkpoints of Octo [28], RT-1-X [9], and OpenVLA [23]. The experiments are described in section 5.1.

4.1.2 Fine-Tuning and In-Domain Generalization

Since we do not expect contemporary models to generalize to a new unseen environment, we fine-tune the models with demonstrations collected in our environment. This serves two main purposes: (1) we prove that our simulation environment can be used to fine-tune and evaluate generalist policies, and (2) we analyze the in-domain generalization capabilities of models. More precisely, generalization is here interpreted in the sense that a single model can complete multiple tasks it was trained on, i.e., perform multiple skills, recognize various objects, etc. Experiments of this category are described in section 5.2. We start with a single task "Pick up the cube" to verify our fine-tuning and evaluation pipeline. Then we test in-domain generalization for different colors, objects, and skills. Due to resource constraints we focus on the Octo-Small [28] model, but the same experiments are applicable to any model.

4.1.3 Cross-Domain Generalization

We introduce the term cross-domain generalization to describe the transfer of knowledge from the original training to our environment. A similar concept in the RT-2 evaluation is referred to as *emergent capabilities* [6]. In the RT-2 case it is evaluated if knowledge is transferred from the Internet-scale pretraining of the LLM. We focus on knowledge from the robotic data that the models have been trained on. The main question we aim to answer with these experiments is whether a model can be conditioned to a new environment to exhibit behaviors it was already trained to perform. To this end. we deliberately leave out instructions from the fine-tuning that exist in the original training data and then evaluate on these instructions. The results are reported in section 5.3.

4.1.4 Out-of-Domain Generalization

In this category, we want to analyze the ability of models to perform unseen tasks. Here, out-of-domain means that some aspect of the task was not seen in our fine-tuning. For example, a new camera angle, an unseen object, or a different interaction position in the scene. We also consider unseen combinations of otherwise seen instances as out-of-domain. If the tasks with instructions "Pick up the cube" and "Place the apple on the plate" are in our training data, then we consider "Pick up the apple" out-of-domain, even though an apple and the *pick up* skill were seen separately during training. We identify multiple dimensions, such as *object*, *skill*, *camera pose*, etc., to evaluate a model's capability to generalize across each dimension. Our goal is to create a benchmark that provides a score for each dimension to see how robust a model is and to help identify deficiencies for certain dimensions.
While the success rates of unseen tasks may already be useful for a benchmark, we are specifically interested in how well a model can generalize and transfer knowledge. Therefore, we need other success rates to compare against, e.g., performance of seen tasks. In section 3.3 we presented the STAR-Gen benchmark that always defines a base task [15]. Generalization for a dimension can then be measured by comparing the success rate of the base task to that of a variation of the task, where the base task was perturbed for the specific dimension. In our example above, the perturbed task would be "Pick up the apple". There are, however, multiple possible base tasks. The task "Pick up the cube" could be perturbed along the *object* dimension to replace the cube with an apple, or the task "Place the apple on the plate" could be perturbed along the *skill* dimension to replace the *place* skill with the *pick up* skill. This is an issue because the result of the comparison depends on the chosen base task. STAR-Gen may be useful to determine how robust a model is to perturbations, but we are more interested in analyzing the transfer of knowledge. Our solution is to compare the task against itself, but when it was seen in training. To this end, we collect two datasets and train two models. We call the first dataset *leave-out* dataset because certain tasks are left out. The second *baseline* dataset contains all tasks present in the *leave-out* dataset and in addition the tasks that were left out.

In this work, we do not design an exhaustive benchmark, but implement a simple version with only three dimensions. We select the dimensions *skill*, *object*, and *camera pose*. In order to create leave-out combinations, we combine all dimensions with each other, resulting in three experiments (*skill/object*, *camera/object*, *camera/skill*). For example, in the *skill/object* experiment certain combinations of skills and objects are not seen. We then evaluate these unseen tasks and compare the success rates to the same tasks performed by the *baseline* model. Based on this evaluation, a score is calculated that is attributed

to both dimensions involved. This score is the percentage ratio of the success rate for unseen tasks to the success rate for the same seen tasks. For example, in the *skill/object* experiment, the unseen task "Pick up the apple" achieves a success rate of 0.6 while the *baseline* model reaches a success rate of 0.8 for the same task. This means that the model generalizing reaches 75% of the baseline. This score is then attributed to the *object* and *skill* dimension. The total score of a specific dimension is influenced by all experiments in which this dimension is involved. For simplicity, we take the average of all the respective scores. This score only reflects the model's ability to generalize to unseen tasks. It does not provide information on the actual success rate. Therefore, we also provide the average success rate for unseen tasks across the dimensions.
Evaluations in related works are often only conducted with around 10 episodes per task and a single model [15, 28]. In our experiments, we notice that this is not enough to get accurate results. Typically, we use five seeds to train models and conduct 20 rollouts to evaluate a task, resulting in a total of 100 episodes per task. The experiments for this category are described in section 5.4.

4.2 Simulation Platform

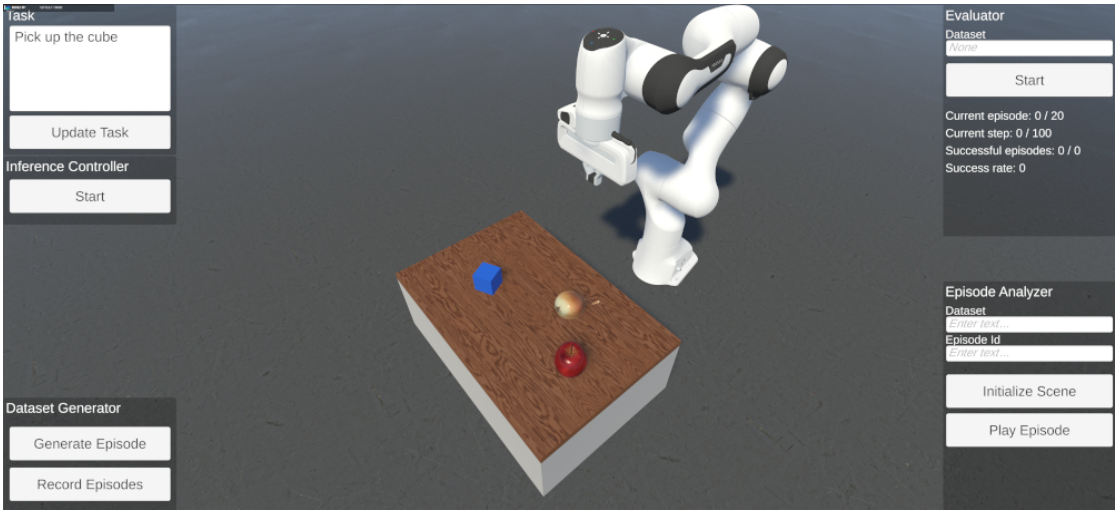


Figure 4.1: A screenshot of our simulation environment.

Our simulation platform is called RoboXim and is implemented with the Unity Game

Engine [48] and a screenshot is shown in figure 4.1. It is used for both generating demonstrations with an expert policy for training and for evaluating trained policies. Figure 4.2 gives an overview of the main functionalities which are explained in more detail in the following sections. Most functionalities are shared between episode generation and evaluation.

The core loop looks as follows: When a new episode is started first the environment is reset. This includes the task of the episode which is generated first. Task generation is explained in section 4.2.1. When the task is generated the rest of the environment is reset, see section 4.2.2. There is a resetter for each task that implements a reset function and prepares the environment for the task at hand, i.e., spawns the manipulation object. Next, the episode is executed. Either by an expert policy, described in section 4.2.3, or by inference, see section 4.2.5. During episode execution the episode can be recorded to acquire data for training which is explained in detail in section 4.2.4. Finally, section 4.2.6 describes evaluating episodes in order to determine if it was successful.

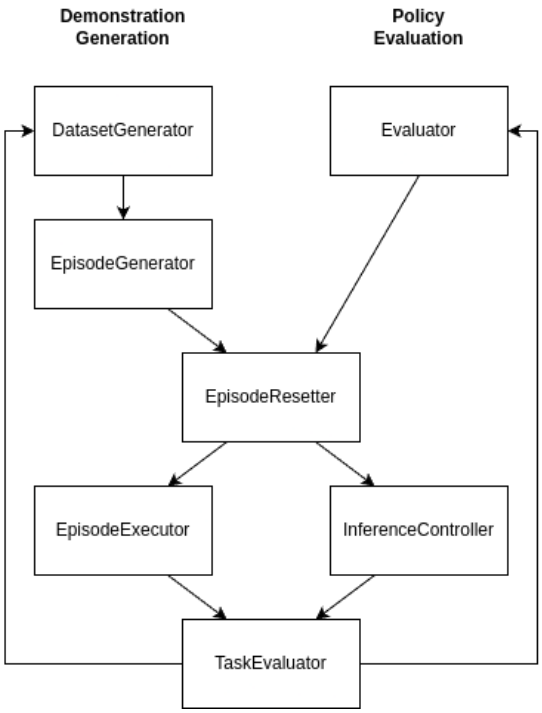


Figure 4.2: Overview of the main functionalities in RoboXim. The left side shows the process of generating demonstrations, the right side shows the process of evaluating a robot policy.

4.2.1 Task Generation

For the purpose of generalist robot policies a task specification could just be a natural language instruction. However, in our simulation environment a `Task` is a data structure that provides information to set up the environment and evaluate the outcome of episodes. Note that in this section and the following sections, the term task is used to describe this data structure and not exactly as defined in section 2.1.2. This is due to historical reasons of the implementation.
Fundamentally, a `Task` consists of a skill that determines what needs to be done. A skill usually has one or more `PhysicalObjects`. For example, the `PickUpObject` skill has a `PhysicalObject` that needs to be picked up. The `PhysicalObject` consists of an id, in order to spawn it from a database, a name, and optionally modifiers `PhysicalObjectModifier` that allow to change, e.g., the color or size of the object.

By consolidating everything in this task data structure, it is both possible to generate a natural language instruction and to have all information available to setup the environment accordingly, e.g., spawn an object with the correct color.
Instead of setting up each task individually, tasks are created by task generators. For each property, such as `PhysicalObject`, `PhysicalObjectModifier`, etc. there exists a corresponding generator class. This allows us to generate tasks randomly and enables a greater scalability because tasks do not have to be created manually. A simplified overview of the task data structure and its generation system is shown in figure 4.3.

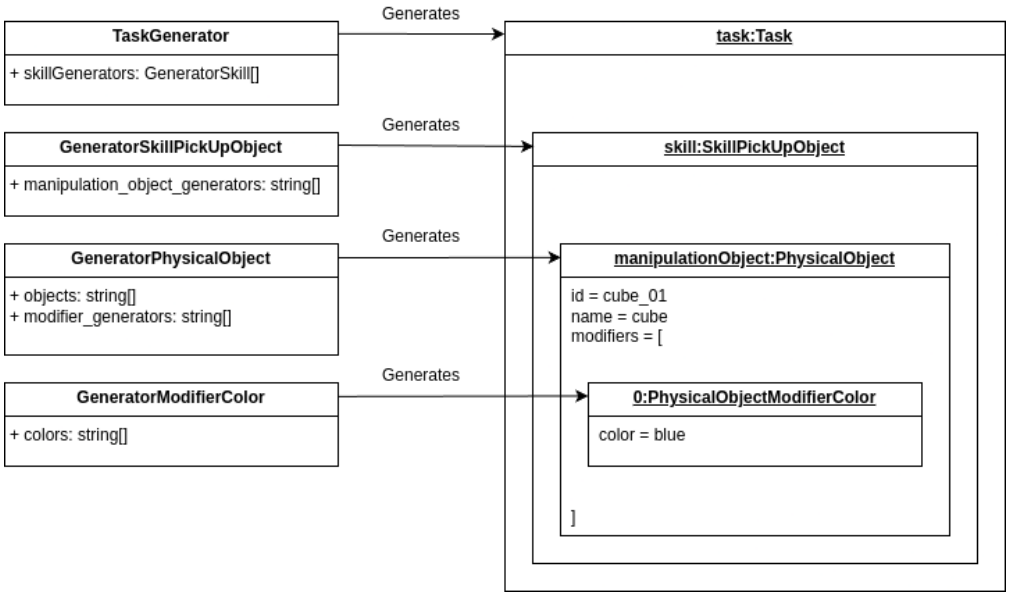


Figure 4.3: A simplified overview of the task data structure using the example of the PickUpObject skill and its generation.

4.2.2 Resetting the Environment

At the beginning of each episode, the environment is reset. Figure 4.4 provides an overview of this process. The first step is to generate a task as described in section 4.2.1. Dependent on the task, an appropriate `EpisodeResetter` is used to initialize the environment according to the task, or more precisely according to the skill of the task. For each skill there is one resetter class that implements the `EpisodeResetter` interface. This is

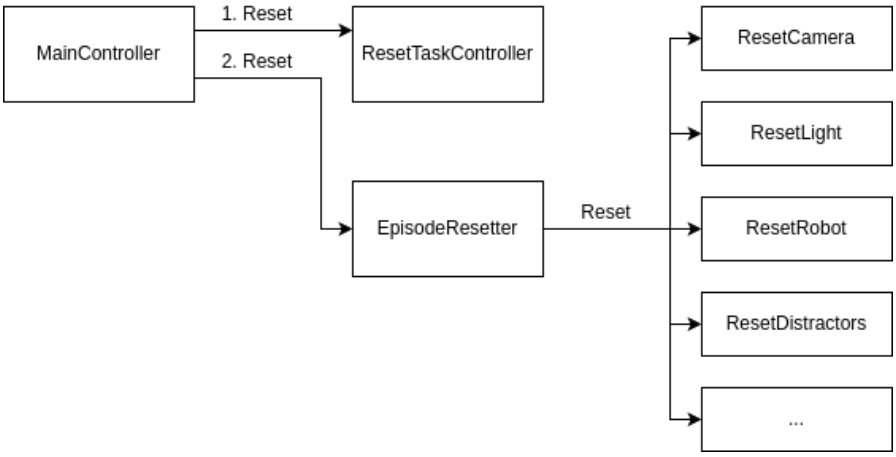


Figure 4.4: Overview of the reset process. First, the ResetTaskController is reset which triggers the task generation. Then, an EpisodeResetter initializes the environment by resetting ResetComponents shown on the right.

done because some skills have specific requirements for the setup of the environment. For example, when we have the *place* skill with the instruction "Place the cube in front of the apple", then we can not just spawn the objects randomly, but we have to make sure that the task is solvable, i.e., there is space in front of the apple, and we ensure that the task is not solved already, i.e., the cube is not placed in front of the apple initially. While some skills have these specific requirements, there is also a lot of functionality shared between all skills. Therefore, we encapsulate shared functionality in so-called `ResetComponents`. These components can then be reused by the `EpisodeResetter` of a skill to reset most of the environment. Some of the `ResetComponents` are shown in figure 4.4. They can be configured to enable domain randomization. For example, the `ResetCamera` component can place the camera at different poses, the `ResetLight` component can set the light to various intensities, colors, etc., and the `ResetRobot` component can initialize the joint positions, or set the end effector to an initial pose. Typically, each `ResetComponent` can be configured with a list of desired values, e.g., camera poses, light intensities, etc. When the component is reset, a value is then selected randomly or in the order of the list. In place of a list, the user can also configure a range of values, e.g., a range for light intensities, or a volume to randomly initialize the end effector in.

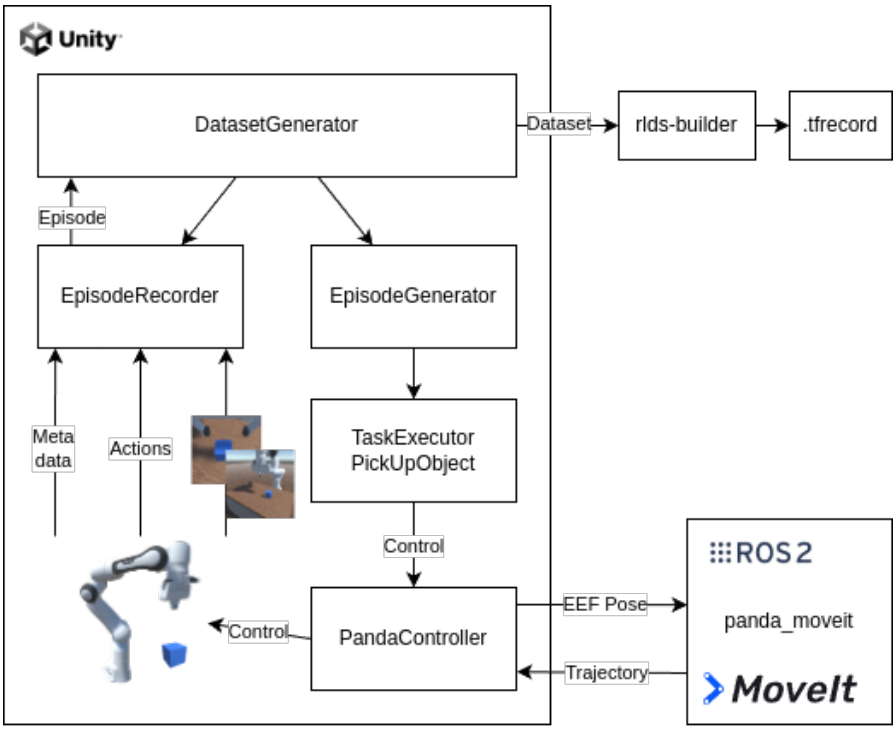


Figure 4.5: Overview of dataset generation and episode recording.

4.2.3 Demonstration Generation

In this section we describe the generation of demonstrations with an expert policy and the creation of a dataset that can be used for fine-tuning. Figure 4.5 shows an overview of this process. The main component is the `DatasetGenerator`. Here, the user can specify how many demonstrations to generate and where to save the dataset. The `DatasetGenerator` then uses the `EpisodeGenerator` to generate the desired number of demonstrations. The `EpisodeGenerator` first resets the environment as described in section 4.2.2 which also creates the task for the episode. Based on this task, the `EpisodeGenerator` determines an appropriate `TaskExecutor` to execute the episode with an expert policy. For example, if the task is to pick up a cube, we use the `TaskExecutorPickUpObject`. The `TaskExecutor` controls the robot through the `PandaController`. The `PandaController` provides functionalities to move the end effector to given poses by querying a ROS [35] service. We implement the service

using the motion planning framework MoveIt [41]. The service returns a trajectory that is executed by the robot in Unity.
While the episode is executed, the `EpisodeRecorder` records the episode. This is described in more detail in section 4.2.4. At the end of an episode, it is evaluated if the episode is completed successfully. This is required because in some cases the episode may fail due to an error in trajectory planning. If the episode was successful, the `DatasetGenerator` saves it to a file in `JSON` format. In order to use the generated dataset for training with common Python machine learning platforms, and especially for fine-tuning contemporary VLA models, the dataset is converted to `RLDS` format [33] using the `rlds-builder`[1].

4.2.4 Episode Recording

Episode recording is implemented in the `EpisodeRecorder` component and illustrated in figure 4.5. During the execution of an episode the `EpisodeRecorder` saves end effector actions (pose deltas) and images of cameras at a certain recording rate. In addition, meta-data about the scene is saved such as initial object positions, camera setups, initial robot state, etc., which allows us to exactly recreate a scene later. This recording of scene objects is facilitated by the so-called `RecordedParameters` components. The `EpisodeRecorder` finds all `RecordedParameters` components in the scene and serializes them. For example, the `RecordedSceneObject` saves the position and rotation of an object, the `RecordedCamera` saves the pose and field of view of the camera, and the `RecordedRobot` saves joint values. A user of our simulation platform can therefore select what to save by adding these components to objects. This system also allows us to easily record custom parameters by implementing a custom `RecordedParameters` component and adding it to an object in the scene.

4.2.5 Inference and Policy Evaluation

Evaluation of a generalist robot policy is handled by the `Evaluator` component and the process is shown in figure 4.6. In the `Evaluator` we can specify several parameters of the evaluation, e.g., the number of episodes and the maximum allowed steps. For each episode, the `Evaluator` triggers the environment reset described in section 4.2.2 before starting the `InferenceController`. The `InferenceController` collects the current instruction and camera images. These observations are then passed on to a Python script that

[1] Based on the bridge_rlds_builder [3].

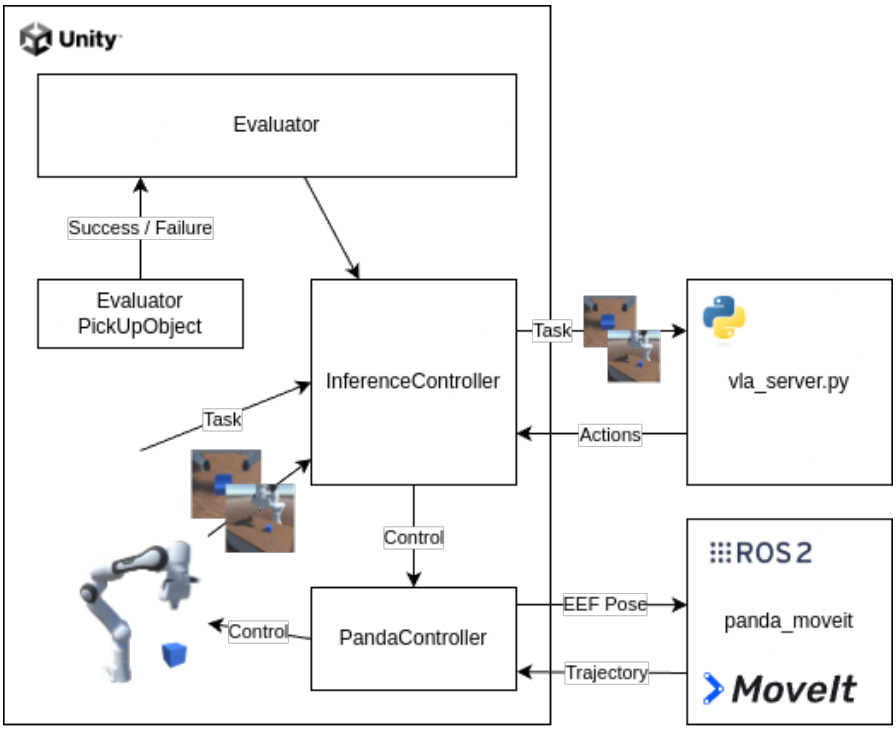


Figure 4.6: Overview of the policy evaluation process.

implements the inference for a VLA model. This VLA server returns the actions predicted by the model back to the `InferenceController`. The `InferenceController` component exposes many properties such as inference rate, window size (number of images in the image history), the action horizon, and which cameras to use. The actions returned from the VLA are delta poses. The `InferenceController` calculates the absolute end effector pose and controls the robot with the help of the `PandaController`. With the help of a `EvaluatorSkill` component we check whether the task is completed successfully and report the result to the main `Evaluator` component. This process is described in more detail in section 4.2.6.

4.2.6 Episode Evaluation

Episode evaluation is performed by an `EvaluatorSkill` component and is required for both demonstration generation (see section 4.2.3) and policy evaluation (see section 4.2.5). For each skill, there exists a specific `EvaluatorSkill` component that determines whether a task is completed successfully. In the example in figure 4.6 we show the `EvaluatorPickUpObject` that checks if the manipulation object is lifted to a certain height. Other examples would be the `EvaluatorPlaceObject` that checks if the manipulation object is close enough to a target location, or the `EvaluatorRotateObject` that verifies that the manipulation object is rotated to the desired rotation within a certain threshold. For most skills, we can configure these thresholds. For example, for the instruction "Rotate the cube clockwise by 90 degrees" we may choose a small threshold when recording demonstrations to ensure high data quality. However, when we test a policy, we may also consider an episode successful if the cube was only rotated by 80 degrees.

5 Experiments

In this chapter we conduct multiple experiments for the categories mentioned in chapter 4. We start with the zero shot category in section 5.1. Then, we fine-tune the models and conduct the in-domain generalization experiments in section 5.2. Next, we evaluate models for cross-domain generalization in section 5.3 and finally perform the out-of-domain experiments and benchmarking in section 5.4.

5.1 Zero Shot

In the best-case scenario and perhaps as the ultimate goal for generalist robot policies, it would be possible to just have a random environment and have the robot execute a given instruction. Although we do not expect this from contemporary models, we test them out of the box in our simulation in a custom environment and an environment that to some extent reenacts a setup from the Open X-embodiment data [9] that was used to train these models.

5.1.1 Custom Environment

In a first experiment, the publicly available checkpoints of Octo [28], RT-1-X [9], and OpenVLA [23] are tested for their zero-shot capabilities in our simulation environment. The environment shown in figure 5.1 consists of a simple table and a concrete floor. A cube is placed at a random position on the table and the models are instructed to pick up the cube. Along with the instruction, the models are provided with a camera image of the current and previous frames for both a workspace camera and a camera attached to the gripper of the Panda robot. However, not all models use the gripper camera or a history of images. Octo is the only model that supports a gripper camera, and it gets two images for history. While RT-1-X receives 15 images, OpenVLA does not support an image history

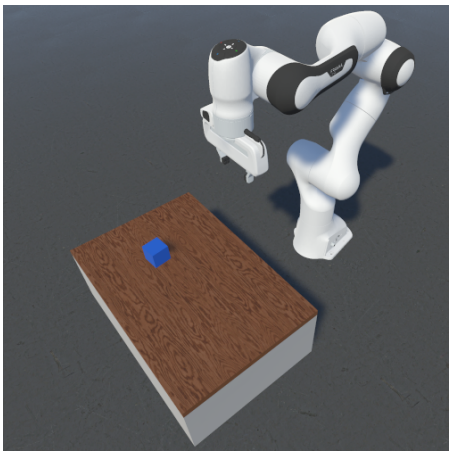


Figure 5.1: A screenshot of our simulation environment.

and only gets the current image.
For each inference call, Octo predicts four actions into the future. When these actions are executed in the simulation environment, no meaningful behavior can be observed. The end effector moves randomly and after some time usually ends up outside of the workspace. RT-1-X also moves the end effector outside of the workspace or does not move at all. OpenVLA usually moves around randomly and often ends up outside of the workspace too.

5.1.2 Reenacted Environment

In this experiment, we try to modify our environment to be somewhat similar to environments seen in the training of the generalist models. We find that in another simulation environment called SIMPLER [24], the RT-1-X model is sometimes able to pick up a coke can zero shot. Therefore, we imitate this environment in the following ways: The workspace camera is placed in a similar pose, the coke can model is imported in Unity to be used in place of the cube, and we use the same world axes definition by mapping the predicted actions on the axes definition used in Unity. However, the visuals of our environment are not changed and therefore are not similar to the SIMPLER environment. None of the models is able to pick up the coke can. Octo and RT-1-X often move down to the table, but not necessarily towards the coke can. No difference in behavior can be observed for OpenVLA compared to the first experiment described in section 5.1.1.

5.2 Fine-Tuning and In-Domain Generalization

In this section, we fine-tune the models with data collected in our simulation environment and then evaluate the models on tasks from this training data. This is done to show that our simulation can be used for data collection and evaluation. We also verify that the tested models can be fine-tuned as claimed by the original authors, and we get a first impression of their performance.

5.2.1 Pick up the Cube

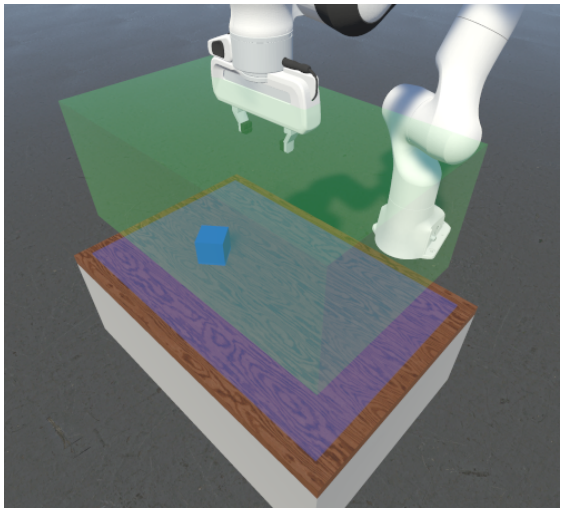


Figure 5.2: Workspace camera view of the scene. The cube is spawned randomly on the table in the blue area. The end effector is randomly initialized within the green box.

In order to check if fine-tuning with our simulation environment is feasible, we start with a single skill and a single object. The task is to pick up a cube from a table. There is no generalization, except that the cube and the end effector are initialized at random positions. The scene is shown in figure 5.2. A dataset of 200 episodes is collected and converted to RLDS format [33]. Then, Octo [28] is fine-tuned with the script released by the original authors. The model is trained for 100k steps with a batch size of 128. Other than that, the hyperparameters are left mostly unchanged, as the default configuration

has been shown to be a good choice for most fine-tuning scenarios [28]. Checkpoints are saved at intervals of 5k steps. Each checkpoint is evaluated with 50 rollouts in our simulation environment and the results are shown in figure 5.3. An episode is considered a success if the cube is picked up to a certain height, and failed if this is not achieved within 100 steps. At the first checkpoint, the success rate is low at 20%, but then immediately goes up to more than 90%, where it generally stays for the subsequent checkpoints. The last checkpoint at 100k steps can even achieve a success rate of 100%.

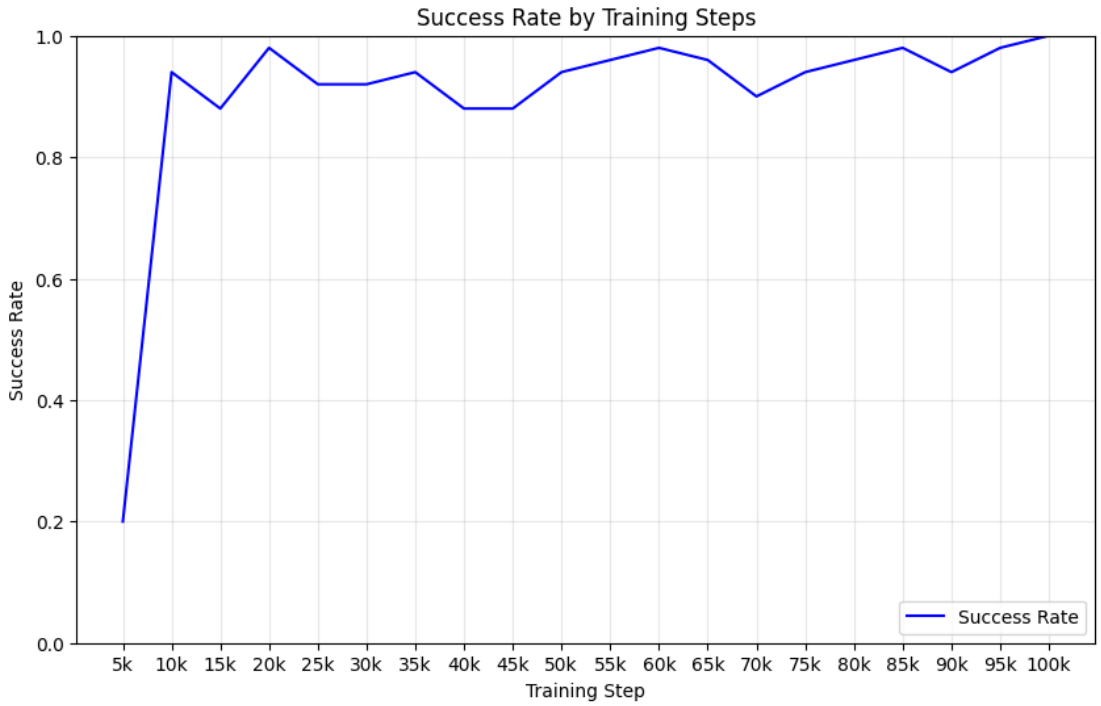


Figure 5.3: Octo success rate by training steps for the instruction "Pick up the cube".

Since the success rate already plateaus after 10k training steps, another experiment is conducted, where checkpoints are evaluated in 2k step intervals, and training is stopped after 20k steps. In addition, a second training run is conducted where the original axes definition from Unity is used, i.e., the axes are not converted before fine-tuning and converted back during inference. The results are shown in figure 5.4. Again, the success rate increases quickly, reaching more than 90% at the 6k checkpoint. There is no

considerable disadvantage of fine-tuning Octo to the Unity axes definition. However, in all following experiments the axis conversion is applied.

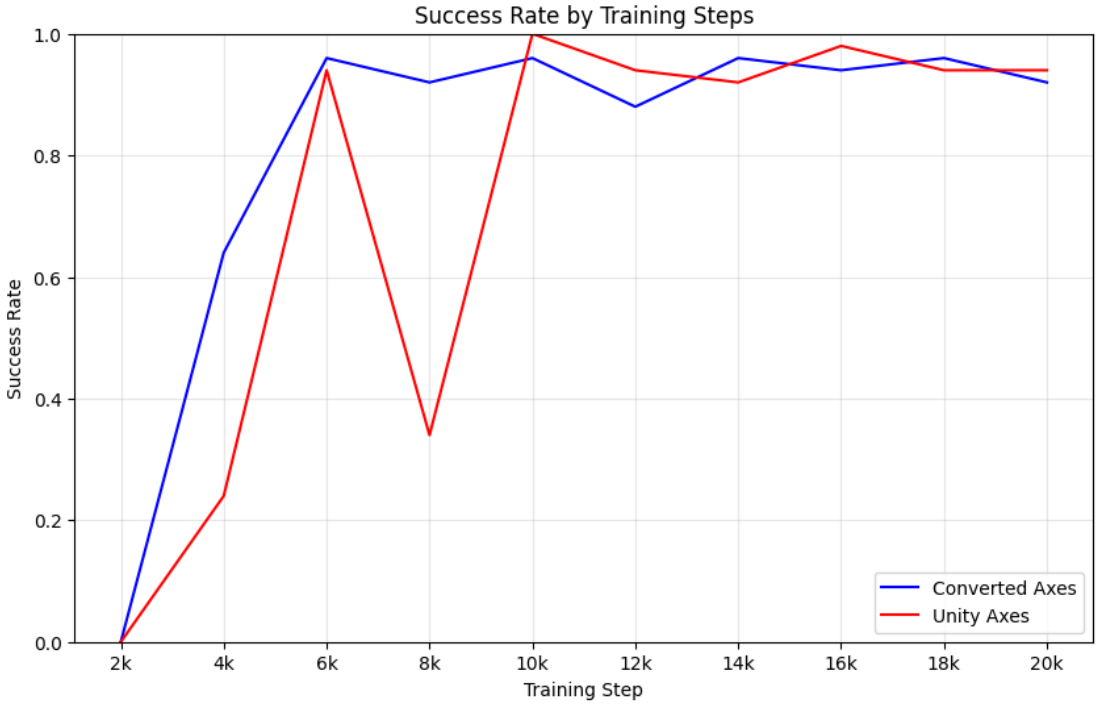


Figure 5.4: Octo success rate by training steps for the instruction "Pick up the cube" and comparison of two training runs, where one uses the Unity axes, and one with converted axis.

5.2.2 Multiple Colors

This experiment aims to check whether a model can distinguish between colors. The instruction is "Pick up the <blue, green, red, yellow> cube". In addition to the target cube, 1-3 distractor cubes of different colors are spawned. An example setup is shown in figure 5.5. A dataset of 400 episodes is collected, where each color is represented equally with 100 episodes. Then, Octo is fine-tuned with this dataset for 10k steps. Instead of relying on a single training run, the training is repeated for 5 seeds.
For the evaluation, we plot the mean success rate across seeds by checkpoints in figure

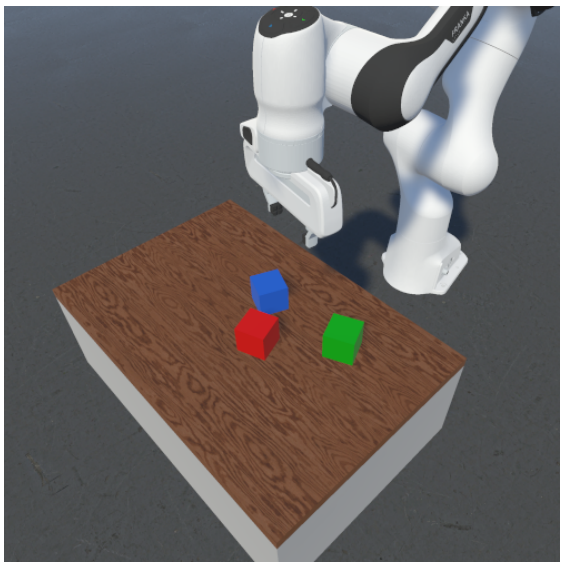


Figure 5.5: An example setup of the multiple colors experiment.

5.6. Each checkpoint is evaluated with 20 rollouts per seed. An episode is only considered successful if the cube with the correct color is picked up. The result is similar to the previous experiment in the sense that after 6k training steps the success rate does not improve by much. The best mean success rate is reached at the 9k checkpoint with 94%. This checkpoint is then used to compare performance by color. For each seed, we evaluate 20 episodes per color and show the results in figure 5.7. The success rate for all colors is approximately the same. We conclude that Octo is able to learn to differentiate between colors. Next, we will check the same for various objects.

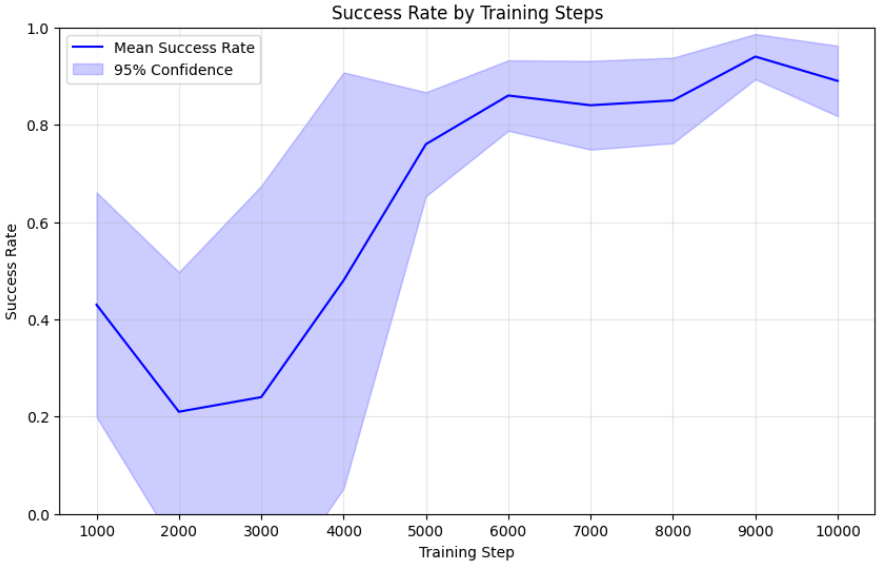


Figure 5.6: Mean success rate and confidence by training steps for picking up a cube in one of four colors.

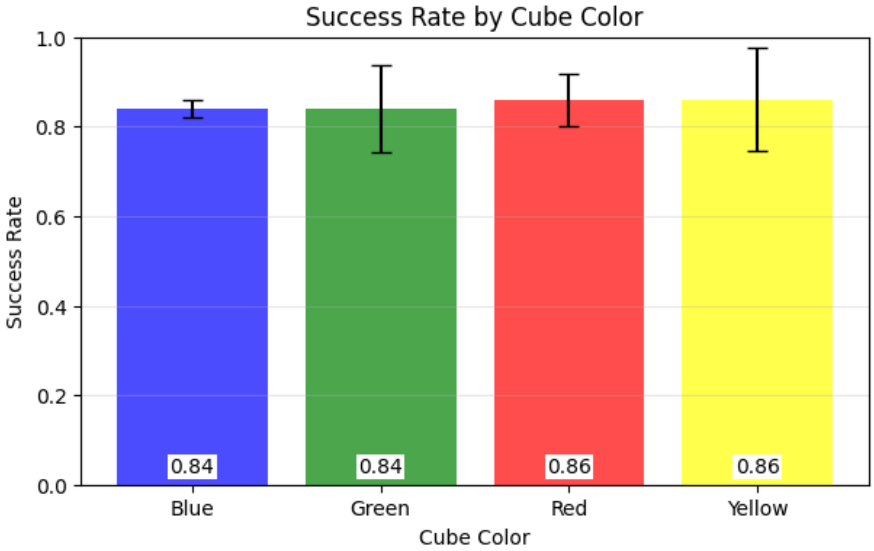


Figure 5.7: Mean success rate and standard deviation by color of the cube.

5.2.3 Multiple Objects

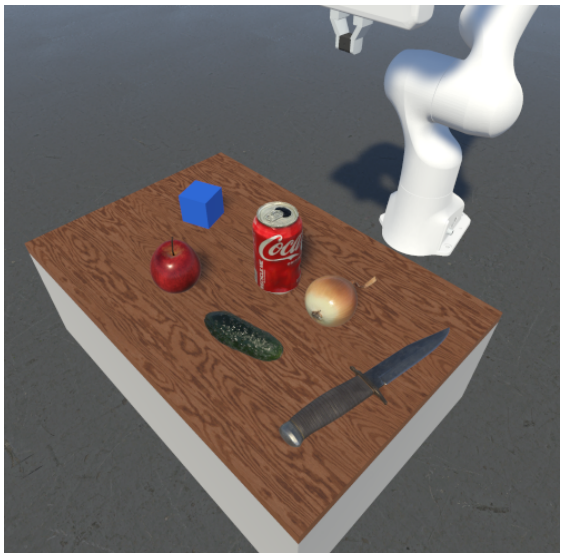


Figure 5.8: The objects used in the *multiple objects* experiment.

The general setup remains mostly the same as in previous experiments. We still only use the *pick up* skill, but instead of just cubes, the object to be picked up is one of the six objects depicted in figure 5.8. This object pool consists of a cube, an apple [13], a coke can [24], an onion [1], a cucumber [1], and a knife [54]. The objects were chosen to form a diverse set of shapes and colors. The cube is included because we already know from previous experiments that Octo can successfully pick it up. In addition to the target object, 1-3 distractor objects are spawned. For each object 100 demonstrations are generated, resulting in a dataset of 600 episodes total. The fine-tuning and evaluation is done similarly to the *multiple colors* experiment (see 5.2.2). However, since the overall complexity with multiple objects in various shapes increased, we fine-tune for 100k steps instead of 10k steps.
A checkpoint is evaluated with 20 episodes per seed looping through the object pool. The success rates for a number of checkpoints are shown in figure 5.9. Compared to previous experiments, the average success rate is lower at around 40%, and no substantial benefit can be observed for training longer than 10k steps. The 40k checkpoint achieves the best performance with 47%. To analyze the lower success rate, we break it down per object. The 40k checkpoint is used to evaluate each object with 20 episodes for the five seeds and the results are reported in figure 5.10. Significant deviations in success rate

can be observed between different objects. The cube performs best with 70% followed by the cucumber with 57%. The other objects only attain a success rate of around 30%. Qualitatively, we observe that Octo always attempts to grasp the correct object. The difference in success rate is therefore not due to object detection or a lack of language grounding. Rather, it is caused by the different physical colliders of the objects. Especially spherical objects, i.e., the apple and the onion, must be gripped precisely in the center or they roll away. The cucumber and handle of the knife are approximated with a box collider that is easier to grasp. However, compared to the cube, the handle of the knife is very flat. Additionally, due to their length, the knife and cucumber can only be grasped on their long sides, meaning that an inaccuracy in the orientation of the gripper is less forgiving than in the case of the cube. The coke can is often knocked over by the end effector, and Octo is usually unable to grasp it from this state.

Figure 5.9: Mean success rate and confidence by training steps for picking up various objects.

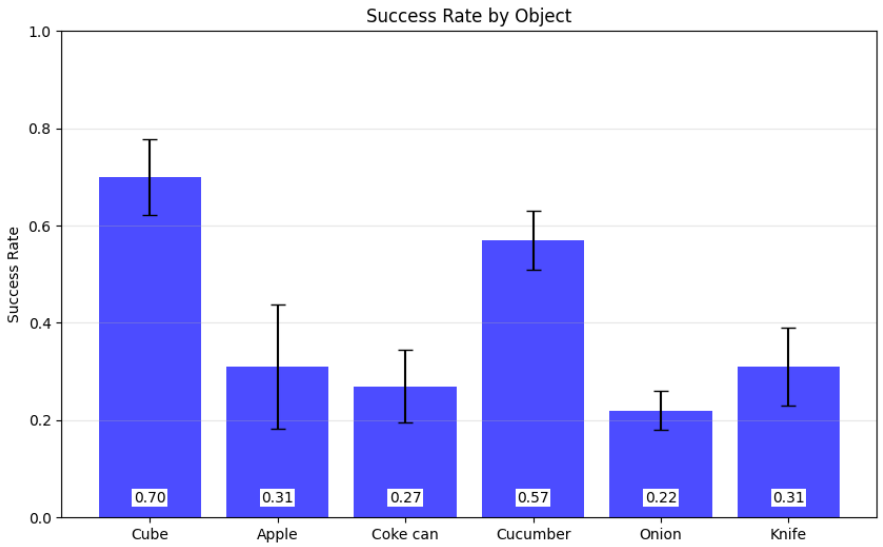


Figure 5.10: Mean success rate and standard deviation by object.

5.2.4 Multiple Skills

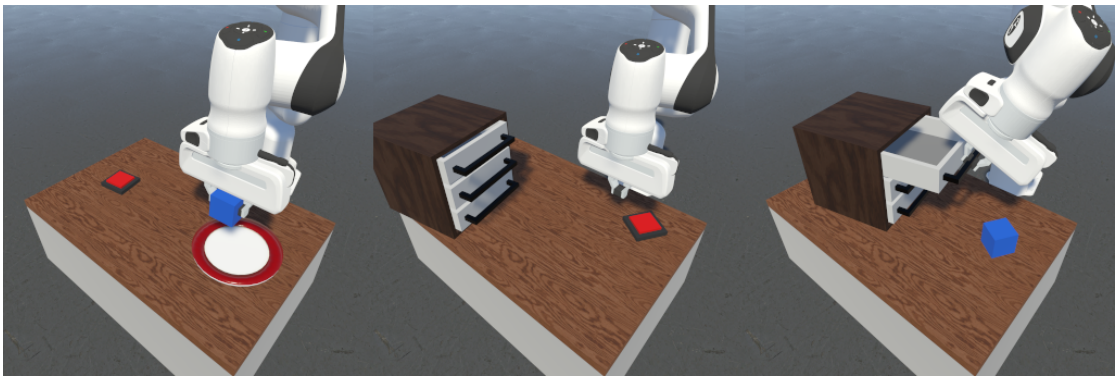


Figure 5.11: Examples of episodes for "Place the cube on the plate" (left), "Push the button" (middle), and "Open the top drawer" (right).

In this experiment, we want to find out how good Octo is at performing various skills. The following five instructions are used in this experiment: (1) "Pick up the cube", (2) "Place the cube on the plate", (3) "Rotate the cube clockwise by 90 degrees", (4) "Push the

button", and (5) "Open the top drawer". A few examples of the Panda robot performing these skills are shown in figure 5.11. In addition to the task relevant objects, there is always one distractor object in the scene that belongs to another tasks. For example, when the instruction is "Open the top drawer", we may also spawn a cube. For each skill, we generate 100 demonstrations. Octo is then fine-tuned with five seeds for 100k steps and the per checkpoint evaluation is shown in figure 5.12. As in previous experiments, we conduct 20 rollouts per checkpoint and per seed. However, the maximum number of steps to complete an episode is increased to 200 because some tasks take longer to complete. For example, to place a cube on a plate, first the cube has to be picked up, then moved to the plate, and finally placed down. The mean success rate for checkpoints is between approximately 60%-70% and no substantial benefit can be observer for training longer than 10k steps.

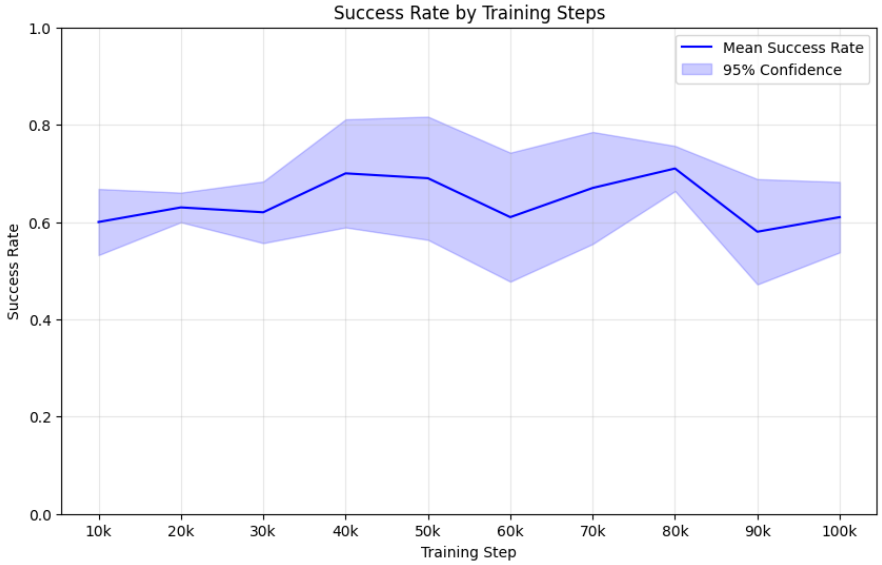


Figure 5.12: Mean success rate and confidence by training steps for performing various skills.

We use the best performing checkpoint saved after 80k training steps to evaluate the success rate of each skill. The skills are evaluated with 20 episodes each for every seed and the results are shown in figure 5.13. The *pick up*, *place*, and *push button* skills achieve a similar success rate of about 80%. Episodes in which the model is instructed to *rotate* the cube are failed more often. We consider such an episode successful if the cube is rotated

by at least 75 degrees. However, we can observe that the cube is sometimes placed down before that. The *open drawer* skill performs by far the worst. Only an average success rate of 11% is reached. Here, we observe that after grasping the handle of the drawer, the end effector often does not execute the movement required to open the drawer but instead moves down towards the drawer below. Across all skills, we find that generally the model attempts to perform the skill that was asked for in the instruction. Only in a few rare cases was the wrong object targeted, e.g., one instance was seen where the model tried to push the plate instead of the button.

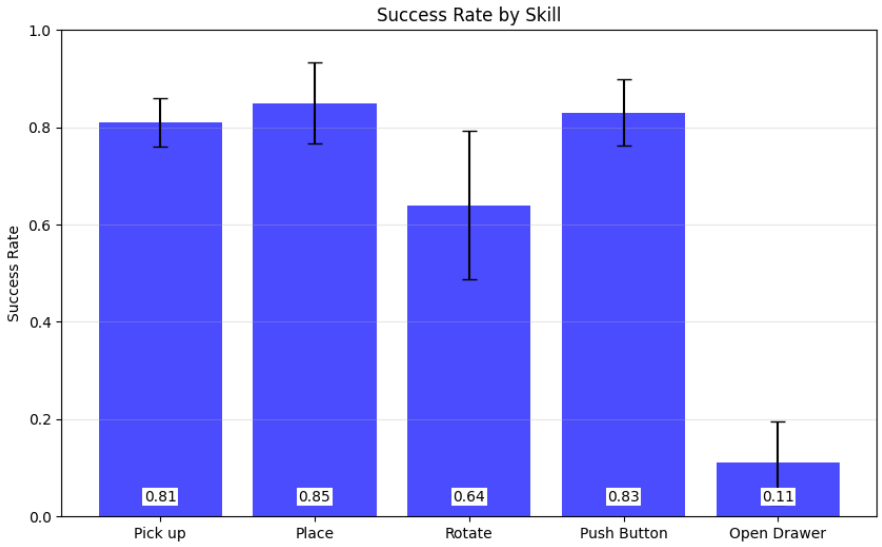


Figure 5.13: Mean success rate and standard deviation by skill.

5.3 Cross-Domain Generalization

After establishing in section 5.2 that models can be fine-tuned to our simulation environment, we now want to explore the question if these fine-tuned models can then transfer knowledge from their original training into our environment. In essence we want to find out if it is enough to just condition a model to the new visuals and action space, for it to access the abilities that it was trained for initially.
First, we use the Octo model from section 5.2.1 that is fine-tuned to pick up a cube. We replace the cube with a coke can and instruct it to pick it up. Usually, the end effector

is not moved towards the coke can and either ends up outside of the workspace or does a pick up motion without an object. When we add the cube as a distractor, Octo always picks up the cube even though it was instructed to pick up the coke can. In fact, the language instruction can be left empty or changed arbitrarily. Octo always picks up the cube regardless.

We suspect that Octo is overfitted on the pick up cube task. Therefore, we conduct a second experiment similar to the *multiple objects* experiment described in section 5.2.3. Instead of all six objects, we only include four objects in the training data. The apple and coke can are left out because these objects also exist in the pre-training data. We train Octo twice. For the first training run, we use the same config as usual and fully fine-tune Octo. In the second run, we only fine-tune the action head. The fully fine-tuned Octo model can still only pick up the objects seen in the fine-tuning. When instructed to pick the apple or coke can, usually one of the other objects is picked up. The Octo model where only the head was fine-tuned is not able to pick up any object, including the seen objects. The end effector movement is too inaccurate, meaning that no object is grasped. We therefore can not observe any cross-domain generalization with Octo.

5.4 Out-of-Domain Generalization

In this section we analyze the out-of-domain generalization across various dimensions and explore how models can transfer knowledge to unseen tasks. In the first two experiments we select a single dimension, specifically the interaction position (section 5.4.1) and the camera pose (section 5.4.2), to leave out certain configurations. For the following three experiments, two dimensions per experiment are selected. Then we leave out certain combinations across these dimensions. The selected dimensions are skills, objects, and camera pose. The resulting combinations are described in section 5.4.3 (skill/object), 5.4.4 (camera/object), and 5.4.5 (camera/skill).

5.4.1 Interaction Position

For this experiment the table is elongated to fit five interaction areas next to each other as shown in figure 5.14. The instruction is to pick up the cube. The cube is randomly spawned within one of the areas, but in the training data a single area is left out. The left out area is placed between seen areas. We conduct training runs for five seeds with Octo

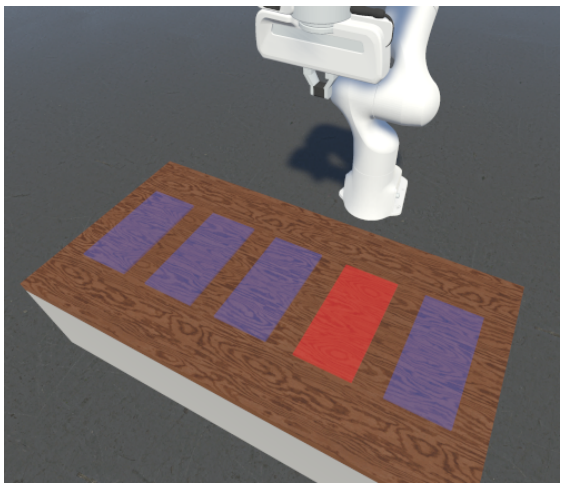


Figure 5.14: Screenshot of the *generalize interaction position* experiment. In the training data the cube can spawn within the blue areas but not in the red area.

and report the per checkpoint success rate for the seen interaction areas in figure 5.15. We evaluate 20 episodes per checkpoint, but only for a single seed. We then use the last checkpoint to compare the success rates for the seen areas with the unseen area. Here, all five seeds are evaluated with 50 episodes each, and the result is presented in figure 5.16. Surprisingly, the success rate for the unseen area is slightly higher than for the unseen areas. To investigate, the success rate for each individual area is plotted in figure 5.17. It turns out that there is one outlier in the seen areas, namely the rightmost position, which decreases the average of the seen areas. The reason for this remains unknown, but we can conclude that the success rate for the unseen area is very similar to the other areas.

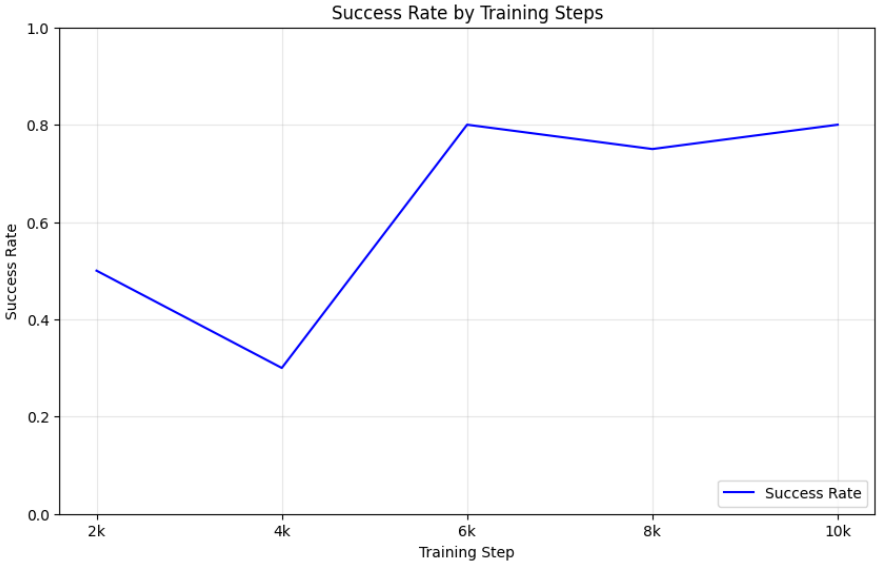


Figure 5.15: Octo success rate by training steps for seen interaction areas.

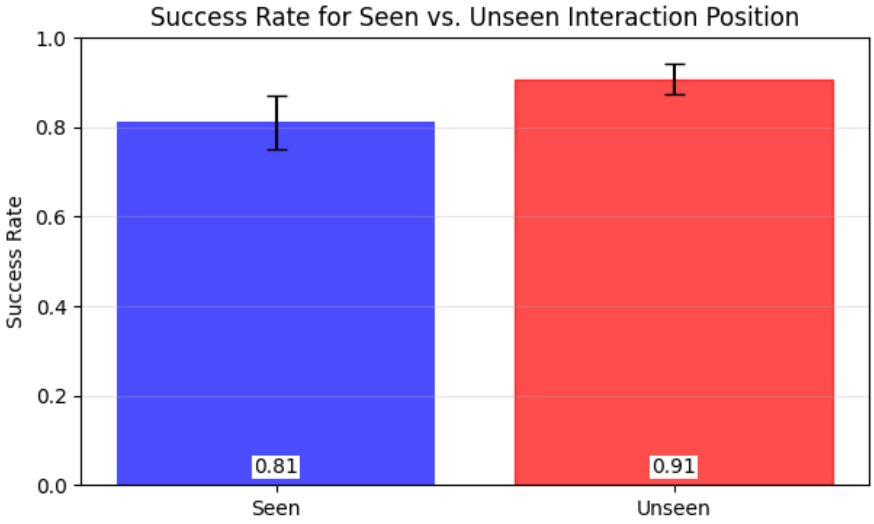


Figure 5.16: Comparison of mean success rate between seen and unseen interaction areas.

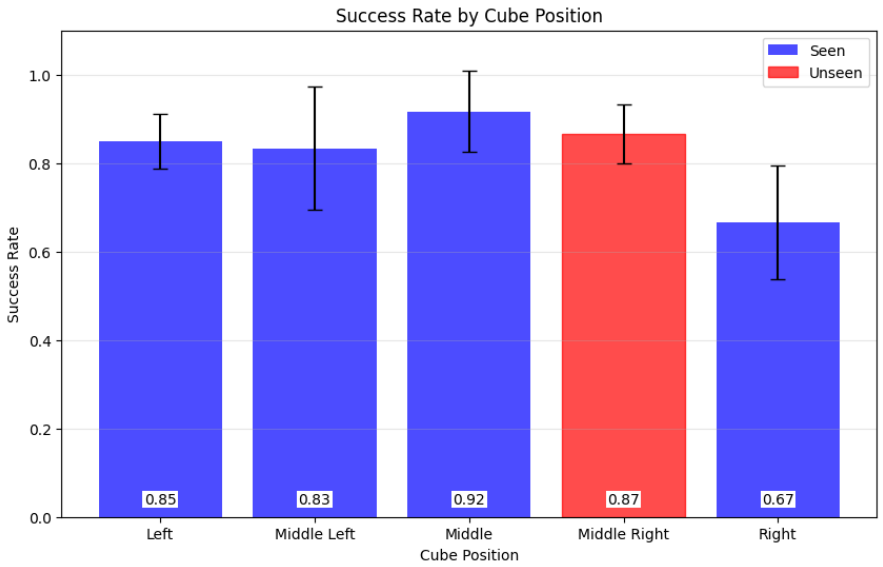


Figure 5.17: Mean success rate for each interaction area.

5.4.2 Camera Pose

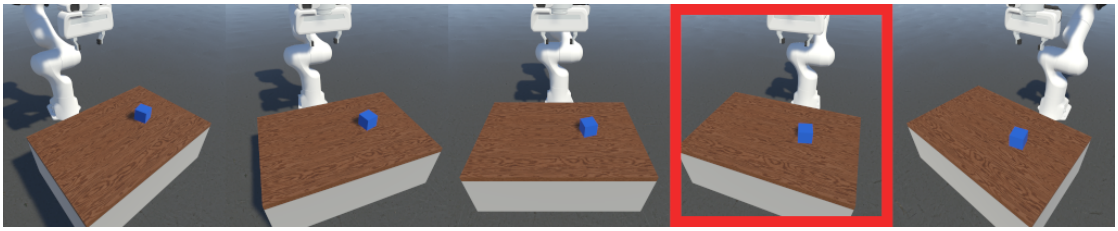


Figure 5.18: Views of the workspace camera at different poses. The view marked in red is not seen in the training data.

This experiment is analogous to the experiment described in the previous section 5.4.1, but instead of an interaction area, a camera pose is left out from the training data. The different camera poses are illustrated in figure 5.18. Octo is trained with five seeds for a maximum of 10k steps. We evaluate with a single seed and report the per checkpoint success rates in figure 5.19. Then, we use the last checkpoint to compare the success rate for seen and unseen camera poses. We conducted 50 rollouts per seed for both seen and

unseen poses, and show the results in figure 5.20. The success rates are practically the same and reach about 80%.

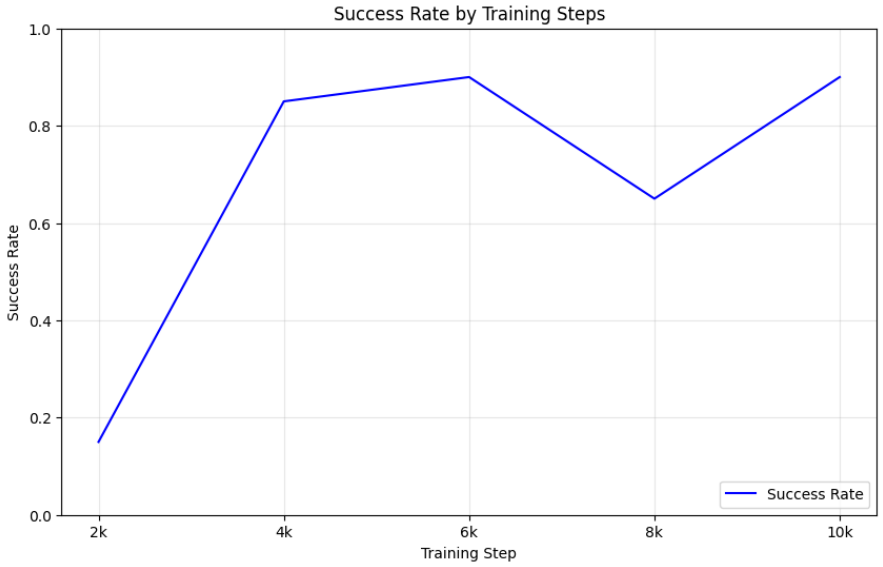


Figure 5.19: Octo success rate by training steps.

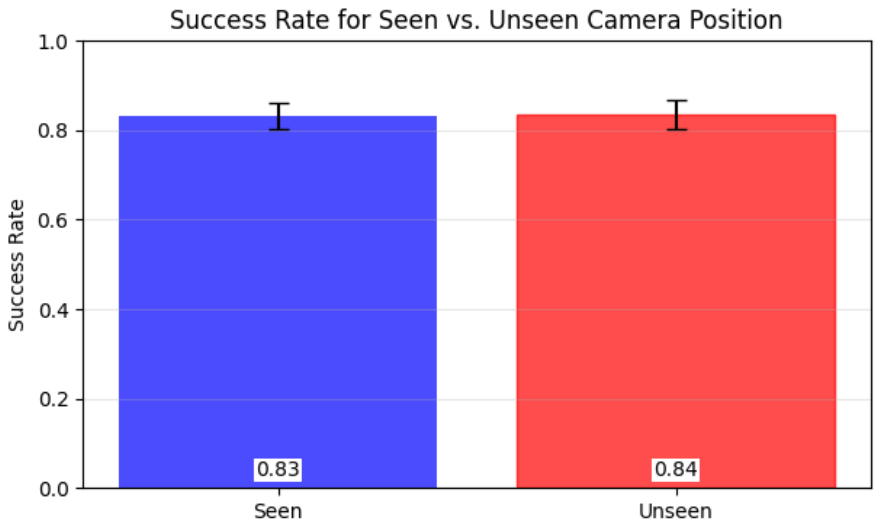


Figure 5.20: Comparison of mean success rate between seen and unseen camera poses.

5.4.3 Skill and Object

In this experiment we leave out certain combinations of skills and objects to evaluate if models can transfer knowledge across these dimensions. We select the *pick up*, *place*, and *rotate* skill. Furthermore, six objects are included, namely a cube, an apple, a coke can, an onion, a cucumber, and a knife. In the training, each skill is only performed with four objects. For the *pick up* skill, the cube and apple are not used. The coke can and onion are not used for *placing*, and the cucumber and knife are not *rotated*. In the evaluation we can then find out if the model can perform the skills with the respective left out objects. Additionally, we conduct a second training in which all objects are seen with each skill to act as a baseline. We aim for an equal distribution of skills and objects in the data. In the first dataset each skill is performed with four objects, and in the baseline dataset each skill sees all six objects. Therefore, we record 108 episodes per skill, since 108 is divisible by four and six. We end up with 27 episode per skill and object combinations in the first dataset, and 18 episodes per combination in the baseline dataset. Both datasets contain a total of 324 demonstrations.

Octo is fine-tuned separately with both datasets for five seeds and 10k steps. Then, we compare the success rates for seen and unseen combinations for both the skill dimension and object dimension. All skill and object combinations are evaluated with 20 episodes

per seed. First, a comparison for each skill is shown in figure 5.21. For each skill, only objects that were left out in the training data are considered, e.g., the "Unseen" bar for the *pick up* skill is the mean success rate of the instructions "Pick up the cube" and "Pick up the apple". The "Seen" bar on the other hand uses exactly the same two instructions, but the episodes are executed with the baseline model, i.e., the respective skill and object combinations were seen in training. For the *pick up* and *place* skill the success rate drops slightly if the objects were not seen in training. While the success rate for the *rotate* skill is very low in general, it is almost never completed successfully when the objects were not seen in training. Only a single episode out of the 100 evaluated was successful.

Next, we plot the same evaluation but broken down by objects in figure 5.22. For most objects the success rate drops when the respective combination of object and skill was not seen in training. The only exception is the cube, where surprisingly the success rate is slightly higher when the instruction "Pick up the cube" was not seen in training. In figure 5.23 we compare the mean success rate of all unseen combinations with the success rate of the same combinations when seen in training. The mean success rate of episodes where combinations of skills and objects were not seen in training reaches 72% of the baseline, i.e., the same combinations, but seen in training. This value of 0.72 is used in our generalization benchmark and is attributed to both the skill dimension and the object dimension.

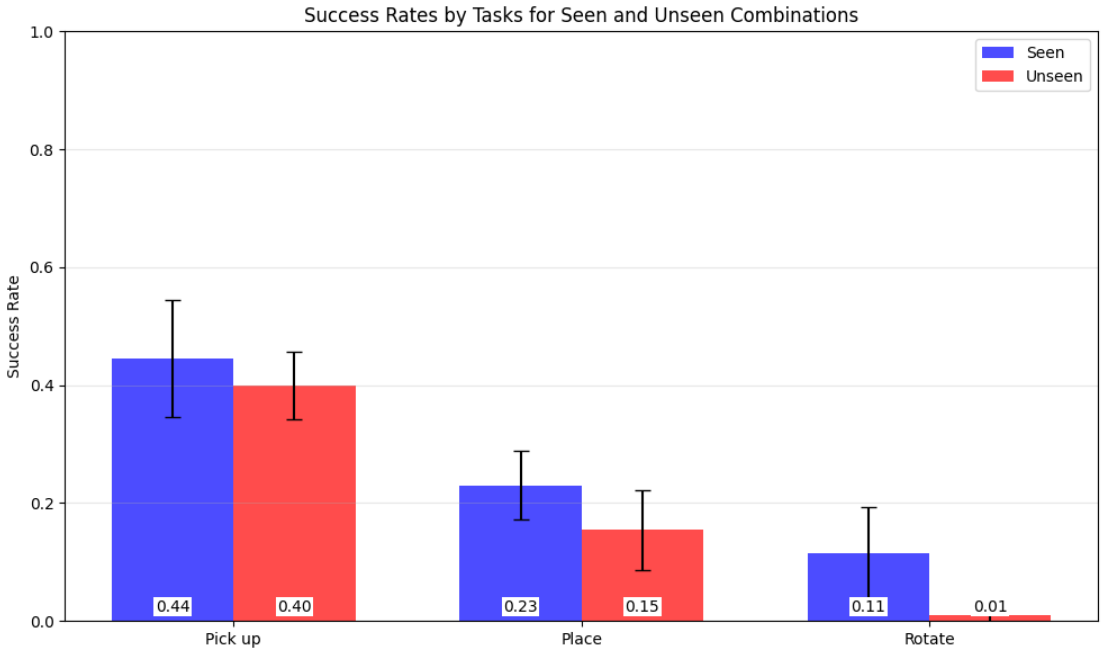


Figure 5.21: Octo success rates for seen and unseen combinations broken down by skills.

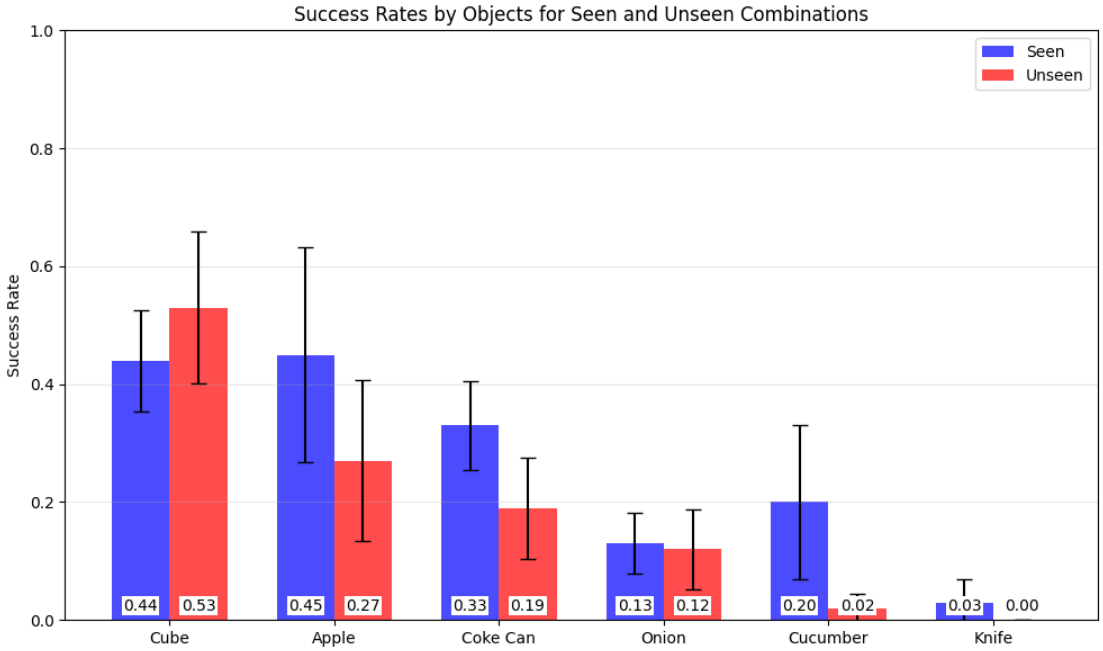


Figure 5.22: Octo success rates for seen and unseen combinations broken down by objects.

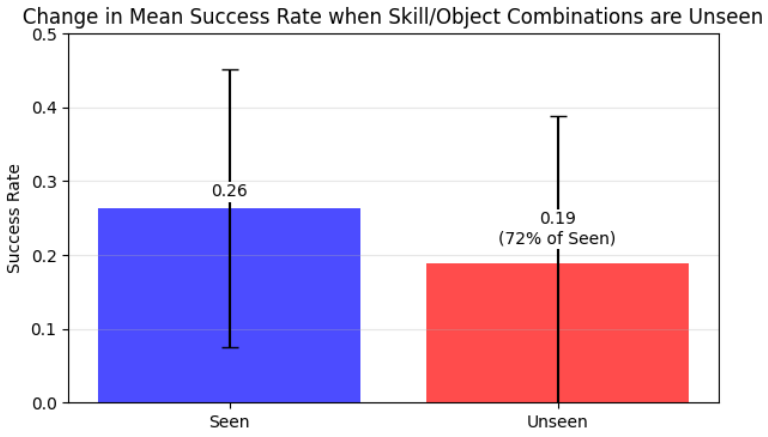


Figure 5.23: Mean success rates across all unseen skill/object combinations compared to the same combinations when seen in training.

5.4.4 Camera Pose and Object

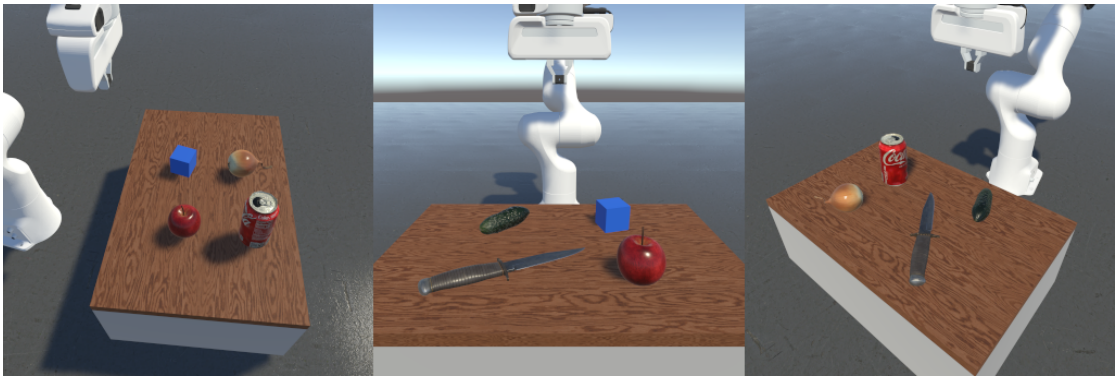


Figure 5.24: Camera views in the *camera pose and object* experiment with the objects that are seen in combination with the respective camera angle.

For this experiment we select the two dimensions *camera pose* and *object*. The methodology is the same as in the previous experiment (see section 5.4.3). We select three camera poses to combine with six objects. The camera poses and combinations with the seen objects for each camera pose are shown in figure 5.24. In episodes viewed from the right camera pose, the cube and apple are not seen. For the camera pose in the middle, the coke can and onion are left out. The cucumber and knife are not seen from the left camera angle. The training datasets in total contain 324 episodes with 108 episodes per camera pose and 54 episodes per object. Each combination is seen 18 times in the *leave out* dataset and 27 times in the *baseline* dataset.

After we fine-tune Octo with 5 seeds for 10k steps, we evaluate each unseen combination with 20 rollouts per seed. The same combinations are also evaluated with the *baseline* model where these combinations were seen in training. The results are reported broken down by camera pose in figure 5.25 and for each object in figure 5.26. For all camera poses the success rate is slightly lower when the combination of camera pose and object was not seen in training. The same is true for each object except for the cube. Figure 5.27 shows that across all unseen combinations the average success rate is 0.23 which is 80% of the average success rate for the same combinations when seen during fine-tuning.

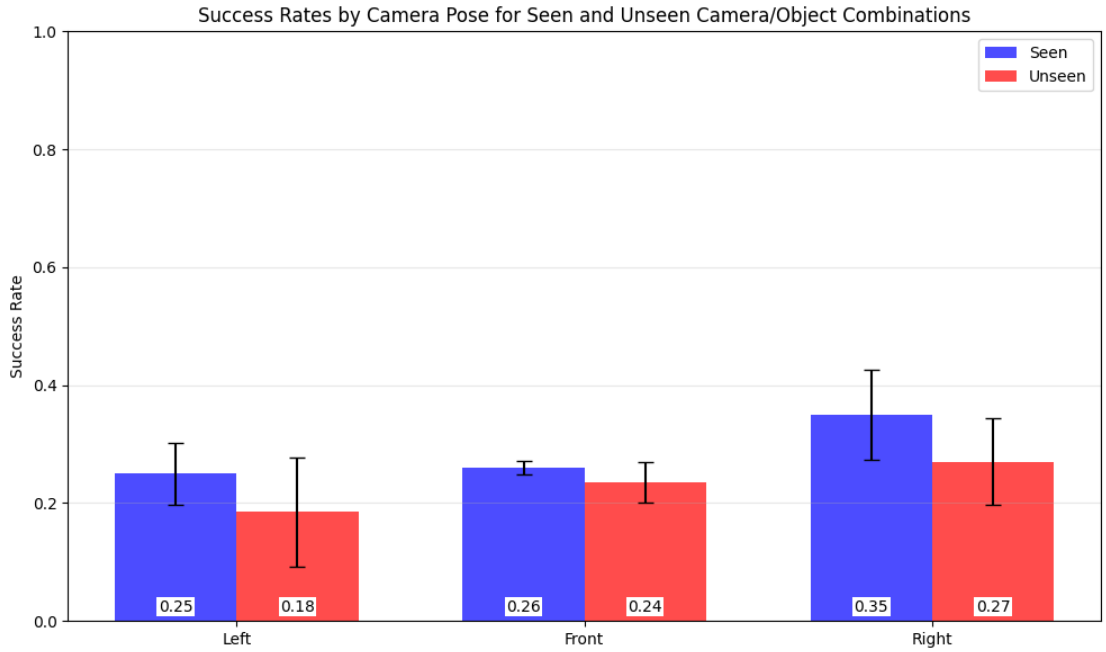


Figure 5.25: Octo success rates broken down by camera poses for unseen camera/object combinations compared to the same combinations when seen in training.

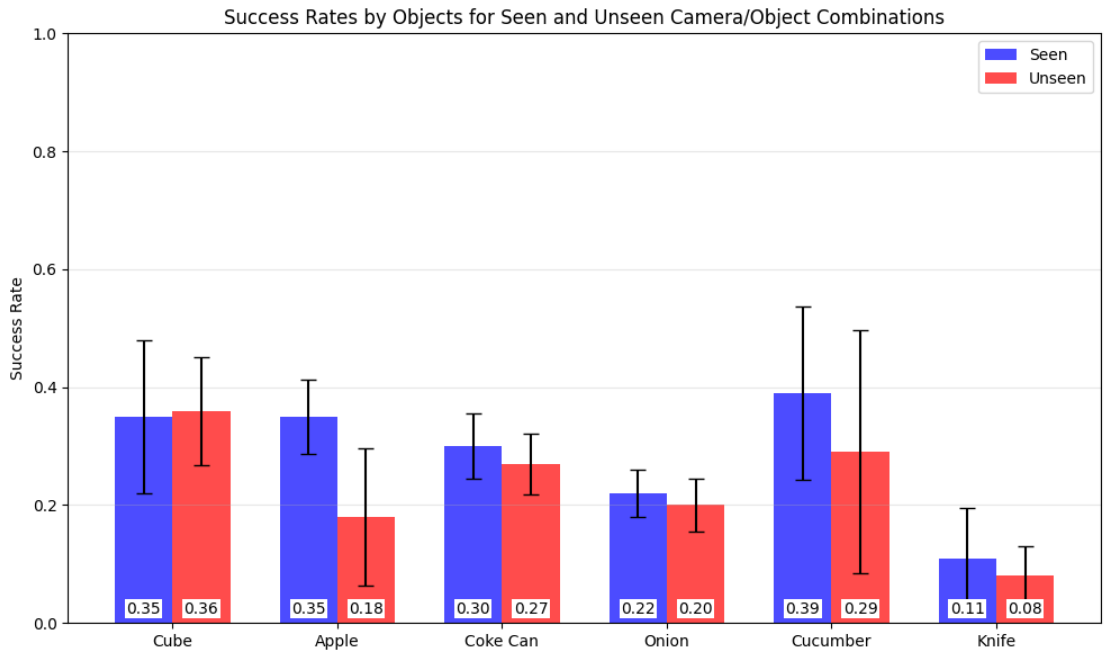


Figure 5.26: Octo success rates broken down by objects for unseen camera/object combinations compared to the same combinations when seen in training.

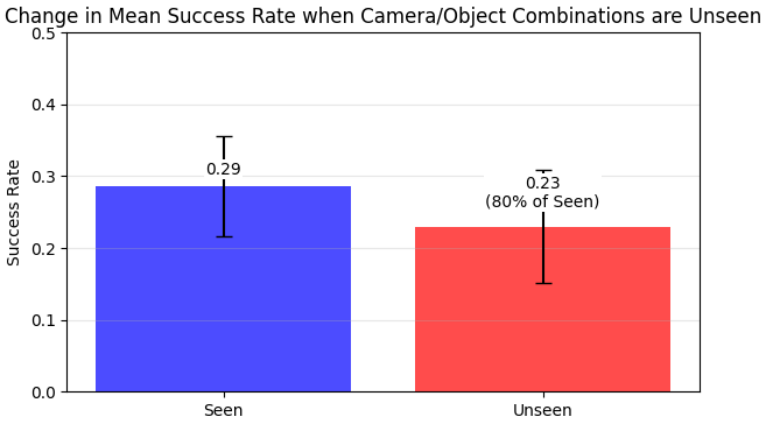


Figure 5.27: Mean success rates across all unseen camera/object combinations compared to the same combinations when seen in training.

5.4.5 Camera Pose and Skill

This experiment is very similar to the experiment in section 5.4.4, but instead of objects we select skills to combine with the camera poses. The camera poses are the same as in the previous experiment and shown in figure 5.24. Instead of six objects, we only select three skills. Those skills are the *pick up*, *place*, and *rotate* skills. In all skills we use a blue cube as the only object. The exact instructions are "Pick up the cube", "Place the cube on the plate", and "Rotate the cube clockwise by 90 degrees". For each camera pose one skill is not seen. The *pick up* skill is not seen from the *right* camera pose, the *place* skill remains unseen from the *front*, and the *rotate* skill is not seen in combination with the *left* camera angle. Since we have two skills per camera pose in the *leave out* dataset and all three skills in the *baseline* dataset, the number of episodes per camera pose must be divisible by two and three to ensure equal distribution of data. We choose 102 episodes per camera pose. Therefore, in the *baseline* dataset we end up with 34 episodes for each camera pose and skill combination. The *leave out* dataset contains 51 episodes per combination.

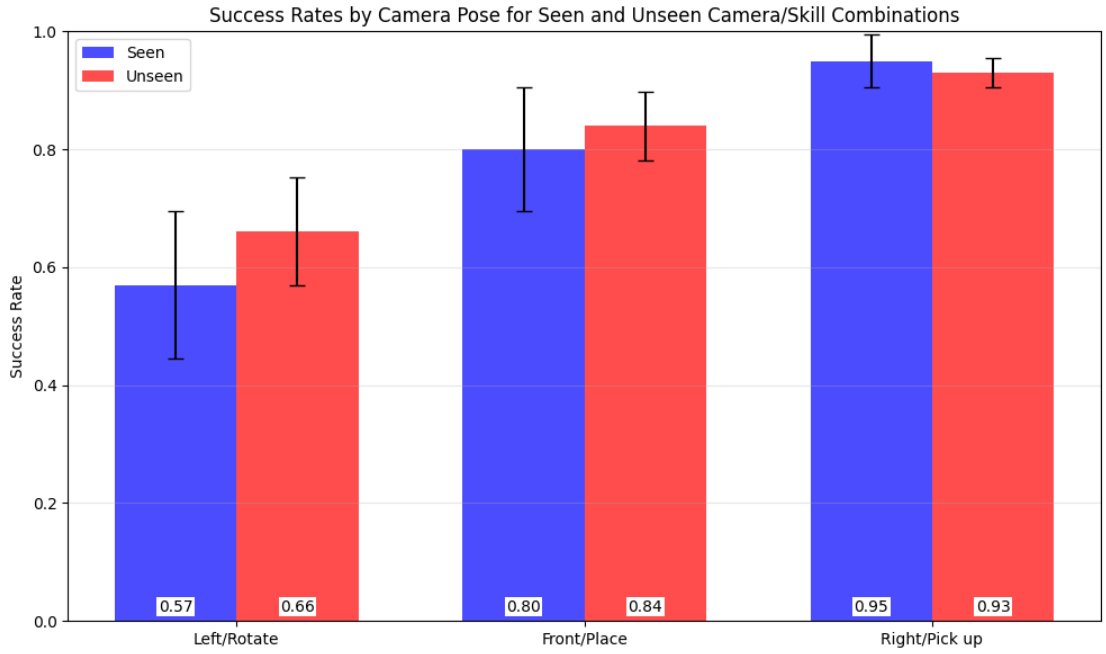


Figure 5.28: Octo success rates for unseen camera/object combinations compared to the same combinations when seen in training.

As in previous experiments, Octo is fine-tuned with five seeds for 10k steps. Each unseen combination of camera pose and skill is evaluated with 20 rollouts per seed. The results for both the unseen combinations and the respective baselines are shown in figure 5.28. Since there are only three unseen combinations, a single figure is sufficient for both dimensions. The success rates of the unseen combinations and their baselines are very similar. In the case of the *left/Rotate* and *Front/Place* combinations the success rate is even slightly higher than for their baselines. The average difference in success rates across all combinations is shown in figure 5.29. Interestingly, the unseen success rate is even slightly higher than the baseline.

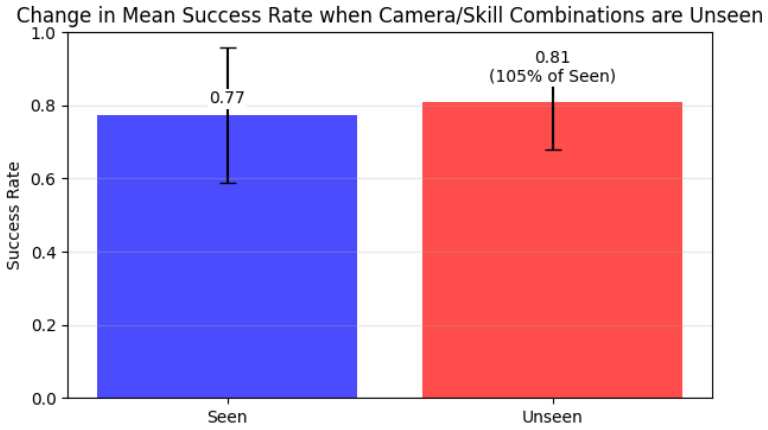


Figure 5.29: Mean success rates across all unseen camera/skill combinations compared to the same combinations when seen in training.

6 Discussion

In this chapter the results from the experiments in chapter 5 are discussed. We start with the zero shot and cross-domain experiments in section 6.1 before continuing with an examination of the fine-tuning in section 6.2. Finally, we present the results for our benchmark in section 6.3.

6.1 Zero Shot and Cross-Domain Generalization

According to our experiments, both the zero shot and cross-domain generalization do not work with the tested models. In the following sections, we discuss possible reasons for this.

6.1.1 Action Normalization

We note that especially with respect to zero shot generalization, the original authors do not expect their model to work in an unseen environment without fine-tuning [28]. Zero shot generalization is only reported for environments that were part of the training data. The models always work with normalized actions, and the normalization function depends on the environment. During inference actions are unnormalized with a key specific to the environment that is used for the evaluation. Since we introduce a completely new environment, such a key does not exist. In our simulation, we try different values to scale the actions. This can prevent end effector poses being immediately out of bounds, but it does not lead to meaningful behavior.
A related issue is that we do not know how to define the axes. Unity uses a left-handed coordinate system, while in robotics a right-handed coordinate system is often used, including in some of the X-embodiment datasets [9]. When we map the actions accordingly

to the axes in Unity we observe in some cases that the end effector stays within the workspace bounds for longer.

6.1.2 Data Quality

Another problem may be the overall quality of the X-embodiment data that is used to train the generalist models. As an example, we investigate a large dataset featuring the Panda robot and find a number of issues. The dataset contains more than 3000 episodes with similar tasks used in our experiments and is called *taco_play* [39, 26]. We report some issues that we found below.

Issue 1: Episode Continues After Task is Done

This issue is present in many episodes and is here demonstrated for the instruction "push right the yellow block". Some images of the episode are shown in figure 6.1. First, the robot pushes the cube to the right as instructed, but then it moves in the opposite direction and up. This introduces contradictory training data because for the same instruction, in one instance the gripper moves towards the cube and in another instance the gripper moves away which is essentially an incorrect action.

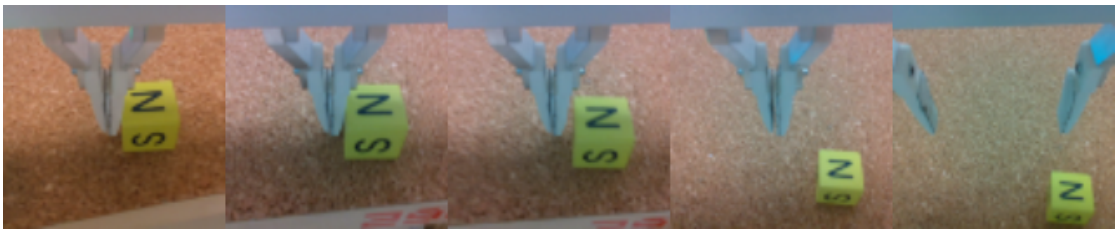


Figure 6.1: Sample images from an episode with the instruction "push right the yellow block" [39, 26]. The episode continues after the task is finished.

Issue 2: Inaccurate Control

This issues is related to the previous issue. Figure 6.2 shows an episode for the instruction "turn off the blue led light". The human teleoperator first misses the button before correcting the mistake and hitting the button. As in the previous issue, this introduces

contradictory data, i.e., for the same instruction, the gripper moves away from the button and also towards the button.

Figure 6.2: Sample images from an episode with the instruction "turn off the blue led light" [39, 26]. The gripper misses the button at first.

Issue 3: Camera Placement

Figure 6.3 shows the workspace camera in an episode with the instruction "go push the pink block into the drawer". The problem is that the pink block is nowhere to be seen because it is occluded by the robot.

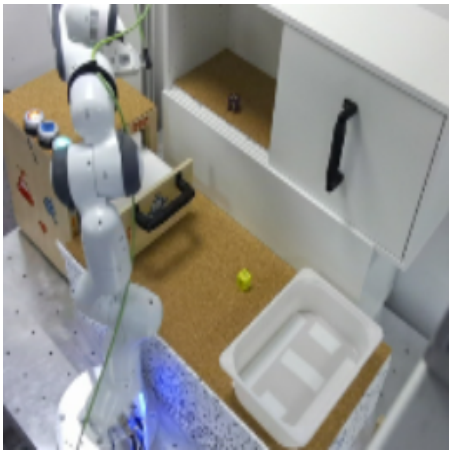


Figure 6.3: Workspace camera view in an episode with the instruction "go push the pink block into the drawer" [39, 26]. The robot occludes the manipulation object.

Issue 4: Wrong Language Instruction

We discover some episodes where the task performed in the episode does not match the language instruction at all. For example, in one episode with instruction "place the yellow object in the drawer" the robot just moves towards a button.

Issue 5: Indistinguishable Visuals

Figure 6.4 shows an image of an episode with the instruction "push the green button to turn off the green light". There are two buttons in the view. One is blue, while the other button is supposed to be green. However, the buttons appear almost identical, making it difficult for a model to properly learn the task.

Figure 6.4: Image of an episode with the instruction "push the green button to turn off the green light" [39, 26]. However, there are two buttons that appear almost identical.

Issue 6: Action Representation of Rotations

In figure 6.5 we plot the actions of a rotation axis for all steps in an arbitrary episode. Presumably, these are absolute rotations in radians. The action value jumps between $+\pi$ and $-\pi$. While $+\pi$ and $-\pi$ may actually represent the same rotation, this introduces ambiguity in the training data, since there are two very different actions for the same desired rotation.

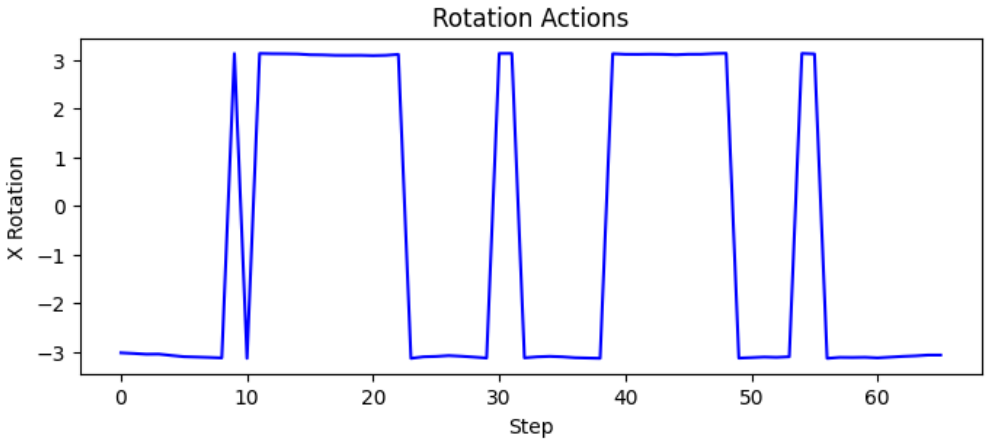


Figure 6.5: Actions of the rotation x-axis for all steps in an arbitrary episode.

6.1.3 Other Domain Gaps

In addition to the poor data quality, there may be more reasons that contribute to models not generalizing to our environment. We use the Franka Panda robot and in the original release of the X-embodiment data, the share of trajectories using this robot is relatively small, as shown in figure 6.6 [9].

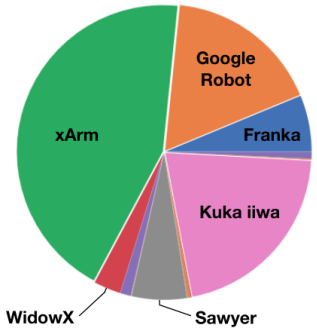


Figure 6.6: Trajectories per embodiment in the Open X-Embodiment data [9].

The majority of the X-embodiment data is recorded from real-world setups and not in simulators. Although Unity is able to render realistic graphics, there is a visual gap that possibly prevents the generalization to our simulated environment.

6.2 Fine-Tuning

In our experiments, we find that fine-tuning generally works. Therefore, we show that our simulation environment can be used both for the collection of demonstrations and the evaluation of generalist robot policies. We extensively analyzed the Octo model for in-domain and out-of-domain generalization. Here, we find that we can achieve a high success rate for a single task such as "Pick up the cube". However, the performance can vary widely for different objects and skills. Especially spherical objects that can roll away are hard to grasp by the model because a high accuracy of positioning the gripper is required. For the same reason, the success rate also drops for objects that are smaller or have less tolerance for grasping mistakes, such as the knife with a small handle.

For the out-of-domain generalization, we find that there is often no substantial difference between seen and unseen tasks. In the case of the experiment with the unseen interaction position (section 5.4.1) we placed the unseen position between the other positions. The same is true for the unseen camera pose in section 5.4.2. Perhaps this methodology is too easy and other experiments should be conducted where the interaction position is further away or the camera pose is on the opposite site. In the experiments where we leave out combinations of two dimensions, the difficulty may be increased by incorporating more dimensions, e.g., test an unseen task "Pick up the cube" with an unseen camera angle where the cube was not seen in the *pick up* skill and both the cube and the *pick up* skill were not seen from the camera angle.

Across almost all experiments, we notice a high standard deviation. Qualitatively, we observe that in some evaluation runs there may be five unsuccessful episodes followed by, e.g., four successful episodes. Therefore, we believe that 10 rollouts as in some evaluations in STAR-Gen [15] or other models presented in chapter 4 are insufficient for accurate results. While real-world evaluations are time consuming, we can utilize our simulation platform to scale the number of evaluation rollouts with minimal effort.

6.3 Benchmark

We use the results obtained from the experiments in sections 5.4.3, 5.4.4, and 5.4.5 to calculate scores for the Octo model in our benchmark. The benchmark for Octo is shown in figure 6.7. We report scores for the three dimensions *skill*, *object*, and *camera*. In section 4.1.4 we describe the methodology for calculating these scores. The score denoted as

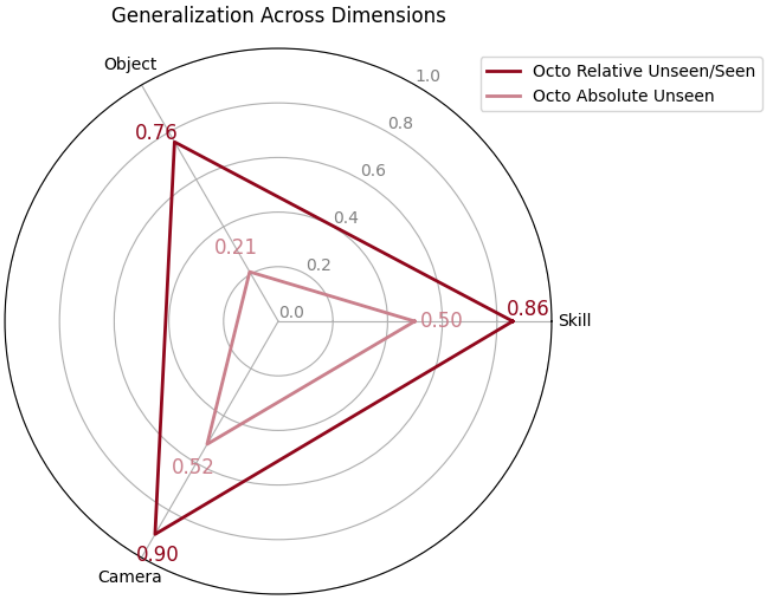


Figure 6.7: Benchmark scores for the Octo model.

"Absolute Unseen" is the average success rate for unseen tasks. In order to calculate this score, we first collect the success rates for unseen tasks from all experiments that included the respective dimension. For example, for the *object* dimension we collect the success rate 0.19 from the *skill/object* experiment (section 5.4.3) and the success rate 0.23 from the *cam/object* experiment (section 5.4.4). For simplicity, we then take the mean of these values to end up with a single score per dimension, e.g., 0.21 for the *object* dimension. This score may be used to compare different models with each other in terms of their performance for unseen tasks.

We are also interested in a model's ability to combine and transfer knowledge from seen tasks to a new unseen task. Therefore, we put the success rate of an unseen task in relation to the baseline success rate of the same seen task. This is useful because it highlights the knowledge transfer and eliminates other factors. For example, handling some objects may be harder than handling other objects, leading to an overall low success rate in the object dimension, as shown in figure 6.7. However, this does not necessarily mean that the model can not transfer knowledge well across the object dimension. In figure 5.26 we see that for most objects, the unseen success rate is similar to the baseline where the objects were seen. We can therefore conclude that the model actually learns to recognize

objects and identifies them correctly even if they were not seen from the particular camera perspective. For this reason, we introduce the score denoted as "Relative Unseen/Seen" which represents the percentage ratio of the success rate for unseen tasks to the success rate for the same seen tasks. For each experiment we calculate this ratio and attribute it to the dimensions involved. The final score for a dimension is calculated by taking the mean across the attributed values. For example, in the *skill/object* experiment (section 5.4.3) the success rate for unseen combinations reaches 72% of the seen baseline. In the *cam/object* experiment (section 5.4.4), the success rate for unseen combinations reaches 80%. Therefore, we report the average of 76% as the score for the *object* dimension in figure 6.7.

Overall, the benchmark in figure 6.7 shows that Octo achieves a high *relative* score across all dimensions. The *absolute* score, on the other hand, is low especially with the *object* dimension only achieving a score of 0.21. The reason why the score of the other two dimensions is higher at approximately 0.5 can be traced back to the *cam/skill* experiment (section 5.4.5) that yields a high success rate and only affects these two dimensions. The high success rate is partly due to the fact that the only manipulation object in this experiment is the cube that is easier to grasp. To compensate for such effects, it may be required to conduct the *cam/skill* experiment with multiple objects (but all objects are seen). Similarly, the *skill/object* experiment may be conducted with multiple camera poses that are all seen. To conclude, we can derive two major takeaways from the benchmark. First, when we want to fine-tune Octo to a new setting with generalization across multiple dimensions, we do not have to train on every task combination. Instead, it is enough to include objects, skills, etc. in a subset of combinations because Octo is good at transferring knowledge. However, the second insight is that Octo's overall success rate is low when a single model is used for diverse tasks. Especially multiple objects cause Octo's performance to drop because grasping some of them requires greater accuracy.

7 Outlook

In this work, we have introduced our simulation platform RoboXim that is specifically designed to evaluate generalist robot policies. We have demonstrated that this platform can be used for the generation of demonstrations as well as for the evaluation of models. However, the platform can be considered to be work in progress as some aspects can be improved. One issues is that the trajectory planning service implemented with Moveit [41] is not perfect. Sometimes, the trajectory planning fails and the episode has to be discarded, costing time and compute resources. In the future, we would also like to incorporate sophisticated collision avoidance in the trajectory planning to support more complex tasks, e.g., tasks where a path is blocked by distractor objects. In general, we would like to see more skills and robots implemented in RoboXim. We kept this in mind when designing the architecture, and therefore adding new skills and robots should be a straightforward process. Another point of interest is parallelization in our simulation. Currently, we support containerization of our application, making it viable to run multiple instances at once. Other platforms such as Maniskill [44] and IsaacLab [27] simulate multiple environments in a single instance of the application. This can be implemented in Unity as well, but it should be investigated whether this brings any benefits in terms of simulation speed or VRAM usage over just running multiple containers.

In chapter 5 we conducted a wide range of experiments for an in-depth analysis of generalist robot policies. We focused on the Octo model [28] and found that fine-tuning works as intended by the original authors, but performance generally drops when the range of skills and objects to generalize over increases. The next step would be to evaluate other models with the same experiments to compare them with Octo and each other. Some of the experiments may also be repeated with higher complexity, e.g., more colors in the *multiple colors* experiment (section 5.2.2), or more objects in the *multiple objects* experiment (section 5.2.3).

The benchmark described in section 6.3 provided us with some key insights. For example, we found that Octo can effectively transfer knowledge across dimensions. Such findings

can, for example, help us to decide what data to include when we fine-tune a model because it follows that we do not need all task combinations when adapting the model to a new setting. However, this benchmark is mostly a demonstration of our method and can be scaled up in a future work. This includes evaluating the generalization across more dimensions, but also conducting more experiments per dimension to increase the reliability of the results. It would make sense to introduce some experiments with a higher difficulty. For example, instead of combining just two dimensions, we can combine three or four dimensions as mentioned in section 6.3. Finally, we would like to evaluate more generalist models with our benchmark to compare them with each other. We hope that RoboXim and our benchmark can help to uncover issues in generalist robot policies, e.g., a lack of generalization in a certain dimension, and help to improve these generalist models.

Bibliography

[1] *4k Scanned Vegetables MiniPack*. URL: https://assetstore.unity.com/packages/3d/props/food/free-4k-scanned-vegetables-minipack-135434.

[2] S. Belkhale and D. Sadigh. *Minivla: A better VLA with a smaller footprint*. https://github.com/Stanford-ILIAD/openvla-mini. 2024.

[3] *bridge_rlds_builder*. 2023. URL: https://github.com/kpertsch/bridge_rlds_builder.

[4] Greg Brockman et al. *OpenAI Gym*. 2016. arXiv: 1606.01540 [cs.LG]. URL: https://arxiv.org/abs/1606.01540.

[5] Anthony Brohan et al. "RT-1: Robotics Transformer for Real-World Control at Scale". In: *arXiv preprint arXiv:2212.06817*. 2022.

[6] Anthony Brohan et al. "RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control". In: *arXiv preprint arXiv:2307.15818*. 2023.

[7] Daniel Cer et al. *Universal Sentence Encoder*. 2018. arXiv: 1803.11175 [cs.CL]. URL: https://arxiv.org/abs/1803.11175.

[8] Xi Chen et al. *PaLI-X: On Scaling up a Multilingual Vision and Language Model*. 2023. arXiv: 2305.18565 [cs.CV]. URL: https://arxiv.org/abs/2305.18565.

[9] Open X-Embodiment Collaboration et al. *Open X-Embodiment: Robotic Learning Datasets and RT-X Models*. https://arxiv.org/abs/2310.08864. 2023.

[10] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2021. arXiv: 2010.11929 [cs.CV]. URL: https://arxiv.org/abs/2010.11929.

[11] Danny Driess et al. *PaLM-E: An Embodied Multimodal Language Model*. 2023. arXiv: 2303.03378 [cs.LG]. URL: https://arxiv.org/abs/2303.03378.

[12] Kiana Ehsani et al. *ManipulaTHOR: A Framework for Visual Object Manipulation*. 2021. arXiv: 2104.11213 [cs.CV]. URL: https://arxiv.org/abs/2104.11213.

[13] *Food and Kitchen Props Pack*. URL: https://assetstore.unity.com/packages/3d/props/food-and-kitchen-props-pack-85050.

[14] Chuang Gan et al. *ThreeDWorld: A Platform for Interactive Multi-Modal Physical Simulation*. 2021. arXiv: 2007.04954 [cs.CV]. URL: https://arxiv.org/abs/2007.04954.

[15] Jensen Gao et al. *A Taxonomy for Evaluating Generalist Robot Policies*. 2025. arXiv: 2503.01238 [cs.RO]. URL: https://arxiv.org/abs/2503.01238.

[16] Ahmed Hussein et al. "Imitation learning: a survey of learning methods." In: *ACM computing surveys* 50 (2 2017). ISSN: 0360-0300. DOI: 10.1145/3054912. URL: http://hdl.handle.net/10059/2298.

[17] *Isaac Sim Requirements*. https://docs.isaacsim.omniverse.nvidia.com/latest/installation/requirements.html. Accessed: 2025-04-24. 2025.

[18] Stephen James, Marc Freese, and Andrew J. Davison. *PyRep: Bringing V-REP to Deep Robot Learning*. 2019. arXiv: 1906.11176 [cs.RO]. URL: https://arxiv.org/abs/1906.11176.

[19] Stephen James et al. *RLBench: The Robot Learning Benchmark & Learning Environment*. 2019. arXiv: 1909.12271 [cs.RO]. URL: https://arxiv.org/abs/1909.12271.

[20] Eric Jang et al. *BC-Z: Zero-Shot Task Generalization with Robotic Imitation Learning*. 2022. arXiv: 2202.02005 [cs.RO]. URL: https://arxiv.org/abs/2202.02005.

[21] Arthur Juliani et al. *Unity: A General Platform for Intelligent Agents*. 2020. arXiv: 1809.02627 [cs.LG]. URL: https://arxiv.org/abs/1809.02627.

[22] Siddharth Karamcheti et al. *Prismatic VLMs: Investigating the Design Space of Visually-Conditioned Language Models*. 2024. arXiv: 2402.07865 [cs.CV]. URL: https://arxiv.org/abs/2402.07865.

[23] Moo Jin Kim et al. *OpenVLA: An Open-Source Vision-Language-Action Model*. 2024. arXiv: 2406.09246 [cs.RO]. URL: https://arxiv.org/abs/2406.09246.

[24] Xuanlin Li et al. "Evaluating Real-World Robot Manipulation Policies in Simulation". In: *arXiv preprint arXiv:2405.05941* (2024).

[25] Bo Liu et al. *LIBERO: Benchmarking Knowledge Transfer for Lifelong Robot Learning*. 2023. arXiv: 2306.03310 [cs.AI]. URL: https://arxiv.org/abs/2306.03310.

[26] Oier Mees, Jessica Borja-Diaz, and Wolfram Burgard. "Grounding Language with Visual Affordances over Unstructured Data". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. London, UK, 2023.

[27] Mayank Mittal et al. "Orbit: A Unified Simulation Framework for Interactive Robot Learning Environments". In: *IEEE Robotics and Automation Letters* 8.6 (2023), pp. 3740–3747. DOI: 10.1109/LRA.2023.3270034.

[28] Octo Model Team et al. "Octo: An Open-Source Generalist Robot Policy". In: *Proceedings of Robotics: Science and Systems*. Delft, Netherlands, 2024.

[29] Maxime Oquab et al. *DINOv2: Learning Robust Visual Features without Supervision*. 2024. arXiv: 2304.07193 [cs.CV]. URL: https://arxiv.org/abs/2304.07193.

[30] Dean A. Pomerleau. "ALVINN: An Autonomous Land Vehicle in a Neural Network". In: *Advances in Neural Information Processing Systems*. Ed. by D. Touretzky. Vol. 1. Morgan-Kaufmann, 1988.

[31] Wilbert Pumacay et al. "THE COLOSSEUM: A Benchmark for Evaluating Generalization for Robotic Manipulation". In: (2024).

[32] Colin Raffel et al. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer". In: *Journal of Machine Learning Research* 21.140 (2020), pp. 1–67. URL: http://jmlr.org/papers/v21/20-074.html.

[33] Sabela Ramos et al. *RLDS: an Ecosystem to Generate, Share and Use Datasets in Reinforcement Learning*. 2021. arXiv: 2111.02767 [cs.LG]. URL: https://arxiv.org/abs/2111.02767.

[34] A. Ren. *Open pi-zero*. https://github.com/allenzren/open-pi-zero. 2024.

[35] *Robotic Operating System*. URL: https://www.ros.org.

[36] Eric Rohmer, Surya P. N. Singh, and Marc Freese. "V-REP: A versatile and scalable robot simulation framework". In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2013, pp. 1321–1326. DOI: 10.1109/IROS.2013.6696520.

[37] *ROS-TCP-Connector*. URL: https://github.com/Unity-Technologies/ROS-TCP-Connector.

[38] *ROS-TCP-Endpoint*. URL: https://github.com/Unity-Technologies/ROS-TCP-Endpoint.

[39] Erick Rosete-Beas et al. "Latent Plans for Task Agnostic Offline Reinforcement Learning". In: 2022.

[40] Michael Ryoo et al. "TokenLearner: Adaptive Space-Time Tokenization for Videos". In: *Advances in Neural Information Processing Systems*. Ed. by M. Ranzato et al. Vol. 34. Curran Associates, Inc., 2021, pp. 12786–12797.

[41] Ioan A. Sucan and Sachin Chitta. *MoveIt*. URL: https://moveit.ros.org/.

[42] Ioan A. Sucan, Mark Moll, and Lydia E. Kavraki. "The Open Motion Planning Library". In: *IEEE Robotics & Automation Magazine* 19.4 (2012), pp. 72–82. DOI: 10.1109/MRA.2012.2205651.

[43] Mingxing Tan and Quoc Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, Sept. 2019, pp. 6105–6114. URL: https://proceedings.mlr.press/v97/tan19a.html.

[44] Stone Tao et al. *ManiSkill3: GPU Parallelized Robotics Simulation and Rendering for Generalizable Embodied AI*. 2024. arXiv: 2410.00425 [cs.RO]. URL: https://arxiv.org/abs/2410.00425.

[45] SIMA Team et al. *Scaling Instructable Agents Across Many Simulated Worlds*. 2024. arXiv: 2404.10179 [cs.RO]. URL: https://arxiv.org/abs/2404.10179.

[46] Emanuel Todorov, Tom Erez, and Yuval Tassa. "MuJoCo: A physics engine for model-based control". In: Oct. 2012, pp. 5026–5033. ISBN: 978-1-4673-1737-5. DOI: 10.1109/IROS.2012.6386109.

[47] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. arXiv: 2307.09288 [cs.CL]. URL: https://arxiv.org/abs/2307.09288.

[48] *Unity*. URL: https://unity.com/.

[49] Yusuke Urakami et al. *DoorGym: A Scalable Door Opening Environment And Baseline Agent*. 2022. arXiv: 1908.01887 [cs.RO]. URL: https://arxiv.org/abs/1908.01887.

[50] *URDF-Importer*. URL: https://github.com/Unity-Technologies/URDF-Importer.

[51] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[52] Homer Walke et al. *BridgeData V2: A Dataset for Robot Learning at Scale*. 2024. arXiv: 2308.12952 [cs.RO]. URL: https://arxiv.org/abs/2308.12952.

[53] Tom Ward et al. *Using Unity to Help Solve Intelligence*. 2020. arXiv: 2011.09294 [cs.AI]. URL: https://arxiv.org/abs/2011.09294.

[54] *Western Weapons | Free Knife*. URL: https://assetstore.unity.com/packages/3d/props/weapons/western-weapons-free-knife-290804.

[55] Fanbo Xiang et al. "SAPIEN: A SimulAted Part-Based Interactive ENvironment". In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2020), pp. 11094–11104. URL: https://api.semanticscholar.org/CorpusID:213175506.

[56] Tianhe Yu et al. *Meta-World: A Benchmark and Evaluation for Multi-Task and Meta Reinforcement Learning*. 2021. arXiv: 1910.10897 [cs.LG]. URL: https://arxiv.org/abs/1910.10897.

[57] Xiaohua Zhai et al. *Sigmoid Loss for Language Image Pre-Training*. 2023. arXiv: 2303.15343 [cs.CV]. URL: https://arxiv.org/abs/2303.15343.

[58] Tianhao Zhang et al. *Deep Imitation Learning for Complex Manipulation Tasks from Virtual Reality Teleoperation*. 2018. arXiv: 1710.04615 [cs.LG]. URL: https://arxiv.org/abs/1710.04615.

[59] Yuke Zhu et al. "robosuite: A Modular Simulation Framework and Benchmark for Robot Learning". In: *arXiv preprint arXiv:2009.12293*. 2020.