
Low Level Robot Control Using A Multi Modal Foundation Model

P10 project

Master's Thesis

Group 1052

Mahed Dadgostar & Victor Bjørholm

Aalborg University
Electronics and IT



AALBORG UNIVERSITY

STUDENT REPORT

Material Processing
Aalborg University
<http://www.aau.dk>

Title:

Low Level Robot Control Using A Multi Modal Foundation Model

Theme:

Multi Modal, Foundation Model, AI

Project Period:

Spring semester 2024

Project Group:

1052

Participant(s):

Mahed Dadgostar
Victor Hagbard Bjørholm

Supervisor(s):

Dimitris Chrysostomou
Chen Li

Copies: 1

Page Numbers: 69

Date of Completion:

May 31, 2024

Abstract:

Robotic systems are often highly specialized, with little flexibility for different tasks. In this report, we outline our work on implementing our own robotic control stack in our pursuit to experiment on Octo, a multi-modal foundation model, for low-level control of a robotic manipulator. Octo is designed for flexibility, capable of running on various robotic hardware and performing a wide range of tasks. We fine-tuned Octo on our own data, recorded using tools developed for this project. This data is in a standardized format for future use in training robotic systems. To train and run Octo, we created a custom robot environment, integrated it with a Polymetis server wrapped in a ZeroRPC server, developed a VR control system for intuitive robot control, and built our own data recording tools. We modified existing Octo scripts to fit our use case, successfully fine-tuning and running Octo in our custom environment. Our model was trained to use two camera inputs and a task description to pick up an arbitrary object

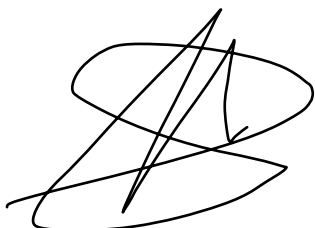
Contents

Preface	iv
1 Introduction	1
2 Literature review	2
2.1 Foundational Models in Robotics	2
2.1.1 Foundational Models Related to Robotics	2
2.1.2 Robotics Foundation Models	7
2.2 Problem statement	13
3 Methodology	14
3.1 Methods overview	14
3.2 Octo	16
3.2.1 Modular Design of Octo	16
3.2.2 Additional Octo components	17
3.2.3 Octo Availability	19
3.3 Reinforcement Learning Dataset	19
3.4 Polymetis	20
3.5 ZeroRPC	21
4 Implementation	23
4.1 Implementation overview	23
4.2 Data capture	25
4.2.1 Franka Environment	25
4.2.2 Camera Environment	33
4.2.3 Gym Environment	34
4.2.4 Envlogger	35
4.2.5 Data Recording	37
4.2.6 Dataset Loader and Modifier	38
4.2.7 Virtual Reality	39

4.3	Octo	43
4.3.1	Finetuning	43
4.3.2	Inference	46
4.4	Data collection	50
4.4.1	Proof of Concept using Dummy Data	50
4.4.2	Recording first dataset	51
5	Testing	56
5.1	Testing configuration	56
5.2	Tests of different models	57
5.2.1	H1 test	57
5.2.2	H2 test	58
5.2.3	H2X test	58
5.2.4	H3 test	58
5.2.5	H3X test	59
5.2.6	H4 test	59
5.2.7	H5 test	60
6	Discussion	61
6.1	Key thoughts and findings	62
7	Conclusion	65
	Bibliography	66

Preface

Aalborg University, May 31, 2024



Mahed Dadgostar
<mdadgo22@student.aau.dk>



Victor Hagbard Bjørholm
<vbjarh19@student.aau.dk>

Chapter 1

Introduction

In the recent years, AI systems have become integral to everyday life, with applications ranging from machine learning (ML) algorithms driving cars to using ChatGPT for creating recipes in the kitchen. The latter has truly exploded in usage, and can be used to automate many general tasks on a computer. However, the integration of AI in robotics presents a different scenario. While many Robotic systems employ ML algorithms to achieve their goal, these systems are often highly tailored to a specific robotic model and a specific task. Even if the systems are made for solving various tasks, they are often tailored to a specific robotic system and environment. Adding a new type of input, is often not possible after training. Even with projects, trying to solve general task solving in robotics, there is a lack of available training data for robotics in a standardized format. This is in stark contrast to LLM systems, which can train on essentially all available text.

In this project we have worked on implementing and training Octo; a Multi Modal Foundation Model, designed to be easily transferable between different physical environments, robots and with various tasks. To be able to use Octo, one must fine-tune the model first for it to be able to use a specific physical environment and specific observation and action spaces. To fine-tune the model, we have implemented tools to record high quality training data in a standardized format - which will be shared in an open source library for robot data. We focused on the low-level control of a Franka Emika robot, using several fine-tuned Octo models trained on over 100 episodes of our data. The subsequent chapters provide a comprehensive review of current state-of-the-art methods, detailed descriptions of our implementation, and thorough testing of the Octo system.

Chapter 2

Literature review

Following the success of foundational language models, which caused a paradigm shift in natural language processing, the same experience with a similar approach has also started in other fields, such as computer vision and, more recently, robotics. The approach underlines gathering vast and diverse data and training massive networks on the given dataset. Transformer-based models are the most dominant architecture of these networks. This chapter presents an overview of recent efforts for the foundational models paradigm in robotics and investigates the currently available solutions for the specified tasks.

2.1 Foundational Models in Robotics

Foundational Models(FM) are defined as models that are generally trained on large-scale data (e.g., internet web text/image) with the capability of adaptability to various tasks[1].

Figure2.1 illustrates some of the more impactful FM robotics projects.

2.1.1 Foundational Models Related to Robotics

Because FMs are trained on vast amounts of data, their training dataset distribution contains useful information that can be used for robotics in many ways, for example, task planning and context and semantic information processing of the object scene[2]. They can be used for action generation in a zero-shot manner. Most works use FMs for task planning, as task planning requires knowledge of other domains that FMs have. Different types of FMs can be incorporated into robotics, such as Large Language Models, Vision Foundation Models, and Multi-

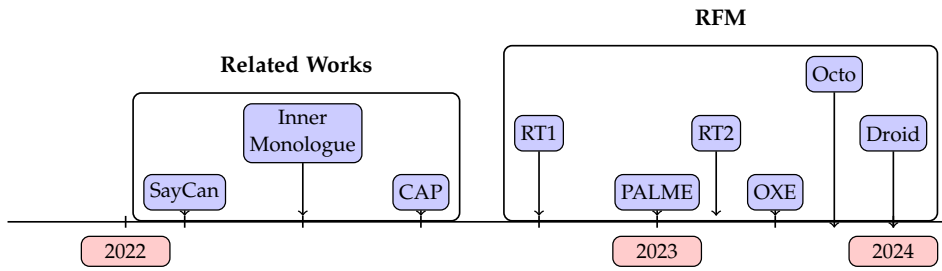


Figure 2.1: Timeline of important papers pertaining to Robotics Foundational Models and related works to foundational models paradigm.

modal Foundation Models. We highlight some important papers that leverage FMs for robotics.

Large Language Models

Natural Language Processing(NLP) has been among the fields most benefited from FMs. The reason for that is the amount of text data available to create data-driven solutions for the NLP problems. Transformer-based architectures[3] have the most success in the field because they can exploit the parallelism in computation and be trained at scale together with the capability of self-attention mechanism to capture relation and context between different parts of an input data sequence[1]. As LLMs understand rule-based and structural text-rich contexts[4], we discuss a few selected projects in which LLMs helped as a core part of the solution of the robotics tasks.

SayCan[5] was one the first leading papers to bring LLMs to robotics. It works by grounding the high-level language instruction input for example, "Can you bring something to drink on the table?", to possible lower-level sub-tasks ("1. "find a water" or "find a soda can" or etc.) with a score showing that how much each sub-task is the correct next sub-task for the next step. Then, for each possible sub-task at hand, a trained value function or affordance function is used to measure how likely the sub-task is to perform successfully. The skill to use is selected based on a score, a combination of the probability of each possible valid skill from LLM and the probability of that action being performed successfully. *SayCan* uses a pre-trained set of skills and doesn't necessarily create a new skill unless it was developed in its training dataset.

Inner Monologue[6], in contrast to *Saycan*, performs better by essentially providing environment feedback to the LLM planner. This feedback should be in text format so the LLM can interpret it. The feedback allows for replanning in case of failed executions, improving the model's performance. Multiple sources of feed-

back were adopted by Inner Monologue[6] to add to the LLM prompt, *Success Detection* is whether the current task is completed successfully, *Passive Scene Description* is the semantic scene information that is done before each planning step, and *Active Scene Description* provides on-demand unstructured semantic feedback. Note that "unstructured" here means this feedback can also include human preferences as well as the output from Vision-Question-Answering(VQA) models[6]. It is also worth mentioning that Inner Monologue employs ViLD[7] and MDETR[8] to get scene descriptions, which are pre-trained open-vocabulary object detection models. These are counted as Vision Language Models. However, since they are not used as the core part of the solution of this paper, we put them inside the LLM category.

Code As Policies(CaP)[9] incorporates LLMs differently; it investigates the code generation ability of LLMs as a robot policy generator for a mobile manipulator. *CaP* formulates the task planning problem for high-level language instruction as code generated by an LLM that can be directly executed on the robot's workstation. The authors achieve this by building language model-generated programs(LMPs). LMPs are generally LLM-generated programs that can be executed on a computer[9]. To generate an LMP, an LLM is prompted with a few-shot examples of demo output in the form of Python comments; this is done to give the LLM the proper context to get the desired behavior. Additionally, the context can also be given directly. LLMs, mainly those trained on large amounts of codes, can use third-party libraries, which expands their abilities to develop better and more creative codes. The results of *CaP* state that, by leveraging LLMs' logical and spatial reasoning capabilities, they could generalize and perform unseen tasks not given in the few-shot prompt. For example, by specifying a pre-trained skill going from place A to B function in the prompt, the LLM can draw a *square* without the need to add a new skill of drawing a *square*[9]. Moreover, they can ground qualitative ambiguous language instructions like "slower" to reasonable quantitative value in the output code[9].

We also mention some of the significant projects that were branched off from the previously mentioned papers.

KNOWNO, also known as *Robot-Help*[10], tries to solve the uncertainty in ambiguous situations using a statistical method. Ambiguous situations can happen when the robot ends in an uncertain situation with respect to the scene environment. As an example, when the user prompts "pick up the can," and in the scene description, the robot receives "Red-Bull can" and "Coca-Cola can," based on the lack of context from the user, the correct object might not get picked up. In such cases, the robot should ask for clarification and intervention from the human user to resume executing tasks. There is a trade-off here: the robot should have minimal frequency

of human interventions as it should handle the input instructions autonomously and successfully. *KNOWNO* operates by first few-shot prompting an LLM with the added context of the task instruction and scene description; then the LLM generates some possible unambiguous multi-options to choose from for the new task instructions; following that, the LLM scores these options by their level of likeliness (certainty) given the current situation. These options are then compared against a calibrated value obtained through Conformal Prediction (CP); if only one option passes this threshold, this means there is no uncertainty here and that the task will be executed; otherwise, the passed set of possible options is presented for the user to choose from. CP guarantees that the robot asks for intervention correctly based on a given success rate. Note that the calibration for CP needs a dataset that can be collected manually or by using heuristic methods and LLMs[10].

DROC [11] proposes an In-Context Learning (ICL) method as a task planner for a manipulator. *DROC* uses a pre-trained LLM task planner that breaks a complex high-level task instruction into simpler sub-tasks that are translatable to the robot's pre-trained skill set. While the robot is operating, *DROC* allows human interventions to make corrections to the plan that the robot is following. Moreover, to decrease the number of human corrections to complete a task, *DROC* consists of three blocks: Correction Handler, Knowledge Extractor, and Knowledge Retriever. Correction Handler deals with how to handle the correction received from the human; this correction can be a high-level task planning matter(semantic and task constraint, e.g., if the robot has one arm and has already grasped an object, it cannot put the object inside a closed drawer before first opening the drawer then open then pick up the object) or low-level(changing a skill primitive parameter like grasp pose). It decides whether to re-plan the task or adjust the skill primitive. Knowledge Extractor filters out the important information to keep as context for future interactions. Knowledge Retriever prompts the LLMs to decide to use past experiences that are relevant to the task. With these blocks, *DROC* uses the history of interaction and corrections over time, resulting in a better performance than baselines like *CaP* [9].

Up to now, the projects discussed in this section do not involve LLMs in the action and trajectory generation part of the robot control; they might change some parameters of the skill function that generates the low-level control. *Language Models as Zero-shot Trajectory Generators*[12] deploys the LLM more extensively for trajectory generation. This work achieves end-effector trajectory generation without providing any in-context examples in the prompt, nor using any external trajectory optimizer, mainly relying on the LLM's physical understanding of the environment, in this case GPT-4[13]. This study proves that by giving the appropriate prompt, the LLM can do 26 everyday manipulation tasks [12]. Note that for the perception part of the pipeline, the LLM uses LangSAM[14] off-the-shelf, a segmentation and

object detection open-vocabulary model.

Vision Language Models

Combining different modalities rather than just one (language) helps with understanding the physical environment[2]. Vision-Language-Models(VLMs) are the most used multimodal models in robotics[2]. In the previous section 2.1.1, in the works we looked into, VLMs were incorporated for language-grounded perception for object detection and segmentation[6, 9, 10, 11, 12]. A natural next step would be to use more recently developed and advanced VLMs like *GPT-4V(ision)*[15] as task planners and have them throughout the pipeline in embodied robotics applications.

"Look Before You Leap: Unveiling the Power of GPT-4V in Robotic Vision-Language Planning"[16] introduces *VILA(Vision-LAnguage Planning)*, a platform which uses GPT4-V for long-horizon planning in robotics. The paper argues that even though LLM planners can utilize external affordance models like open-vocabulary object detectors [9] and value functions[5] to give a grounding in the real world for them, using language as the only modality to convey the information for the task planner faces challenges when it comes to a scene with complex relations between the objects[16]. For example, in a scene with cluttered objects on a table or a scene with the desired object inside a drawer, affordance models cannot see that object to relay the information to the LLM planner, which causes the LLM not to be able to reliably come up with the right plan to solve the task. LLM-based methods often lack the fine details needed for these kinds of tasks. Having an inherent vision integration to describe this scene can make it much easier than translating all the required context into text. Commonsense knowledge about semantic information of objects, which sometimes is task-specific and spatial reasoning, are some of the main reasons that *VILA* has better results over the tested LLM-based baselines, namely SayCan[5] and Grounded-Decoding[17]. *VILA* has a closed-loop architecture that allows for replanning based on visual feedback; the VLM also acts as a success detector for the executing task, tracking whether each plan has been performed successfully. Additionally, it works based on the assumption that the robot has a pre-trained set of skill primitives. As an extra feature, *VILA* supports image goal-conditioned tasks; this adds to its versatility in handling various situations.

"MOKA: Open-Vocabulary Robotic Manipulation through Mark-Based Visual Prompting"[18] also powered by GPT-4V[15], incorporates the VLM differently, it goes furthermore to use the pre-trained VLM to the point that it can be used for low-level motion generation. It relies on the VLM itself to generate affordance representations to achieve this. This paper breaks down task planning and motion generation

problems for manipulation applications into high-level and low-level modules that both query the VLM. In the high-level module, the VLM is prompted to divide the arbitrary language instruction from the user(e.g., *"wipe"* into easier-to-handle sub-tasks, manifested as a list of dictionaries[18]. Each dictionary describes the details of a subtask, including a language description of the subtask(e.g., *"Wipe the trash to the upper side of the table using the broom"*), together with detailed information about the object to be grasped(e.g., *"the broom"*), the target object to be manipulated(e.g., *"the trash"*), the motion description(e.g., *"from down to up"*), and the objects to be ignored(e.g., *"snack package, etc."*). This gives the low-level reasoning side adequate details for generating affordance representation. For the low-level reasoning, *MOKA* leverages GroundedSAM[19] for keypoints generation for the objects highlighted in the high-level reasoning dictionary. Then, it divides the scene RGB image taken by the camera at the start of the subtask into a chessboard-coordinated grid. Finally, the VLM is prompted to choose the waypoints to reach the goal of the subtask. It is worth noting that providing the VLM with two to three in-context examples for both high-level and low-level reasoning enhances the performance in the experiments. *MOKA* doesn't need predefined skill primitives as opposed to *VILA*[16] as it can even be used as a framework to learn new low-level policies through its policy distillation feature.

VLMs are not the only multimodal used in robotics; in fact, with more promising results coming from transformer-based models[2, 1], other modalities like proprioception can be used as the input for dedicated robotics model[2]. To step even further, even the control command for the robot and trajectory generation can be produced internally by the model without needing any external trajectory optimizer or pre-trained skill primitives.

2.1.2 Robotics Foundation Models

So far, we have mentioned some of the applications of foundational models that weren't directly designed for robotics purposes but still benefited from them. A Robotics Foundation Model(RFM) should be able to support multi-modality for input and solve robotics tasks; it can be manifested as low-level action control of the robot(Action Generation RFMs) or higher-level motion planning[2]. In addition, there hasn't been a web-scale diverse dataset specialized for robotics developed yet compared to the ones for known VLMs and LLMs; however, we investigate the major projects toward this.

"RT-1: Robotics Transformer For Real-World Control At Scale"[20] was one of the first RFMs that could take images of the scene and the language instruction from the user to output low-level action command for the robot. *RT-1* has a transformer-based model at its core; the *RT-1*[20] model architecture, as one of its primary con-

tributions, consists of FiLM[21] conditioned EfficientNet[22], a TokenLearner[23], and a Transformer[3].

The language instruction and a set of images from the scene are the inputs to the model, with the images being the most recent sequence of images, which is chosen to be the length of 6[20]. For tokenization of these different modalities for the input, the 300×300 input images are passed through ImageNet pre-trained EfficientNet-B3[22] for feature extraction that results in $9 \times 9 \times 512$ feature map. Next, this feature map is flattened to create 81 visual tokens for the next layers of the model. To count in the effect of language instruction, it is embedded at first using the Universal Sentence Encoder[24]. Then, this embedding is the input for identity-initialed FiLM[21] layers that are interweaved inside the EfficientNet to condition it on the language task instruction input[20]. After this, TokenLearner[23] compresses the 81 visual tokens to 8 final tokens for transformer layers, resulting in a 2.4x faster inference[20]. Note that the final tokens are in place for every image in the history; therefore, for the history of 6 images, there are 48 total tokens. In the Transformer module, a decoder-only sequence model with 8 self-attention layers is adopted[20]. Finally, the Transformers model gives the action tokens a vector quantized to 256 bins for each action dimension. As *RT-1* was mainly designed to work with a mobile manipulator, this vector includes information about the arm end-effector state($x, y, z, roll, pitch, yaw, gripper\ state$), the mobile robot's base(x, y, yaw), and a discrete variable to indicate the robot's mode(controlling arm, base, episode termination)[20]. The total number of parameters for *RT-1* is reported to be 35M[20]. During inference, it can generate action tokens at a rate of 3 Hz, which is suitable to be deployed for real-time applications[20]. Additionally, there is a time limit for executing each task, and at the occurrence of surpassing it, the robot's current task is terminated.

RT-1 mentions that the success rate of completing long-horizon tasks decreases with the length of the task. Therefore, combining it with methods that focus on task planning and high-level reasoning like *SayCan* produces a better outcome in experiments with longer horizons(as long as 50 steps)[20]. In such cases, *RT-1* is the low-level policy of these pipelines[25, 26].

Training *RT-1* involved 17 months of data collection, during which human-demonstrated datasets were gathered across various kitchen tasks using 13 Everyday Robot[27] mobile manipulators, making in a total of 130,000 episodes[20].

Adding simulation data for the training dataset helped the model work better only in simulated environments, not real-world experiments[20]. Also, mixing the training dataset with the data QT-Opt[28], which has different robot morphology(KUKA), helps the success rate of Bin-Picking evaluation[20]. It indicates suc-

successful transfer between different setups. The authors of *RT-1* mention that using imitation learning as the method in this paper limits task performance to that of the demonstrators. Additionally, they note that generalization to unseen instructions is restricted to tasks already observed within the dataset.

"PaLM-E: An Embodied Multimodal Language Model"[25] is a general-purpose multimodal RFM that can handle long-horizon complex tasks. However, it cannot directly generate action commands and requires a low-level action generator like *RT-1*[20] or another pre-trained skill set.

PaLM-E encodes language instructions, state estimates from the robot, and scene images into a latent vector embedding. As suggested by its name, *PaLM-E* is structured such that the output vision tokens from the 22B Vision-Transformer(ViT)[29] are integrated into 540B PALM[30], creating a model with 562B. To our knowledge, this is still the largest RFM. Note that *PaLM-E* also comes with smaller scale models, 84B, and 12B, with better generalization to more tasks as the number of parameters increases[25].

PaLM-E's LLM has a decoder-only architecture that only generates text output autoregressively, meaning it can be conditioned on the multimodal input prompt. The prompt is in VQA format, as an example, *"Given . Q: If a robot wanted to be useful here, what steps should it take? A:"* where *""* is the image embedding. The output can be either: 1) a normal text answer to a question, or 2) a plan for a robot, breaking the main goal into subgoals executable by pre-trained low-level policies[25]. Each subgoal is a short-horizon task that does not independently solve the main goal[25]. Although *PaLM-E* cannot generate low-level policies, it differs from the previously mentioned VLM task planners in section 2.1.1 in the sense that high-level task generation is done naturally inside the model, which is trained to do this, this means that *PaLM-E* should search for available policies and plan the right ones for execution. In one of the experiment setups, the low-level policies developed in *RT-1*[20] are leveraged[25]. *PaLM-E* has a closed-loop structure, enabling it to replan and adapt to the disturbances that are happening in the environment.

For training *PaLM-E*, it used pre-trained weights of the ViT and PaLM, followed by finetuning(co-training) on data mixtures of various vision-language tasks, with only about 10 percent of robotics-related tasks[25]. *PaLM-E* reports various emergent capabilities, from zero-shot multimodal chain-of-thought reasoning to multi-image reasoning, even though it was trained on single-image input. Moreover, experiments show that this method successfully generalizes to unseen tasks and can significantly transfer its abilities to new robots[25].

"RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control"[31] is the next generation of the RT series models. RT-2 continues the RT-1 approach of generating direct action values for running on the robot. This paper introduces a new category of models called Vision-Language-Action (VLA) models. VLAs go along with the capabilities of VLMs to leverage vision and language inputs; additionally, they can generate low-level action outputs for running directly on the robot[31]. To achieve this, RT-2 takes an already existing VLM and co-finetunes it on the mixture of robot trajectories and internet-scale vision-language data. No new data were collected for RT-2; it utilizes the RT-1 dataset for robotic trajectories and the vision-language dataset that includes both VQA tasks and image-text pairs from different sources[31]. In addition, since a pre-existing VLM is used for finetuning, there are no additional parameters to be trained from scratch[31].

RT-2 defines the action output vector in the delta-pose action format, indicating how much the robot should move from its current state. The delta-pose action format is represented in Cartesian coordinates and the Euler orientation of the end-effector. A terminate flag and a value for the gripper extension are also concatenated to this to complete the action vector, yielding to this vector[31]:

```
"terminate,  $\Delta pos_x$ ,  $\Delta pos_y$ ,  $\Delta pos_z$ ,  $\Delta rot_x$ ,  $\Delta rot_y$ ,  $\Delta rot_z$ , gripper_extension"
```

Similar to RT-1, each continuous dimension of the action vector in RT-2 is discretized into 256 uniformly distributed bins. However, as RT-1 used its own transformer block to generate action vectors, RT-2 is built on top of a VLM, requiring a different approach. The output type of the VLMs used in RT-2's experiments is text, which cannot be used directly to pass to the robot. For example, a final output can be "1 128 91 241 5 101 127"; Tokenization of integer numbers varies between VLMs. In some VLM like the PaLM-E[25] 12B models used in RT-2[31], there is not a straightforward way to associate the 256 integers with their equivalent tokens in the model. In this case, the 256 least frequent tokens are selected to be mapped to the physically grounded values for the action vector[31]. This process is called detokenization in the RT-2 report[31].

The robotic data is arranged in the Visual Question Answering (VQA) format to co-finetune the VLM. Multimodal inputs are the scene's camera image(s) and the textual task instruction. The output is the action vector representation of the equivalent/least frequent tokens explained above (e.g., "Q: Given what should the robot do to <task_instruction> : A:")[31]. This robot data is then combined with the customized web data, with the robot data comprising between 50 to 65 percent of the mixture, resulting in improvement in generalization in the experiments done in RT-2 [31].

RT-2, while capable of generating low-level actions, can also function as a generalist robot, performing chain-of-thought reasoning[31]. In essence, *RT-2* combines the low-level action generation capabilities of *RT-1* with the reasoning and higher-level planning functionalities of *PaLM-E*[25]. Tokens in the action vector should be valid to be executed on a robot. Therefore, *RT-2* filters out its VLM decoder’s output to include only valid action tokens[31] when requested to generate action tasks.

Although the investigated RFMs demonstrate significant results compared to their baselines, they are not yet on par with the performance of state-of-the-art LLMs and VLMs for their respective tasks. To replicate the success of VLMs and LLMs in robotics, a large community of academic and industrial labs came to put together and share their robotics data to create one very big dataset to train better RFMs; this project is “*Open X-Embodiment: Robotic Learning Datasets and RT-X Models*”[32]. This project investigates whether a robot policy trained across different embodiments works better than the one trained just with that robot and setup. This isn’t limited to different robots; it may also be different setups and tasks[32].

Open X-Embodiment has two main contributions:(1) The newest version of *RT* series, *RT-X* models, which are the updated version of *RT-1*[31] and *RT-2*[31] trained with the newly gathered larger dataset, and (2) The release of *Open X-Embodiment(OXE)* repository, which consists of pure robotic dataset with different embodiments together with the pre-trained model checkpoints for *RT-1(-X)* that is trained with *OXE*[32]. *OXE* is an ongoing effort, and new datasets can still be submitted. *OXE* dataset consists of 1M+ robot trajectory with at least 22 various embodiments, with the Franka[33] robot having the most presence[32]. This dataset is in the RLDS[34] format.

RT-X introduces two updated versions of *RT-1* and *RT-2*, called *RT-1-X* and *RT-2-X*[32]. As the models’ architecture is unchanged, *RT-1-X* and *RT-2-X* were trained with the same input and output structures as their original models[32]. Experiments in *OXE* demonstrate that *RT-1-X* outperforms most models(50% on average) specifically designed for the tasks the dataset was collected for. This holds for the experiments dealing with small-scale dataset domains *RT-1*[32]. For *RT-2-X*, larger models work better in large-scale dataset domains, whereas *RT-1-X* seems to underfit.

Additionally, finetuning solely on the *OXE* dataset did not underperform co-finetuning with web data for *RT-2-X*, which was not the case with *RT-2*[32, 31]. The reason for that could be the *OXE* dataset, compared to the previous smaller dataset for *RT-2*, has much more diversity[32]. Furthermore, experiments for generalization to unseen tasks revealed that *RT-2-X* gets around 3x success rate compared to *RT-2*; this shows that training the robot policy on *X-embodiments* also improves the

generalization of the policy in unknown settings[32].

OXE is the biggest robotics community effort to provide a unified structure and diverse dataset to build the next generation of RFMs. Since the initial release of OXE, we have tracked two inspired papers contributing to RFM cause: (1) *Octo*[26], an open-source transformer-based robot policy that leverages the OXE dataset. (2) *Droid*[35], A newly multi-lab effort dataset together with the hardware code for data collection, which the dataset compared to OXE, has much more scene diversity as it was gathered outside of lab environments.

"*Octo: An Open-Source Generalist Robot Policy*"[26] is one of the recent leading developments of the RFM paradigm, which focuses on lower-level action generation for various robotics tasks. This paper, same as *RT-1*, enjoys a transformer-based for its architecture. Compared to the previous projects we investigated, it has a significantly more open-source-friendly approach for reproducing the training and fine-tuning steps. The *Octo* for its inputs, first tokenizes them; for the language input, it uses the *t5-base* model[36], and for image observations and image-based goals, it uses a shallow convolution stack then be sequentially flatten. Afterwards, with a learnable positional encoder is combines these tokens in sequential format.

One of the main strengths of *Octo* is its flexibility to various forms of inputs.

The action output of *Octo* is, by default, in a delta-pose format, which is also similar to *RT-1*. However, the output, which is the action head of this model, can be finetuned for both Cartesian and joint positions delta actions.

Octo for pre-training uses a selected mix of datasets in OXE, only using the datasets that have the delta-pose action and at least one image stream. All of the datasets were then analyzed and weighted based on diversity in tasks and environment. High diversity datasets had double the weight of low diversity datasets. Increasing the training dataset's heterogeneity. They are also ranked and weighted based on some other key metrics, like task relevance and camera resolution.

Moreover, it is one of the first RFMs that successfully employs proprioception as one of the input modalities. A new task for *Octo* to execute can be either in the format of a goal image or a language instruction. *Octo* comes with two models with different sizes, 27M and 93M,

As for our project, we have tried to be inspired by *Octo*[26] and *Droid*[35] for model architecture and data collection throughout the way.

2.2 Problem statement

After going through these papers, it's clear that there is a general lack of standardized and generalized robotic control. This may be in large part be a result of the lack of high-quality training data. In this project, we aim to strengthen the field by implementing Octo in our environment and fine-tuning it on our own dataset, additionally furthering the field by contributing to the lack of robotic training data. To achieve this, we have worked from this problem statement: *"How can Octo be used to perform low-level control of a collaborative manipulator in an unknown environment?"*

Chapter 3

Methodology

3.1 Methods overview

For this project, we set out to implement Octo, along with a set of different technologies enabling Octo to be trained and run locally at the university. From a hardware perspective, five different components are used, a manipulator (Franka Emika), a computer for running inference, a computer for controlling the Franka and two cameras, one for the wrist and one for a third person view. Additionally, a CUDA enabled GPU is also necessary to train or fine-tune the Octo model. These different components, except the machine used for training, can be seen on figure 3.1, where the data-flows from each component can be seen, along with what software stack runs on each computer.

The Real Time Computer (RT computer), is the one responsible for controlling the Franka manipulator. It does this using a *Polymetis* server, which sends desired joint positions to the Franka. In turn it receives the state from the Franka as well. The *Polymetis* server is used by the inference computer, by wrapping the *Polymetis* functions in a *ZeroPRC* server. This server is used by the *ZeroRPC* client on the Inference computer. The movement commands from the Inference computer, is generated by the Octo Policy, which in turn uses the Gym Environment and the two cameras, from which it receives a RGBD stream from.

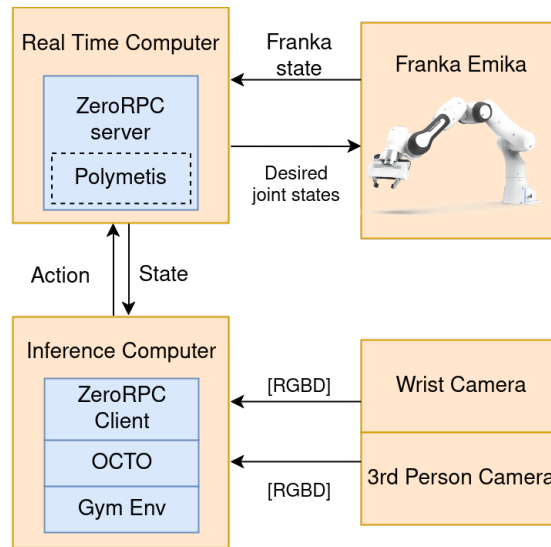


Figure 3.1: A block diagram of the architecture of the system running Octo inference. It shows the tasks of the various systems, along with what kind of data gets transferred from system to system. The blue blocks are software blocks running on physical local systems.

When fine-tuning the model, additional technologies must also be used. Besides from using different functionality from Octo, data must also be recorded. Octo is made to natively work with RLDS datasets, so we will use RLDS format for our datasets in this project. To create the datasets, a different policy for controlling the robot must also be implemented. For this, a custom VR policy was used. Additionally, a separate server must also be used to fine-tune Octo, as VRAM would be a blocking constraint on any laptop. This slightly different architecture can be seen on figure 3.2

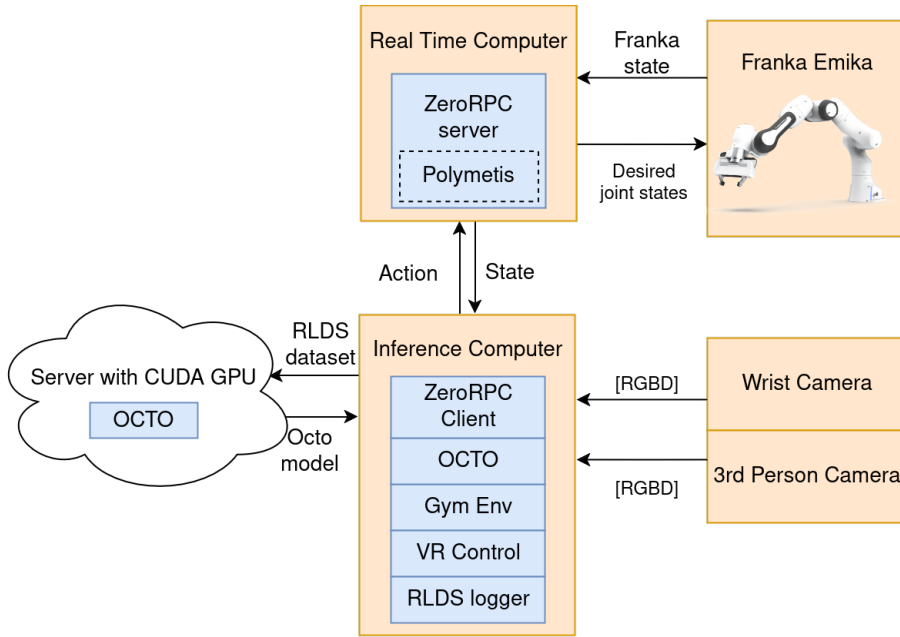


Figure 3.2: A block diagram of the architecture of the system running Octo fine-tuning. It shows the tasks of the various systems, along with what kind of data gets transferred from system to system. The blue blocks are software blocks running on physical systems. The blue blocks are local systems and the white cloud being a server for fine-tuning

3.2 Octo

For the policy part of the robot control, different systems as described in chapter 2 are available at the moment. The policy generation we used in this project is Octo. Octo is a novel approach to make a single model capable of performing various tasks and be compatible with virtually any type of robot, regardless of its specific model. While fine-tuning may be required to optimize performance for a particular setup / robot and the task, Octo is designed to function work with a wide range of robots, from a Spot robot dog to a 14DoF WidowX manipulator with dual end effectors. Octo is capable of performing some tasks zero-shot, i.e., without these tasks being present in the training data[26]. The reason Octo is capable of working across such a wide amount of different robots and

3.2.1 Modular Design of Octo

Octo's modular design allows it to adapt to different robotic platforms and tasks, making it a versatile tool in robotics. This flexibility is achieved through a transformer-

based diffusion policy that processes different inputs using a merged sequence of tokens. All of the input modalities: language instructions, goal images, and observation sequences, are tokenized into a common format, using specific tokenizers for each modality, e.g., using a CNN for tokenizing the image modality. The images are patched with 16×16 pixels sizes for Octo.

Patch size in pixels	Action head used	Diffusion steps for diffusion action head	Prediction Horizon
16×16	L1	20	50

History Window	Proprio-ception	No. Of cameras	Partial fine-tune (Frozen weights)
1	Enabled	1 (3rd person)	Disabled

Table 3.1: Tables showing the default values of the configurations for Octo in the fine-tuning scripts, based on the official example fine-tuning script.

Octo uses a transformer backbone with a special attention mechanism. This setup ensures that observation tokens only use information from the same or earlier time steps, maintaining the sequence’s order. One of Octo’s strengths is its efficient fine tuning capability, which can retain pre-trained weights and allow for the addition of new positional embeddings, encoders, or output heads as needed, without needing to do a "full fine tune". A full fine tune, is when you fine tune the entire Octo model i.e., the tokenizers, the entire transformer backbone and the action heads. This modular approach enables the seamless integration of new sensory inputs and action outputs, making Octo adaptable to different robot configurations and tasks. Some important design choices support Octo’s modularity and scalability. For instance, it uses shallow CNN patch encoders for image inputs, focusing computational effort on the transformer backbone. This is different from other models that use deep image encoders, making Octo more effective with large and diverse datasets. This also enables Octo to run and train on more accessible hardware. Additionally, Octo uses a simple channel-stacking method for goal images, to also achieve lesser compute load[26].

3.2.2 Additional Octo components

With Octo having such a modular architecture, as described in the previous subsection, enables Octo to have a highly modifiable observation space and action space. Not only can the action space and observation space be modified to work on vir-

tually any robot with a Gym interface, Octo also has several Gym class wrappers for further modification of the action heads and observation heads.

Action heads

Octo has several different type of action heads. The L1 action head (uses L1 loss function) is the default and is used in several of the examples, but did not provide the best results[26]. The diffusion action head uses a diffusion decoder to generate actions, and uses 20 diffusion steps. The diffusion action head was what most of the results in the paper was based on, but it does not work well on all types of robot setups [37]. However it should work on the Frank Emika robot, which we use in the project. Additionally there are a couple additional action heads, but these do either not work very well, as described in their "Things that did not work" appendix [26], or are not described at all but present in the code. Whichever action head is chosen for the fine-tuning can be defined in the parameters when instantializing the action head in the fine-tuning script.

Additionally, all action heads have the capability to output a arbitrary amount of actions from a single observation, an "action-prediction-horizon". The default value for this is 50 in the fine-tuning script, but it is recommended to be 4 or below for most robot setups. For using this action-prediction-horizon in inference, an object wrapper has been made available for this purpose. All of the default configurable values of Octo can be seen on table 3.1.

Observation space

Besides from a high level of customizability in the action heads, the observation space in Octo also has a high degree of customizability, with built in functions to support it. Octo can support multiple cameras, with people in GitHub claiming to use three cameras with success. It can use basically any sensor data available, including proprioception(information about the robot's state). Additionally, Octo also has support for a history window, which are previous observation instances, which are appended to the current observation. In the first step, no previous observation would be available, but Octo automatically zero-pads the missing observation[26]. Not much information about the History window is available, besides from the appendix where they mention it improved performance. Additionally, in the GitHub they recommend using a history window with a size between 0 and 5, 1 being the default. This observation space is defined when instantializing the model, e.g., when fine-tuning a previous model for a new observation space. When using the model, Octo also has built in object wrappers for the Gym environment, which handles the history windows during inference.

3.2.3 Octo Availability

Octo is a completely open source project, with open source weights as well. The code base for Octo can be found on their GitHub, and the pre-trained Octo Models can be found on HuggingFace. Octo has several sizes of their model on HuggingFace, but only two of them are documented in the GitHub and their paper; Octo Base and Octo Small. The difference in size in these models are the amount of parameters in the transformer backbone, with Octo Base having 93 million parameters and Octo Small having 27 million parameters[26]. Additionally, it is possible to instantiate a completely new transformer backbone using the Octo code, and it is even possible to change the amount of parameters to an arbitrary number.

3.3 Reinforcement Learning Dataset

To offline train any neural network, training data is needed. Depending on the type of neural network and how it is trained, the data structure will vary. A "regular" CNN can be trained using standard image formats like PNG and possibly an accompanying file containing annotation data, such as a JSON file, which is the format used in COCO annotations [38].

When training RL networks offline or doing imitation learning, the structure and types of data needed to be saved can vary quite a bit. RLDS (Reinforcement Learning Dataset), developed by Google Research, is a standardized way to record sequential data from any Gym-based environment. Many previous dataset structures were tailored to specific experiments or needs. This often resulted in necessary information not being available for other experiments or information being interpreted incorrectly. This is a problem that has held back the robotic industry, as there is a lack of general training data, and the data that exists may not be usable due to formatting[34].

An RLDS dataset can consist of many episodes, with each episode typically containing multiple steps. An episode represents a complete scenario; for example, when recording data for picking up a brick, each time the robot resets and picks up a brick constitutes one episode. Each episode will likely contain multiple steps. These steps include the observation of the system (the obs variable in a gym environment) and the action generated by the policy used in the gym environment, using the observation from the current step. Additionally, each step contains a reward for training RL networks. Metadata for each step is also included, along with metadata for each episode. The dataset itself contains metadata, which includes details such as the number of episodes, data structuring and sharding, and the dataset name.

The general structure of an RLDS dataset can be seen on figure 3.3

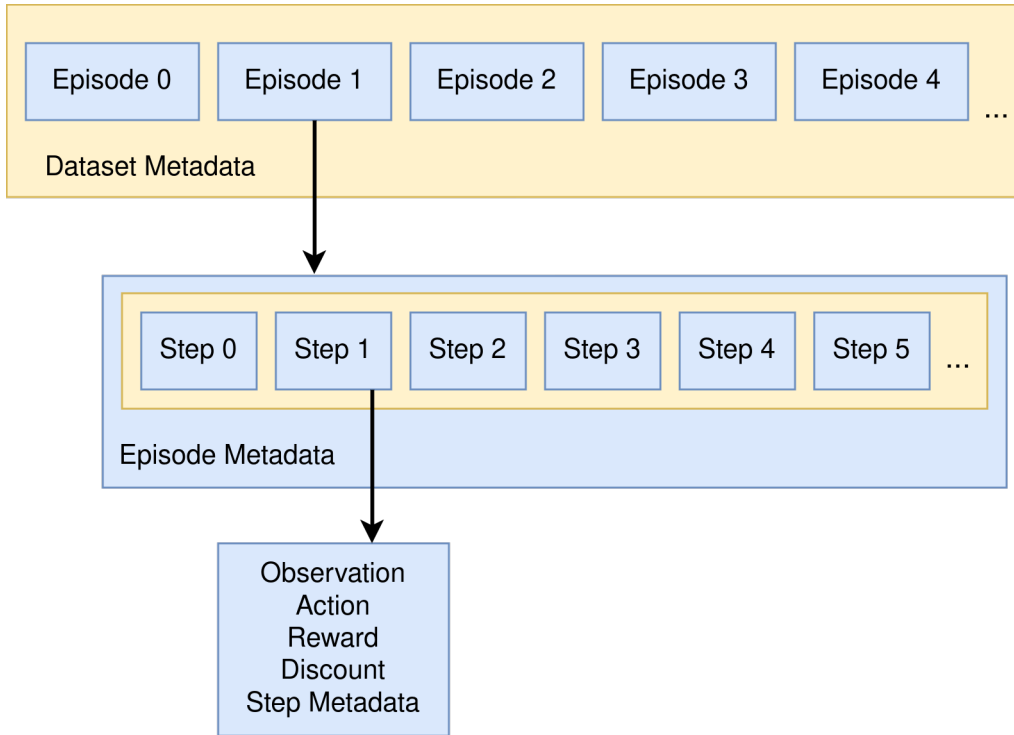


Figure 3.3: The diagram illustrates the structure of RLDS (Reinforcement Learning Dataset). The top section shows the dataset metadata encompassing multiple episodes. Below, Episode 1 is expanded to reveal its structure, which includes multiple steps. Step 1 is further detailed to show the various data fields present in a step: observation, action, reward, discount, and step metadata.

To use and build RLDS compatible datasets, Google Research has released two different tools, *EnvLogger* and *RLDS Creator*, which can be used to create RLDS datasets. As RLDS is based upon TFDS (TensorFlow Dataset), any RLDS dataset can be loaded and edited using the TFDS tools.

3.4 Polymetis

For controlling the Franka Emika, some sort of real time control will be needed. As the control will need to be unblocking, the robot will need a 1 kHz control signal[39]. *Polymetis* is a tool built by Meta research, and capable of handling the high frequency control of the Franka Emika. *Polymetis* is an open-source robotics middleware, made to simplify the development and deployment of real-time control algorithms. It provides a modular framework that allows for easy integration with different kinds of robotic hardware and software stacks. *Polymetis* supports control

loops with a frequency of 1 kHz, which is required to be able to keep the Franka Emika robot stable and responsive. *Polymetis* has a modular architecture, separating different control components, which enables the testing and development of individual control methods, without affecting the rest of the system. *Polymetis* has built-in tools for monitoring system performance and error handling.

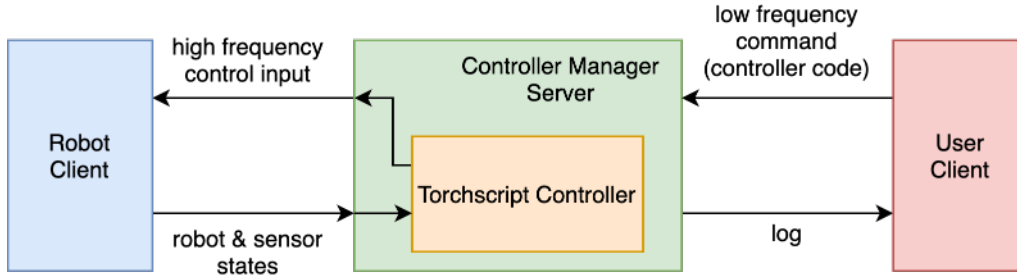


Figure 3.4: Diagram from Polymetis documentation [40], explaining the relationship of the different components in Polymets

The main component of *Polymetis* is the Controller Manager Server with the TorchScript controller. The Controller Manager Server acts as the main control unit, overseeing the various control tasks and ensuring coordinated execution. It also responsible for interfacing with the robot hardware, handling the real-time data communication necessary for high-frequency control. The TorchScript controller is the part of the Controller Management Server that handles the actual robotic calculations, from the commands received by the Controller Manager server, sent by the User Client. The TorchScript controller, can be replaced by a custom controller.

3.5 ZeroRPC

ZeroRPC is an open-source Python library designed for Remote Procedure Calls (RPC). It is based on the ZeroMQ and MessagePack protocols, both of which are open-source as well. An RPC service allows a program to make function calls in a different address space. This simplifies the development of distributed systems, like the one created for this project, by abstracting the networking and communication protocols into a single library. This is useful in our project, as we will have one computer running a real-time kernel and another running the robot environment and the policy. While it would be simpler to run inference on the computer controlling the robot, the real-time requirements of the Franka control prohibit any compute-intensive applications on the server, besides the *Polymetis* server. By using *ZeroRPC* and exposing a class to control the robot's movement on the real-time kernel computer, the other computer (the inference computer) can use the objects and

methods as if the code were running natively. *ZeroRPC* is also based on lightweight protocols, keeping the system resource usage to a minimum[41].

To use *ZeroRPC*, the server side has to be initialized as a server. This is done by instantiating a "server" in Python, which references an internal object. The server is bound to a port on the server itself, then it can be run. On the client side, a client object is instantiated and connected to the server's endpoint. The server object can now be used natively by the client as if the code was on the client side all along.

Chapter 4

Implementation

In this chapter, the implementation we did for this project is described, along with some of the challenges we encountered in the process. All aspects of the implementation are described in the chapter, from robot control and data recording to how the training data for the training was captured.

4.1 Implementation overview

The implementation of this project has largely relied on the software aspect. As most of the software components used in this project come from open-source GitHub repositories, implementation was a matter of getting these components to work with each other but also working on the specific robot used for this project, the Franka Emika. While getting many of the systems running with the Franka did prove troublesome, simply getting some systems running at all proved an even greater challenge, as many open source codes stacks are badly documented.

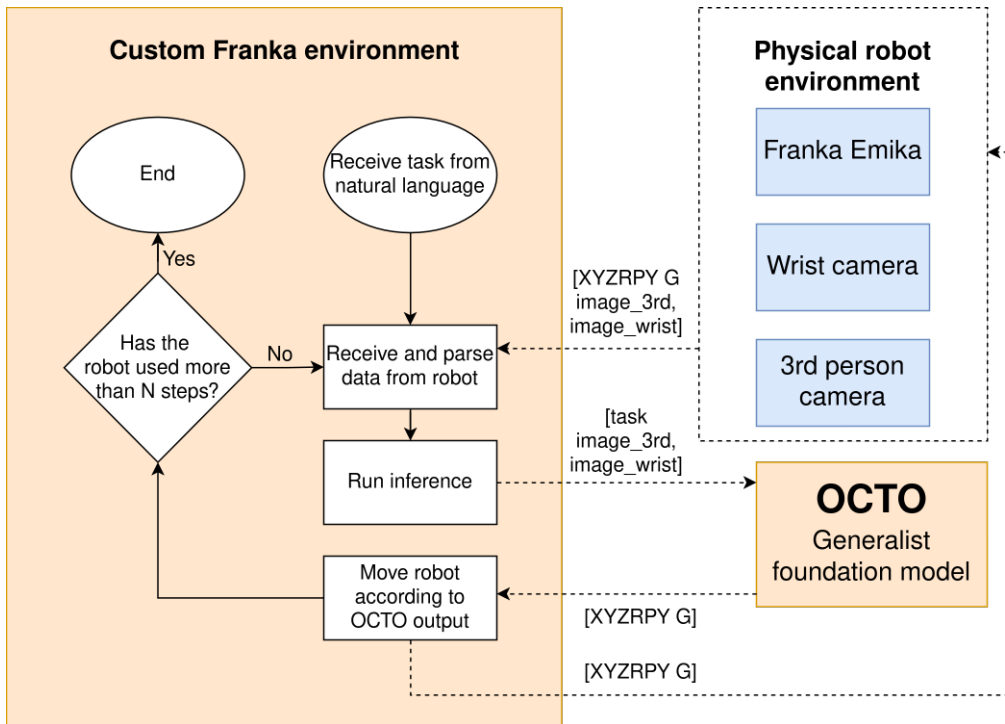


Figure 4.1: This flowchart shows how the Octo model is utilized for inference. All the dashed lines signifies data being passed from component to component, with the data structure being written near the lines

The general flow of our Octo implementation, as can be seen in figure 4.1, starts with getting a command in natural language. After that it gets the robot's initial state (Cartesian position) and camera data (RGBD) from the physical environment. It uses this data to run inference using a fine-tuned Octo model. The action output by the Octo model is then used to move the robot. This is repeated for N steps, and after that the inference loop will be finished. The implementation of Octo and the preceding steps to get it implemented are documented in the coming sections. For the cameras, two Intel RealSense D435 were used. A Lenovo T14 Gen 2 with an Intel I5, was used for the real time computer, which was the computer controlling the robot. For inference, a Lenovo P53 with an Intel I9 and an Nvidia Quadro RTX 4000 was used. The inference computer was also used when recording data. When recording data, a VR policy was used. For the VR policy a Meta Quest 2 headset was used. Finally, a Franka Emika was used as the manipulator. A picture of the entire physical setup can be seen on figure 4.2. When fine-tuning Octo, it was done on a server with an NVIDIA RTX A6000.

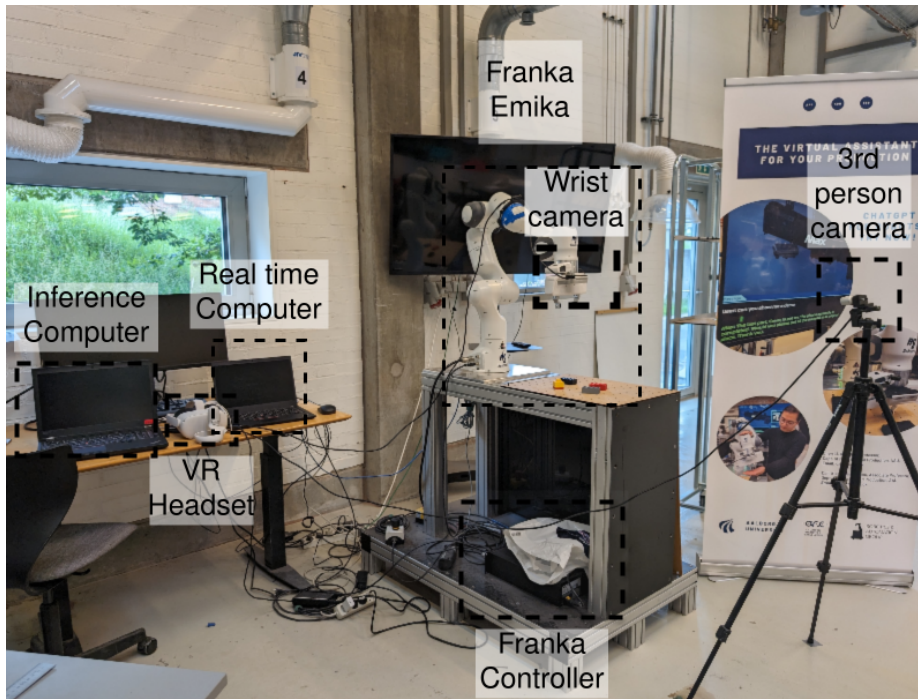


Figure 4.2: This figure shows all of the different components used for recording data and running inference, including the robot itself and the VR headset and its controller.

Figure

4.2 Data capture

4.2.1 Franka Environment

Franka Environment handles all the input and output for the Franka[33] robot. It uses the Polymetis[40] API to communicate with the robot, which was explained briefly in section 3.4. It also includes the functions and wrappers for a robot agent/policy, which sends the control commands to operate. This section explains the "Real Time Computer" in figure3.1

We looked into various repositories to develop our Franka environment, Franka-PolymetisEnv. We checked how others implemented their robotic control stacks, and we took the good ideas of each implementation and applied them to our own stack[42, 43, 35]. We are mostly inspired by *Robot-Lightning*[42] repository that also implements an environment for the Franka robot. We found *Robot-Lightning*[42] implementation had some good features to adopt in our implementation, among which:(1) Modular and lightweight code, and (2) taking into account the user and physical constraints when updating the robot joint positions, more about these

constraints later in this section.

`FrankaPolymetisEnv` consists of various function wrappers for Polymetis API functions written in Python. To use Polymetis, two different servers should be launched beforehand. These servers are run inside the Realtime Computer:

- (1) Franka Arm server: This server handles configuration of the API workspace limits (there are also soft workspace limitations in the higher-level Python codes), frequency operation of the internal controller (default=1000Hz), stiffness parameters of the impedance controller, tuning collision behavior (adjusting the extent of torque for a joint to prevent it from locking), and other initialization values [40]. These parameters can be set when launching the server.
- (2) Gripper server: This is the dedicated server for communication between the gripper hand and the Realtime Computer. The gripper can be either the main Franka Hand or any other third-party gripper. The configuration of the gripper parameters can also be set at launch.

Termination of these servers depends on user demand, or they can also be terminated if a severe communication error occurs between the Realtime Computer and the Franka Controller or if internal hard-coded safety limitations for the arm or gripper are violated. As indicated above, Polymetis differentiates the Franka Arm control group and its gripper. The advantage of this is that transferring to a new gripper's API would be easier if using another gripper other than the default Franka gripper.

To produce the necessary methods for the next parts of the project, most of the functions in `FrankaPolymetisEnv` were developed by us, while some were inspired by *Robot-Lightning*[42] and modified to meet our needs afterward. Here is a selected list of these functions:

- `robot_init()`: Creates the robot arm(`RobotInterface`) and the gripper(`GripperInterface`) Polymetis API objects. The servers mentioned earlier should be launched so this function executes successfully. Additionally, it starts the user-selected controller, which can be a *joint impedance controller*, *Cartesian impedance controller*, or *joint velocity controller*. The joint impedance controller is the default controller used in this work. `robot_init()` also imports the configuration file that contains all the constant values for safety boundaries and coefficients that are used in `FrankaPolymetisEnv`.
- `get_state()`: Combines `get_arm_state()` and `get_gripper_state()` to return the current state of the whole robot. It also saves the previous state of

the robot. Each robot state contains information about joint angles and velocities, Cartesian position and Euler orientation, gripper width, success status of the previous command, timestamp, etc.

- `go_home()`: Sends the robot to the pre-set robot home position.
- `reset()`: Executes the `go_home()` function, and if `randomize` flag is `True`, it moves the robot to a new position within the workspace boundary. It uses uniform distribution to randomize the position.
- `primitive_joint_move()`: Can move the robot to the desired joint or end-effector positions using the Polymetis joint PD controller. It is generally applied for longer actions(takes more than 0.5 seconds to complete).
- `primitive_delta_move()`: It is the extension of the primitive `primitive_joint_move()` function. If the input is Δp , and the current and final positions of the robot are p_1 and p_2 respectively, the equation $p_2 = p_1 + \Delta p$ should hold. However, applications of this function are limited as each time it is executed, it restarts the controller(policy) and causes communication errors, which is discussed in section 4.2.1.
- `solve_inv_kinematics()`: We created this wrapper on top of the current inverse kinematics solver function to incorporate any desired inverse kinematics solvers. The reason for using a different solver other than the default one can be that some solvers have better convergence for a stable joint angle distribution. Ideally, a good solver should keep the joints near the middle of the joint's angle limitations. We experienced that the default solver(*Pinocchio*[44]) used in Polymetis often led the robot to unstable positions and close to joints' limit boundaries. Inspired by *Droid*[35], we tested `dm_control`[45], which worked slightly better for us, though the solver parameters need more fine-tuning. Other solvers should also be explored in future works.
- `observation_space()`: It is the dictionary of boundaries for joint angles, joint velocities, Cartesian position, Euler orientation, and the gripper width used in the correspondent Gym environment[42]. These boundary values are obtained intuitively from tryouts.
- `action_spaces()`: Depending on the type of action command (delta joint angles, delta Cartesian EE pose, pure joint angles, or pure Cartesian EE pose), similar to `observation_space()`, it returns a dictionary for the *Gym* environment of the action space boundaries. The values for these boundaries are set within the hard safety limitations.

- `update()`: This function is the most imperative part of the environment because it will directly be deployed for implementing the `step()` in the *Gym* environment 4.2.3 later on. The function does these subfunctions in order for an input:
 - `calculate_action()`: For an input action command (can any of the types in `action_spaces()`), it filters the action command recursively step-by-step in terms of the defined soft limitations for workspace, inverse kinematics success, joint positions, joint delta positions (increments on the current positions) [42]. In the case of violation of any limitation for a value, the function clips that value and calls the `calculate_action()` returns the filtered action in delta and pure positions for Cartesian EE pose and joint angles. The output is assured to be an executable action that satisfies all soft safety limitations.
 - `update_reference_tracker()`: Updates the reference tracker, simply an indicator for a given calculated action, ideally, what the robot state should be after executing the action. This function is meant to be applied when the input action type is either "delta Cartesian EE pose" or "delta joint", because it tries to keep track of the small incremental movement for each iteration with respect to the start of the trajectory and if the action type is pure positions (joints or Cartesian EE pose), the function just assigns the input value itself to the reference tracker states. More about the necessity of this process is explained in section 4.2.1.
 - `update_controller()`: Sends the desired positions to Franka. This is where the robot is supposed to move. The input should be in pure position format. Depending on the selected controller type in `robot_init()`, `update_controller()` uses one of the functions in `update_X_desired()` to run the corresponding controller. `update_X_desired()` includes: (1) `update_desired_joint_positions()` for the joint impedance controller; (2) `update_desired_ee_pose()` for the Cartesian impedance controller; (3) `update_desired_joint_velocities()` for the joint velocity controller.
 Moreover, to prevent occasional joint locks caused by errors from Polymetis [40] (such as communication errors, discussed in section 4.2.1, and excessive torque errors), an exception handler is used. This handler catches the error, restarts the controller, and resumes the current action command.
- `close()`: Executes the closing procedure, executes `go_home()` and terminates any existing running policy.

Additionally, we developed some helper functions that are used in the main functions, including:

- `min_jerk_trajectory()`: Exposes the Polymetis function for minimum trajectory planner in XYZ space.
- `forward_kinematics()`: We use this function numerous times to mostly convert joint space values to Cartesian+Quaternion space vector.
- `quat2euler()`: Many functions like this were created for converting an orientation vector to our desired coordination system. They are mostly powered by Scipy library[46].
- `open_gripper()` and `grasp()`: These two functions are the Polymetis APIs used for working the Franka Hand. In our project, the gripper is used in a binary context. Therefore, `open_gripper()` just opens the gripper, and `grasp()` simply closes the gripper until it detects a certain amount of force while it is not completely closed, which indicates an object has been grasped.

In the following section, we discuss in more depth how our `update()` differs from the *Robot-Lightning* way.

Reference Tracker

We explain the reason for adding the reference tracker with an example. Suppose the robot needs to execute a trajectory from EE point $P_1 = [0.4, 0.0, 0.3]$, where units are in meters, to point $P_2 = [0.6, 0.0, 0.3]$. In this case, the minimal jerk trajectory function generates an almost 20 cm straight-line trajectory for the EE to follow in a control loop. The trajectory action commands are converted from a pure EE pose trajectory to a delta EE position format to emulate a realistic scenario. Mathematically, this can be shown as:

$$\text{min-jerk-trajectory}(P_1, P_2) = T = (\tau_1, \tau_2, \dots, \tau_n) \text{ such that: } \tau_1 = P_1 \text{ and } \tau_n = P_2$$

where T is the the set of the generated sequential trajectories and τ_i represents each of the trajectories. If we define T_Δ as the sequential delta-pose trajectories and δ be each one delta-pose values, the above equation can be converted to:

$$T_\Delta = (\delta_1, \delta_2, \dots, \delta_{n-1}) \text{ such that } P_2 = P_1 + \sum_{i=1}^{n-1} \delta_i \text{ and } \delta_i = \tau_{i+1} - \tau_i$$

Then, if we set n to be 100 and the control loop frequency to 20 Hz, and let the u

`update()` function run the delta actions without the reference tracker (as it is out-of-the-box from *Robot-Lightning* [42]), the robot does not reach P_2 . This can be explained by how *Robot-Lightning* updates the desired position for `update_controller()`. For each new δ received by `update()`, after the `calculate_action()`, the robot's current state is added to the δ value to get the desired pure joint or EE pose position for the controller to execute. Ideally, this should correctly process a new δ . However, at execution, during execution, the robot has millimeter-scale accuracy errors (1-2 mm in Cartesian space). When our δ values are small enough and within that range, the controller incorrectly assumes it has reached the desired position earlier than expected. This has a cumulative effect over all the steps in the trajectory that prevents the robot from completing the trajectory.

The approach we took is to make sure the robot is aware of the desired pure positions with respect to the starting state of the trajectory. To formulate this, the update method changes from:

$$C_k = \delta_k + S_k$$

where C_k , S_k , and δ_k are action pure position, the robot's state, and the delta all at time-step k to:

$$C_k = P_o + \sum_{i=1}^k \delta_i$$

where P_o is the initial state of the trajectory. We assign the robot's current state to P_o at the end of the `reset()` function when initiating a new task. We also reinitialize P_o whenever a joint lock happens in the exception handler in `update_controller()` to avoid joint locks causing drifts between the desired positions and the obtained values in our formula.

To summarize the differences of our Franka environment to *Robot-Lightning*: (1) We modified the structure of the Polymetis class to be considerably more modular and easier to add customized functions like external inverse kinematics solvers, PyTorch controllers[40], etc.. (2) We added the Reference Tracker which leads the robots where it is ideally supposed to be at.

Troubleshooting

During the project, we encountered "communication_constraints_violation" errors in the Polymetis Arm server, which results in interruptions to the current running policy of the robot and caused erratic and junky behaviors from the robot. FCI[39] states that this error happens when the Inference computer connected

to the Franka Controller cannot handle more than 20 consecutive packets from Franka. The error can have different sources, each of which can be sufficient to reproduce it: (1) Faulty or old Ethernet cables (CAT5) used for connecting the Realtime Computer and the Franka Controller; (2) The CPU of the Realtime PC is not powerful enough to manage the communication load sufficiently; (3) Possible malfunctions in the Franka Controller network module; (4) The installed real-time kernel might cause slow performance issues.

To minimize the occurrence of the error, we took the following measures: (1) Disabling CPU scaling and setting the CPU in "performance" mode and other CPU-performance-boosting techniques. (2) After trial and error, it was observed that the error is much more frequent in the Cartesian impedance controller mode than in the joint impedance controller mode. Therefore, the joint impedance controller was used as the default controller type for the `update()` function and other primitive functions used for later usages [47]. Moreover, `primitive_X_move()` functions were used minimally because of their heavy load of executions(they restart and send policy for each call, which frequently causes the error.). (3) As the error is most likely to happen for `update_desired_joint_positions()` function, putting an exception handler to prevent the program from closing and restarting the controller to resume the policy also helped deal with this issue. (4) Tested multiple real-time kernel versions and chose the one with the best scores in the Polymetis and FCI communication benchmarks. (5) Although selecting the internal controller frequency from 1000 Hz to a lower value like 500 Hz decreases the chances of the error occurring, it doesn't solve the actual issue as the problem is processing the packets for each step, and it is independent of the time interval between the steps. (6) Changing to faster cables(CAT6A), turning off Bluetooth and Wi-Fi, and closing programs unnecessary to the robot execution like browsers also helped mitigate the issue.

Interface server

There are two ways of exposing the `FrankaPolymetisEnv` to be used in the Inference computer:

- (1) Using the Polymetis package on the Workstation PC, which requires installing it in the same Python environment where *Octo* and other packages are located,
- (2) Wrapping the Polymetis class around an RPC server like ZeroRPC[41], then exposing the needed functions of `FrankaPolymetisEnv` in the ZeroRPC server inside the Realtime Computer, then to be used in the GPU-enabled PC by connecting to the RPC server via its client's API.

The problem with method (1) is the coexistence of Polymetis with other packages like *Octo*, *JAX*, etc., in a single Python environment. There are numerous dependency issues during both installation and runtime for Polymetis packages

to be compatible with the Inference computer. Also, the Polymetis repository was discontinued to be maintained after early 2023. For this project, we used a community-maintained fork of Polymetis, *monometis*[48], which changes the required package versions to be compatible with a newer Python version (namely 3.9). However, even with *monometis*, the issues were not solved.

After many attempts and tweaking the versions of infrastructure packages to make Polymetis compatible with other packages used on the Inference computer, testing it all to see if the environment works stably at runtime, all the tries failed either in build-time or in runtime. It was concluded after this to go with method (2).

The concept of incorporating ZeroRPC was first seen in *Droid*[35] code base, however we saw the adaptation *Robot-Lighting*[42] fits the best with our implementation. In order for ZeroRPC to work, there are two sides: the server side and the client side. We outline the important details of both sides below:

- **Server Environment:** We create a ZeroRPC Server object, which requires `FrankaPolymetisEnv` as the input. Using the common method to launch the server, we first bind the server object to listen to all network interfaces on port "4242" in TCP protocol and run the server afterward. However, the ZeroRPC package is only compatible with Python data types (`dict`, `list`, `tuple`, etc.) and `FrankaPolymetisEnv` outputs contain various third-party object types such as `Numpy`, `PyTorch`, and `Gym`. To solve this, we create an environment, `ZeroRPCPolymetisServer`, that wraps `FrankaPolymetisEnv` functions, modifying their return types via a type converter function called `parse_to_lists()` to convert all non-Python types to common Python types[42]. We use the new environment to create the ZeroRPC Server object. Overall, it runs on the Realtime PC and allows accessing to `FrankaPolymetisEnv` environment, which itself is connected to the Polymetis launchers servers (Arm and Gripper).
- **Client Environment:** To connect to the server from the Inference Computer and use the `FrankaPolymetisEnv` as if there is no ZeroRPC process in between, we need to create a client environment where the function names exactly match those of the main class functions. This environment, `ZeroRPCPolymetisClient`, at initialization, connects to the IP address of the Realtime PC (either wired or wireless) with port "4242". We set a static IP address to Realtime PC to avoid issues of possible changes in the IP of the target for a new try. With a similar logic on the server side, to use types as they are intended, we employ the function `parse_from_lists()`[42]. This function converts Python data objects to their original non-Python types (such as `Numpy` arrays, `Gym` boxes, etc.).

One of the advantages of using this approach is once the robot is up and running on the server, there is no downtime anymore to wait for the robot to launch on the client side, and the client program can be run multiple times without the need to launch the server more than once. In addition, ZeroRPC does not require any extra packages for build or runtime.

Apart from adding our own functions and customized types to the ZeroRPC client and server in our implementations, we based our work on the *Robot-Lightning* code[42] in this part.

Figure 4.3 demonstrates an overview of our Franka environment, which contains various components that we explained in this section.

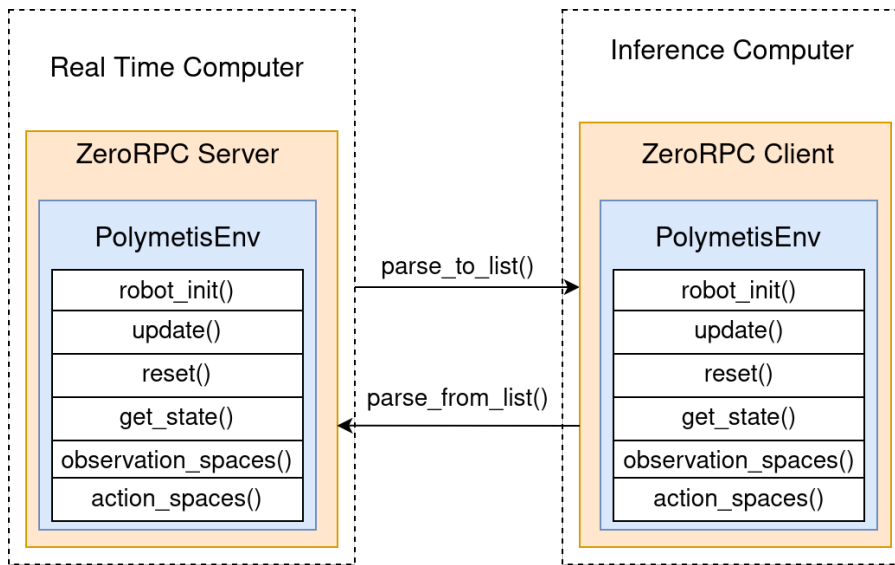


Figure 4.3: This block diagram depicts the ZeroRPC server and client relationship, and the Polymetis client and its recreation on the inference computer, using ZeroRPC.

4.2.2 Camera Environment

This is responsible for the perception part of our stack. It consists of cameras, one located on the Franka Hand("wrist" camera) and one that is viewed from a third-person viewpoint("primary" camera). "Depth Camera D435 Intel RealSense" is the model of both cameras for this project. It is capable of capturing RGB images together with depth images. One can use the official SDK ("librealsense") Python API or the OpenCV camera streaming feature to access the camera images. Both methods are available in our implementation and can be used based on the context, with a preference for using the official method. The official SDK has access to more features of the camera that can be customized to our needs.

We draw inspiration for our perception stack implementation from the following repositories: [40, 42, 49]. The entire pipeline consists of three modules, which are described in order from low-level to high-level:

- `RealSenseEnv`: This class initializes the camera and starts the camera streaming pipeline. The main functions of `RealSenseEnv` include obtaining the camera's intrinsic and extrinsic parameters, receiving frames from the camera (RGB-D), and visualizing the received images. An object of the class maps to only one of the cameras and runs on the main thread in Python. This means it works in a blocking manner, which is not ideal as we need to stream two cameras while the robot and other components run in parallel, leading to the following environments.
- `RealSenseThreaded`: It creates an object of the above class and then passes the stream pipeline on a thread using `threading` library. The stream of the frames from the camera continuously updates a queue with a length of one, so for every new frame, the queue is updated with the latest frame. In this way, the camera can manage occasional performance dropouts and ensure that the latest frame isn't returned null.
- `MultiCameraWrapper`: It creates instances of `RealSenseThreaded` for both cameras and assigns them on different threads. It has two functions that all subsequent modules need to use the cameras.
 - `latest_observations()`: Combines the latest frames from the cameras and returns all the frames in a dictionary data type.
 - `stop_cameras()`: Calling this function stops the cameras from streaming and closes the cameras without causing a crash.

Overall, our perception pipeline, which runs on the Inference computer, assures that the *primary* and *wrist* cameras are streamed in a thread-safe and non-blocking approach. The default resolution of the cameras was set at 480x640, which is the most common for the D435 model. Moreover, we have also done Eye-to-Hand and Eye-in-Hand camera calibration during the early stages of the project, which can be used for future phases of the project.

4.2.3 Gym Environment

This is the main environment where the Franka and the perception modules are integrated together to complete the robot environment class. We use the standard *Gym* structure to create our `RobotEnv`[50]. The following explains the crucial parts of the *Gym* environments:

- `init()`: The camera and the robot environments are instantiated using the `ZeroRPCPolymetisClient` and `MultiCameraWrapper` classes, as introduced in previous sections. Additionally, it defines:
 - `observation_space`: Merges the `observation_space` defined inside the main Franka class with the boundary boxes of 4 different images (depth + RGB for 2 cameras).
 - `action_space`: It is a replication of the `action_space` defined in the Polymetis Franka class.
- `get_observation()`: This function calls `latest_observations()` to get the latest camera frames and `get_state()` to get the Franka state from their respective environments. It then concatenates and returns them.
- `reset()`: Essentially, it runs Franka's `reset()` function, which sends the robot to its home state. This is executed at the start of a new task.
- `step()`: Given the input action obtained from the policy function, it performs the `update()`. Then, after a precise wait to ensure the control loop frequency is consistent, it calls `get_observation()` to update the robot's state. If there is a reward from the environment, it is calculated afterward. Additionally, `step()` checks if the current step violates the time limit `set(truncated)` by the user and determines if the current run should be terminated based on the robot's state. Finally, the function outputs a tuple consisting of the observation, reward, termination status, truncated flag, and any other information gathered throughout the function.

The *Gym* environment provides a proper platform to work with such a sequentially structured system. Assuming we have established a robot policy, the following section discusses how to save and log relevant robot information for training a model.

4.2.4 Envlogger

There are various ways to save and record a dataset. However, we want to use our dataset to train a model like *Octo*. Moreover, *Octo* follows the same approach as *Open-X Embodiment* (OXE) [32] to handle the dataset [26]. Therefore, we should aim to use the same dataset structure as OXE, namely RLDS, which was introduced in section 3.3.

One way to transform a dataset into the RLDS format is to first collect our dataset and serialize it using common libraries like "h5py" or "pickle" and then use OXE

tools to convert it to the desired structure[32]. The other way, which is what we chose to do, is using the *EnvLogger*[51], a library designed for this purpose created by the same group that developed RLDS[51]. We have found that *EnvLogger* is less explored in the creation of related robotics code stacks; one reason for that can be that the package is relatively new, and *OXE* has just started taking off since the last quarter of 2023. Using *Envlogger* prevents dataset conversion after collecting and also the problems that come with different serializations. Furthermore, if there are probable post-processing modifications to the dataset, efficient tools for that can be created, ensuring the datasets' format remains consistent.

To implement *EnvLogger*, we modified *oxe_envlogger*, a repository that wraps *EnvLogger* to make it easier to use with *Gym* environments. At the time, this repository was unstable and couldn't be used right away, so we modified some components of *oxe_envlogger*, especially in type and image casting, to be able to be used with our dataset features and *Octo*. Note that *oxe_envlogger* built itself mostly on top of the examples of *EnvLogger* and *RLDS* repositories.

oxe_envlogger has two variants for *EnvLogger*: *OXEEnvLogger* and *AutoOXEEnvLogger*. They both follow the same logic to record the data; however, based on our experience testing them, *AutoOXEEnvLogger* was more comfortable for us to use to work and had few errors to deal with.

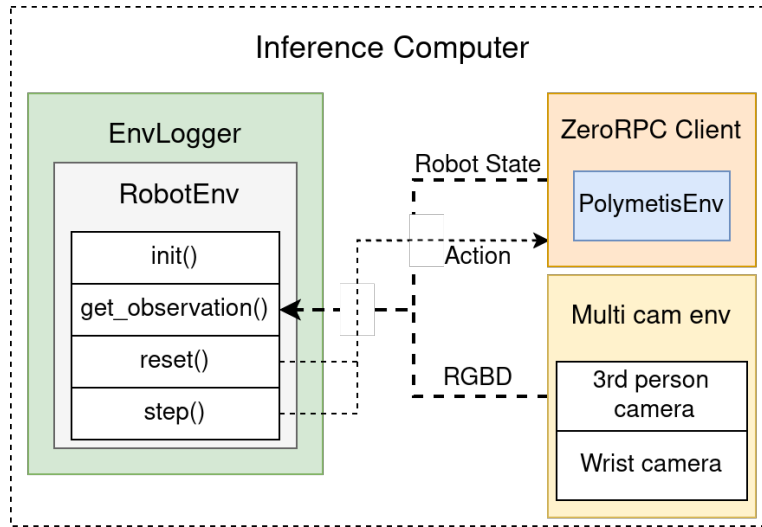


Figure 4.4: This block diagram depicts the RobotEnv and its usage of the ZeroRPCPolymetisClient and the MultiCameraWrapper environments. The dashed lines depict data streams. The thicker dashed line is only a visual aid since the two data streams cross.

Creating `AutoOXEEnvLogger` environment requires a one-line script that the current *Gym* environment, dataset name, and dataset directory are passed as input, and the return is an object the *EnvLogger* environment, which can be used similarly to any *Gym* environments. To understand better how saving a dataset works here, consider the scenario where the user wants to record 10 episodes for their dataset, and each episode contains a limited series of steps that contain information about the actions and observations of the real-world environment with which the robot was interacting. Normally, a new episode starts or ends with the `reset()` function in the *Gym* environment. Since *EnvLogger* wraps both the `reset()` and `step()` functions, logging all inputs and outputs, every new episode is saved with each call to `reset()`. Aside from that, *EnvLogger* allows the addition of episode and step metadata to be saved, which is useful for language labeling applications later on. The saved dataset files come with the "tfrecord" file extension, which is the standard TensorFlow dataset extension. The configuration for serializing the dataset can also be changed on-demand by the user. For example, whether each "tfrecord" file represents an episode in the dataset or if all the files in the dataset have the same storage size can be defined by the user.

Figure 4.4 illustrates the integration of our cameras and Franka environments to build `RobotEnv` in section 4.2.3, which consequently is used to create the `EnvLogger`.

4.2.5 Data Recording

Here, we explain the recording of an episode we implemented for our dataset. Check out section 4.2.7 regarding the robot's policy for our dataset. The recording pipeline starts with the initialization of `RobotEnv`, which prepares the Franka and the cameras to run. Then, the *EnvLogger* is initiated. Now, the robot interactions can be logged. As mentioned earlier, *EnvLogger* can add metadata for each step and the entire episode. For our project, the metadata we used were "language_instruction" and "episode_status". The first describes the task done in this episode, and the latter is a flag whether this episode was a "successful" or "failed" one, which is decided by the user before starting the next episode (`reset()`). Moreover, as this dataset should comply with *Octo* structure, each step of an episode must have "language_instruction"; we included that in each step's metadata, not just the episode metadata.

In parallel to *EnvLogger*, we added a trajectory replayer function, where we manually append the actions sent to the robot to a list. At the end of the episode, the robot can replay these actions to replicate the exact trajectory of that episode, providing a more hands-on way to keep track of the recorded dataset.

Furthermore, the RLDS library supports importing and merging from multiple dataset directories. This means that the whole dataset can be divided into collections of smaller datasets. We recorded our smaller datasets in batches of 5 episodes to build up to a larger dataset.

4.2.6 Dataset Loader and Modifier

Supposing that we have a dataset using the method of the previous section. This dataset is in raw format, and it is not yet ready to be used by *Octo* because we recorded the images in full resolution for both the "wrist" and "primary" cameras. *Octo* is trained with the camera images to be 128x128 and 256x256 resolutions, respectively. For the best performance, our dataset should have the same resolution. Aside from that, we should have access to solutions if needing a change in our dataset. Therefore, we developed `OfflineRobotEnv`, a class that reads our recorded RLDS dataset and transforms it into a *Gym* style environment. Functions of `OfflineRobotEnv` are:

- `reset()`: It reads the episode in the dataset, then initializes an iterator for iterating through all the steps in the episode.
- `step()`: Increments the iterator for reading a new step in the episode and returns the observation of that step. It does not need an input in this class.
- `get_action()`: Outputs the action for the current step.

In addition, it can read multiple datasets with different directories and treat them as a single dataset. Although it cannot directly interact with the real robot, it enables us to replay any desired episode. To do that, we simply use the `get_action()` values as input to our real world `RobotEnv` to replay trajectories. Moreover, to replay the images of `OfflineRobotEnv`, we just need to visualize the output of the `step()`. Overall, this approach gives us a more intuitive way to work with the collected dataset.

Using our `OfflineRobotEnv` and *oxe_envlogger's* `RLDSLogger` [52, 51], we create the following modifier functions for our dataset:

- `merge_dataset()`: Reads all the raw smaller datasets and then saves them as a new RLDS format dataset. It can also skip the episodes marked as "failed", making sure that only the "successful" ones remain in the new dataset.
- `resize_image()`: It goes through the dataset and resizes all images of the observations to the custom shape defined by the user and saves it as a new dataset.

- `change_language_embedding()`: This function replaces the "language_instruction" of the metadata with user input description.

4.2.7 Virtual Reality

To be able to move the robot and record the data, some method of robot control had to be implemented. We decided to use a VR(Virtual Reality) policy to control the robot when recording. VR control was chosen as it was easy to record, intuitive to use and the hardware was available in the lab. The VR headset used was the Meta Quest 2(Formerly known as the Oculus Quest 2). The headset relies on IR tracking to track the controllers. The controllers position as well as their orientation can be tracked. To get the data from the Quest headset, an APK has to be used and installed on the Quest. This APK and the Python script used to communicate with the APK are from an open-source project called *Oculus Reader*[53]. *Oculus Reader* outputs a dictionary containing a 4x4 transformation matrix that captures both translation and rotation. It's important to note that the transformation matrix reflects the relative motion between the controller and the headset. For instance, if the controller is moved 1 cm to the right while the headset is simultaneously moved 2 cm to the left, the resulting data measured by the Quest headset will indicate that the controller has moved 3 cm to the right. To avoid this, and only use the movement of the controller, the headset itself was often placed on a table, where it would experience almost no exterior disturbance.

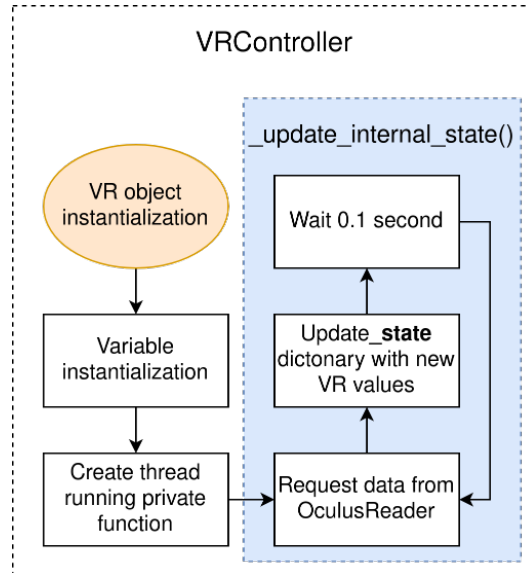


Figure 4.5: Flowchart of how the `_update_internal_state` function updates `VRController`'s internal values, for later usage. The blue box is the thread that gets started, and it will run at a frequency of 10Hz independently of other functions being called from the `VRController`

In addition to the 4X4 matrix, all of the button states are also included in the dictionary. The polling rate from the headset was set to 10 Hz, but was tested up to 30 Hz without causing any problems. However, as recommended in the Octo repository[26], 10 Hz was used for data collection. To use the data from the headset, multiple scripts were created based on the *Robot-Lightning* repository [42]. To communicate with the *Oculus Reader* library, a script called `vr.py` was created, which was originally based on the *Robot-Lightning's* `vr.py` script. The `vr.py` script contains a few functions implementing necessary linear algebra and a class called `VRController`. As the VR control was supposed to work as the policy for the environment, when recording training data, the output of the VR function would have to be the exact same shape and format as the Octo network's action head's defined output. As previously mentioned, this is a 7-dimensional vector, consisting of the end effector delta position in Euclidean space and delta rotation, using Euler angles and absolute gripper state. $[X, Y, Z, R, P, Y, G]$. The positional and rotational values are measured in cm moved since the last time step, and radians rotated since the last time step. As the gripper is an absolute value, the value 1 will correspond to having the gripper closed, with 0 being open.

To get these values, the `VRController` class is used. When `VRController` is initialized as the policy in the data-recording script, a thread is started which runs the internal function called `_update_internal_state`. This function will run at the specified frequency, which was 10 Hz for this project.

`_update_internal_state()` keeps track of the state of the VR controller. It does this by running the

`get_transformations_and_buttons()` function from an `OculusReader` object, which returns the entire state of the right VR controller, in a Python dictionary format. This dictionary is then parsed and all of the values from the dictionary is then saved as internal states in the `VRController` object. The `VRController` can then use these values in different functions. A simplified flowchart of `VRController` and `_update_internal_state()` can be seen on figure 4.5. The `predict()` function of the `VRController`, is used to get the 7D action vector in our environment, and takes position of the end effector as an argument. The `predict()` function checks whether or not the physical VR controller is activated. The VR controller will be activated if turned on and connected, and the right gripper button on the controller is pressed. If it is activated and the output from the headset is valid, it will run the function `_calculate_action()`, which takes the current position of the end effector as an argument. `_calculate_action()` is the function responsible for calculating the $[XYZ]$ vector(`pos_action`) and the RPY vector. A flowchart depicting these functions can be seen on figure 4.6.

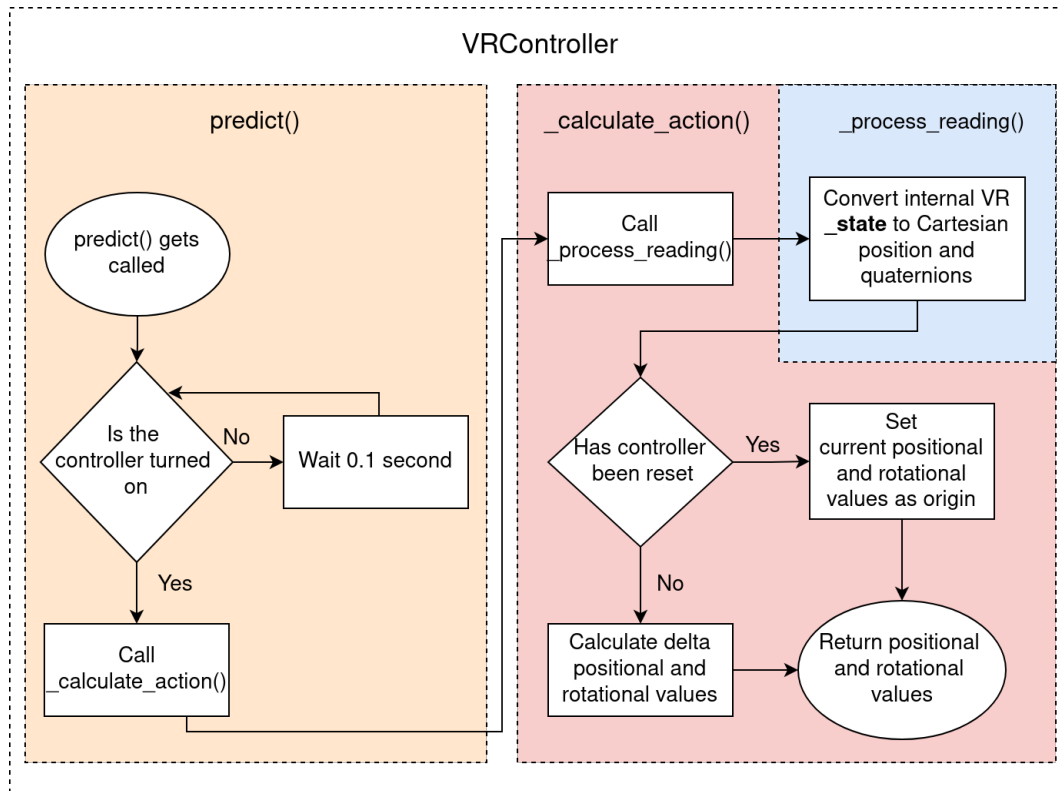


Figure 4.6: Flowchart depicting the program flow when the `predict()` function gets called. Each differently colored block is a separate function. All functions are functions in the `VRController` class. `_state` is written in bold, to highlight the connection from this flowchart to the thread running in 4.5

Calculating the positional vector

The `pos_action` vector is calculated by subtracting two vectors from each other, $target_pos_offset - hand_pos_offset$. The `target_pos_offset` vector, is a vector that describes the delta movement of the VR controller from the start of activation to the current position. Similarly the `hand_pos_offset` is a vector that describes the delta movement for the end-effector (EE) from origin position to the current EE position.

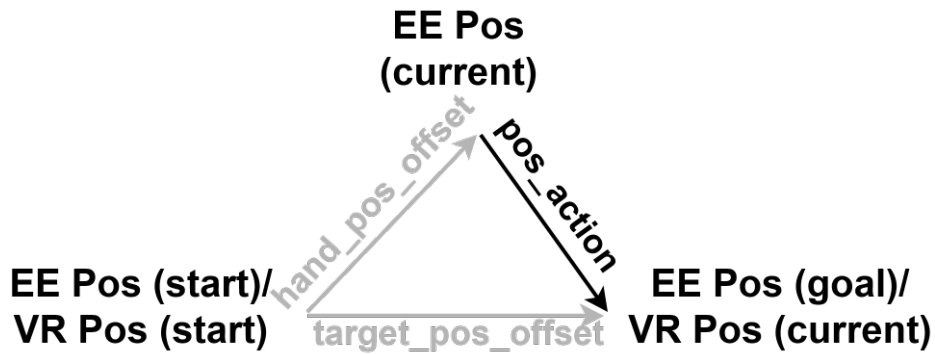


Figure 4.7: Graphics depicting how the vector `pos_action` is derived, using the "VR" vector going from VR pos(start) to VR pos(current) and the "EE" vector going from EE pos(start) to EE pos(current). Subtracting the two vectors from each other will result in the `pos_action` vector, which is the vector used for end effector control

The `pos_action` vector could also be calculated by subtracting the variables VR Pos(prev) and VR Pos(current). In a perfect system, VR Pos(prev) would in the figure 4.7 be placed in the same position as EE Pos(current). However, as drift can happen, we used the aforementioned approach. In this approach, the computation of positional offsets between the end effector (EE) and the robot's origin, as well as between the VR controller and its origin, is designed to address cumulative positional errors and potential drifts within the robotic system. By recalculating these offsets, the method aims to enhance the precision of the system's response to VR inputs.

Calculating the rotational vector

Initially, the rotational vector was calculated using the same methodology as the positional vector, using offsets for enhanced accuracy. However, this proved detrimental as it resulted in erratic behaviour from the robot, in almost every start of the tracking. Instead, the rotational vector was calculated solely by tracking the VR controller's change in rotation from previous to current step. Then that rotation was used for the robot. While this is in theory prone to rotational drift, it should not prove to much of a problem as a human operator could account for this drift when doing the recordings. In practise, we did not find this to be a perceivable problem. A human would probably also be capable of accounting for drift in the positional vector, but as the error correcting method worked, it was deemed to preferable.

After the positional and rotational values of the vector have been calculated, the gripper action is read from the field "gripper" in the internal state dictionary used

for the controller state. The "gripper" field in the dictionary is updated every 100ms in the aforementioned `_update_internal_state()` function. This value will be a float ranging from 0 to 1, and will be concatenated with the `XYZRPY` list. The value can now be returned from the `_calculate_action()`, and the `predict()` function will now in turn, return this function to the environment which uses this policy. The `predict()` function and in extension the `_calculate_action()` function is run by the environment that hosts the `VRController` object. Thus they are run independently from the `_update_internal_state()` function, which runs on its own thread, and in theory they do not have to adhere to the same frequency. If at any point, the user using the physical VR controller releases the right gripper button, `predict()` will not send the 7D vector, but instead it will return a `None` value. When the user presses the right gripper button again, the origin of the VR controller and the origin of the robot position will then be updated, to reflect the origin points at the start of the button press. This functionality enables the user to do larger ranges of motion, as they are not limited by their own physique, e.g., they can do rotations that would have otherwise required them to rotate their wrists more than 360 degrees by pressing and releasing the button multiple times. In addition to the previously mentioned functionality, the `VRController` class also have adjustable multipliers for the rotation and the translation. Additionally, it also has functionality for quickly changing the corresponding axis' from the VR to the robot, this is especially useful if the orientation of the VR headset in relation to the robot changes, as it was found that the most intuitive way of controlling the robot, is to use the robot as the reference point, from a user's point of view.

4.3 Octo

When implementing a system like Octo, into a custom environment, multiple aspects have to be taken into account. To use Octo to generate a single policy-output can be done in 4 lines in Python according to the Octo GitHub [26]. But this single "policy-output" will only be around 0.1-0.4 seconds of movement. If more outputs are required, the Octo policy has to be run multiple times, performing the action from the output before getting new outputs. However, even doing this, Octo will probably not be capable of generating any worthwhile action outputs. Octo will need to be fine-tuned to any specific environment before being capable of properly controlling a robot [26].

4.3.1 Finetuning

For the fine-tuning of Octo, a Python script was used. The fine-tuning script, was based on the example from the Octo GitHub repository, as this script already implemented all of the required features for fine-tuning. The script already had a

dataset loader implemented - though it had to be modified to support the dataset created for this project, a tokenizer for the text input, configuration for a new action head and the training function, including the loss function. As there does not seem to be any convention of how to name fields in an RLDS file, any script utilizing the data will need to have the names of the fields of the RLDS file mapped to internal values. In the finetuning script, it is done in a dictionary called `dataset_kwargs`. The dictionary, as implemented, can be seen below on the code snippet in algorithm 1. Here the name and directory of the dataset is passed, along with the names of the dataset's fields for the two cameras, and the name for the language instruction field. Additionally, the dimensionality of the action space in the dataset is also defined in the dictionary.

Algorithm 1 Pseudo code for dataset parameters

```

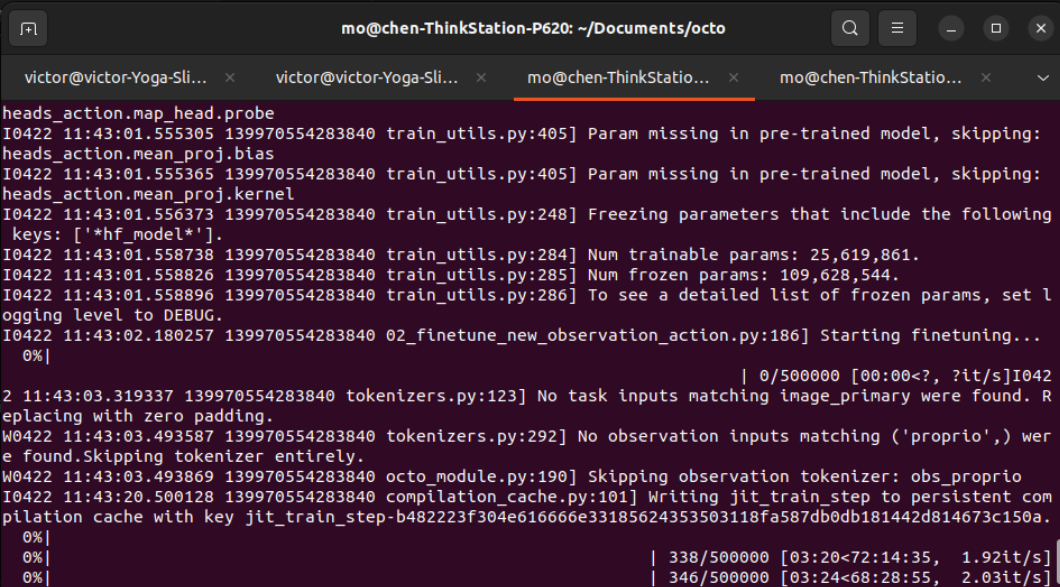
1: BEGIN dataset_kwargs
2:   name  $\leftarrow$  "robotenv"
3:   data_dir  $\leftarrow$  FLAGS.data_dir
4:   image_obs_keys  $\leftarrow$ 
5:   {
6:     "primary"  $\leftarrow$  "static_image",
7:     "wrist"  $\leftarrow$  "on_wrist_image"
8:   }
9:   language_key  $\leftarrow$  "language_instruction"
10:  absolute_action_mask  $\leftarrow$  [True, True, True, True, True, True, True]
11: END dataset_kwargs

```

Besides from changing the dictionary, few lines had to be changed to conform to the datasets created in this project. A dictionary used to define frame transformations (resizing of images), had to be extended to include the wrist camera, as the example script only used one camera. Additionally the action head also had to be changed, to have the same action dimensionality, as the example had a 14D action space. Due to the nature of the robot used for the example, the prediction horizon was also set to 50, which was changed in this project to 4, as is recommended in the GitHub issues page of Octo, by the authors, for traditional one-limb robot control. Initially the history window was set to 0, due to VRAM constraints, but was later on changed to a size of 1.

When everything in the fine-tuning script had been set up for the RobotEnv dataset, the Octo model could be fine-tuned for the first time. While it was successfully fine-tuning on the dataset, it ran significantly slower than fine-tuning on the example dataset with the 14D action space. The test dataset could be fine-tuned at a speed of around 1.5 it/s (iterations per second) and sometimes hitting slightly above 2

it/s as seen on the figure 4.8. The example dataset could be used for fine-tuning at around 9 it/s.



```

mo@chen-ThinkStation-P620: ~/Documents/octo
victor@victor-Yoga-Sli... x victor@victor-Yoga-Sli... x mo@chen-ThinkStatio... x mo@chen-ThinkStatio... x
heads_action.map_head.probe
I0422 11:43:01.555305 139970554283840 train_utils.py:405] Param missing in pre-trained model, skipping:
heads_action.mean_proj.bias
I0422 11:43:01.555365 139970554283840 train_utils.py:405] Param missing in pre-trained model, skipping:
heads_action.mean_proj.kernel
I0422 11:43:01.556373 139970554283840 train_utils.py:248] Freezing parameters that include the following
keys: ['*hf_model*'].
I0422 11:43:01.558738 139970554283840 train_utils.py:284] Num trainable params: 25,619,861.
I0422 11:43:01.558826 139970554283840 train_utils.py:285] Num frozen params: 109,628,544.
I0422 11:43:01.558896 139970554283840 train_utils.py:286] To see a detailed list of frozen params, set l
ogging level to DEBUG.
I0422 11:43:02.180257 139970554283840 02_finetune_new_observation_action.py:186] Starting finetuning...
0%|
| 0/500000 [00:00<?, ?it/s]I042
2 11:43:03.319337 139970554283840 tokenizers.py:123] No task inputs matching image_primary were found. R
eplacing with zero padding.
W0422 11:43:03.493587 139970554283840 tokenizers.py:292] No observation inputs matching ('proprio',) wer
e found.Skipping tokenizer entirely.
W0422 11:43:03.493869 139970554283840 octo_module.py:190] Skipping observation tokenizer: obs_proprio
I0422 11:43:20.500128 139970554283840 compilation_cache.py:101] Writing jit_train_step to persistent com
pilation cache with key jit_train_step-b482223f304e616666e33185624353503118fa587db0db181442d814673c150a.
0%|
0%|
0%|
| 338/500000 [03:20<72:14:35, 1.92it/s]
| 346/500000 [03:24<68:28:55, 2.03it/s]

```

Figure 4.8: First octo finetuning run on a test dataset. It is running at around 1.5-2 iterations per second, whereas the official dataset ran at around 9-10 it/s. While the training speed of the does not impact the results of the training, slower training greatly impacts the amount of iterations that can be done in the scope of this project

After looking in to the differences of our dataset and the example, it was clear that our dataset had too high resolution. This was both the images, at this time they were native resolution, and the data types, e.g., instead of using float32 like the test script, we used float64 initially. After this discovery we created functionality, as mentioned in section 4.2.6, to downscale the entire dataset. After downscaling, our dataset's training speed was on par with the example.

While fine-tuning, the progress was logged and displayed in real-time using the library called *Weights and Biases* (*Wandb*). *Wandb* is a machine learning platform that provides tools for experiment tracking, model optimization, and dataset versioning to aid the development of ML models. The *Wandb* integration, already implemented in many scripts in the Octo repo, requires only a *Wandb* account and an API key from the *Wandb* website, which you paste into the terminal upon the first boot of the training script. It is also possible to forego live tracking by disabling *Wandb* when starting the script.

Saving fine-tuning checkpoints

The fine-tuning was performed on the server, as described in section 4.3.1, and the model saved on the server needed to be transferred to the computer running inference. To do this, an FTP server was set up on the aforementioned server using the service *vsftpd*, enabling seamless file transfer from any computer on the same network, while simultaneously being password-protected. FTP transfers were selected for transferring large-sized checkpoints, each approximately 550MB, due to limitations with Git and other cloud storage options. Git itself does not support such large files, and while Git LFS (Large File Storage) can handle them, hosting challenges remain. The free version of GitHub, for example, offers only 1GB of LFS storage per user, effectively limiting uploads to one checkpoint per month. Similarly, efforts to use *Hugging Face*, another Git-based system supporting LFS, were thwarted by repeated upload failures and platform instability, as evidenced by frequent '500 - Internal Error' messages. Consequently, local storage proved to be the most reliable method for hosting the checkpoints.

4.3.2 Inference

Inference using Octo can be done using only 5 lines, as previously mentioned. However, as inference needs to be run multiple times, the inference is wrapped in a loop, with the robot environment performing the action, that is the the output of the inference, before running inference again. The Octo repository has got multiple examples on how to run inference.

Inference on dataset

One of the examples in the Octo repository, performs inference on a pre-downloaded dataset. This script unpacks the dataset and uses its data as the observation for each environment step. It then processes these steps, applying inference using the images as observations, and subsequently displays these images alongside a comparison of the ground truth and Octo's output. The ground truth is embedded within each step's action field of the dataset. Adapting this script to work with a custom dataset created for this project involved few changes of code to reference the third-person images in the dataset steps. Additionally, the dataset's path also needed to be change, to reflect the new dataset's path. Successfully running this script on the custom dataset confirmed its compatibility with Octo for fine-tuning. Additionally, the simplicity of the script was instrumental in documenting the requirements for operating Octo, providing essential information ahead of conducting online inference.

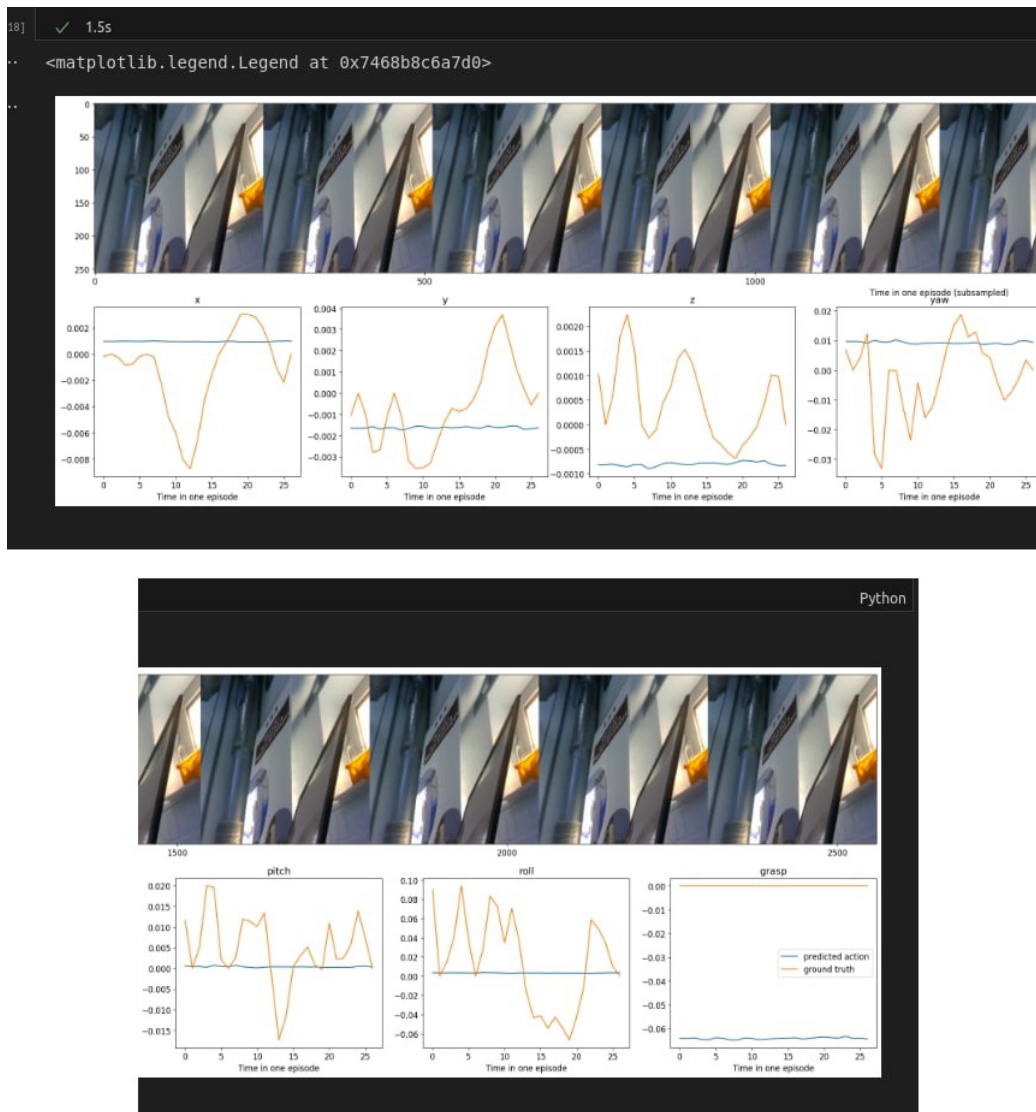


Figure 4.9: Inference performed on the pre-recorded dataset, with orange lines representing the ground truth and blue lines representing the predicted actions. The seven different graphs correspond to the actions for X, Y, Z, Roll, Pitch, Yaw, and Gripper, respectively.

Online inference

For online inference, the Octo repo has two different examples. One example for online inference is on a simulated robot in a *gym* environment. The other example is doing inference on a physical WidowX robot. The WidowX robot is a two-limbed 14 DoF robot, which makes the script used for this robot a bit different than for the Franka. Additionally, a lot of code in the example was unnecessary to run

inference on the Franka, which resulted in a lot of changes from the original. In the end, the code for running inference on the Franka ended up being around 10% less lines, while adding additional observation spaces as the example script only used one camera. The inference script ultimately comprised three main parts: flag instantiation and imports, environment setup, and the inference loop. The flag instantiation and import section consists of the first 83 lines out of 220 (at the time of writing), where all necessary libraries are imported. These include Octo models and utility scripts. Subsequently, all the flags are defined, which are later used by the model. These flags determine characteristics such as prediction horizons, history windows, and similar parameters.

After this step, the `main()` function is defined and this is where the rest of the code is written. In here, the robot environment is instantiated and the pre-trained Octo model is loaded. The robot environment is then wrapped in the History Wrapper from Octo. After this, the wrapped environment is wrapped once again in the Temporal Ensemble Wrapper from Octo. The history wrapper is a Gym wrapper from Octo, which enables the History window in the observation space. Likewise, the temporal ensemble wrapper is a Gym wrapper from Octo, which enables the prediction horizon when running inference. These wrappers enables us to use the Gym environment as usual, with `environment.step()` functions, without having to deal with multiple actions and historic observation space.

After the local functions have been defined, the loop that runs the Octo interface is initiated. This loop is a `while True` loop, meaning it will continue indefinitely until the program is closed. At the start of the loop, the program waits for user input to determine the robot's task. An `Octo Task` object is then created. The robot environment is reset to gather the initial set of observations. Subsequently, a `while` loop executes for `x` steps, where `x` represents the number of time steps Octo uses for a single task. Octo utilizes all allotted steps, as there is no other termination condition for this loop. The action is then predicted using Octo, and afterward, it is executed on the robot using the environment's `step` method. Performing the action generates a new set of observations, which are used in the next iteration of the loop. See Algorithm 2 for the pseudocode representation of the action execution based on the policy function.

When the `while` loop in the previously mentioned algorithm has finished executing, the `while True` loop containing 2 will restart. It will then await a new task defined by the user via text before re-initiating the Octo inference loop.

Algorithm 2 Procedure for executing actions based on policy function

```

1: while  $t < \text{FLAGS.num\_timesteps}$  do
2:   if  $\text{time.time()} > \text{last\_tstep} + \text{STEP\_DURATION}$  then
3:      $\text{last\_tstep} \leftarrow \text{time.time}()$ 
4:      $\text{action} \leftarrow \text{np.array}(\text{policy\_fn}(\text{obs}, \text{task}), \text{dtype}=\text{np.float64})$ 
5:      $\text{obs}, \_, \_, \text{truncated}, \_ \leftarrow \text{env.step}(\text{action}/200)$ 
6:      $t \leftarrow t + 1$ 
7:     if  $\text{truncated}$  then
8:       break
9:     end if
10:  end if
11: end while

```

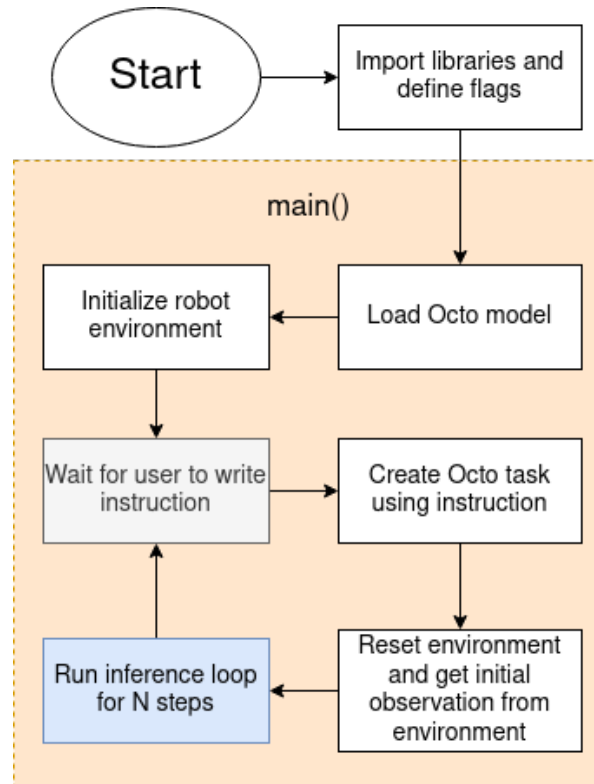


Figure 4.10: The flowchart illustrates the entire script running Octo inference. Most of the functionality is contained within the `main()` function, shown as a light beige box. The inference loop, described in Algorithm 2, is represented as a blue box within the `main()` function.

4.4 Data collection

4.4.1 Proof of Concept using Dummy Data

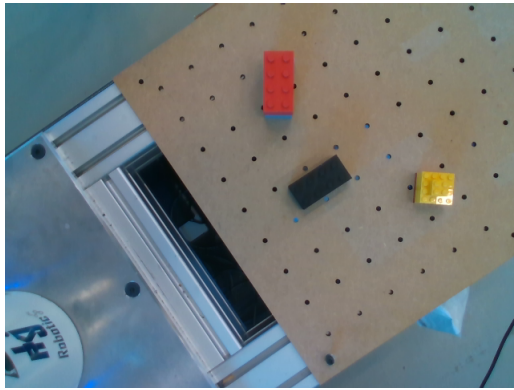
After implementing Octo and establishing the data pipelines, it was necessary to record some trajectories to collect training data for Octo. During the initial stages, 'empty' datasets lasting 3 seconds(dummy data) were recorded. These datasets did not involve any significant movements, and the cameras were not attached to the robot. However, these empty datasets proved to be sufficient for testing the data acquisition pipeline (VR + *EnvLogger*), the fine-tuning script that saved a fine-tuned model, and the offline evaluation of the dataset alongside the fine-tuned model. After the offline evaluation proved successful, the online evaluation script described in section 4.3.2 could be developed, effectively linking robot control directly to Octo's output. For the initial test, Octo was evaluated using the fine-tuned model. Since the model was trained solely on limited data from the 3-second episodes, the robot's movement was erratic. Nonetheless, this served as a successful proof of concept, demonstrating that every component of the pipeline, from data acquisition to local inference and actual robotic action based on that inference, was functioning as intended.

This proof of concept using dummy data was essential before commencing the recording of actual data. It confirmed that the system was capable of handling all necessary processes required for this project. This advancement was key in transitioning from a development stage to a data gathering and testing stage.

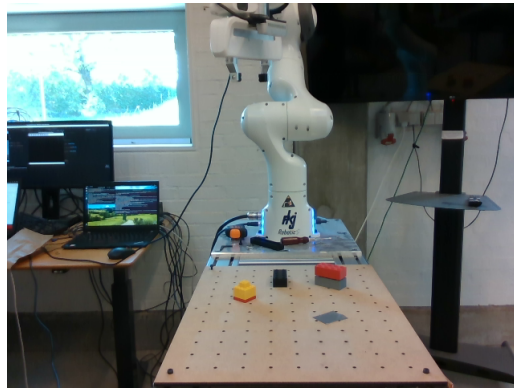
4.4.2 Recording first dataset



Figure 4.11: The setup of the Franka and the cameras. This position of the third person camera, is the position used for recording training data and running inference. Additionally the 4 differently colored bricks can be seen on front of the robot, with the red brick being stacked on top of the gray brick



(a) Image taken using the wrist camera



(b) Image taken using the static 3rd person camera

Figure 4.12: Images from the cameras used for Octo, side by side. These images show what Octo would be able to see in the setup as shown on figure 4.11

The recording of the first "proper" dataset could take place after the initial proof of concept. This first dataset was compromised of 21 episodes and recorded in 7 sessions, with 3 episodes each. As the data-recording tools did not initially have any method of discarding a particular episode in the session, only 3 episodes were recorded at each time, as to not risk invalidating too many episodes in a session by having one recording going badly. These 7 sessions all resulted in a RLDS dataset compatible with Octo. These were then merged into a single dataset using the functionality described in section 4.2.6. Each episode was a recording of the robot performing a task. For this dataset, the robot had to pick up a yellow Lego brick. These episodes were recorded using a human operator, sitting on the right side of the robot, tele-operating it using the VR controller. Each episode was around 10-20 seconds long. At the beginning of each episode the Lego brick was placed randomly on the table in front of the Robot. The third person camera was also placed in front of the robot, so as to allow the camera to observe the entire work table and most of the robot manipulator. To ensure that the third person camera would always be in the same location when recording or running inference, the position of the tripod was marked using Duct-Tape on the flooring. The position of the robot and the cameras can be seen on figure 4.11. This merged dataset was then used to fine-tune Octo. The fine-tuning of Octo was performed on the previously described server with a NVIDIA RTX A6000. It was trained for 5000 iterations, saving a checkpoint of the model for each 1000 iterations. This took around 11 minutes of the server. The training progression can be seen below on figure 5.1. The action head used for this specific model was an L1ActionHead, which can be seen described in section 3.2

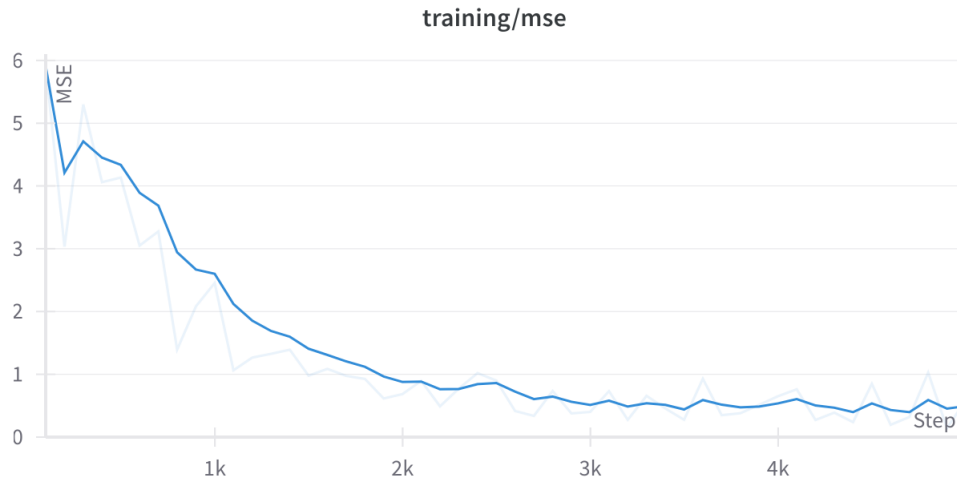


Figure 4.13: Training MSE Over Time: This graph illustrates the performance of Octo during training, showcasing the mean squared error (MSE) metric. The MSE starts at a high of around 6 and steadily decreases, indicating improved model accuracy as the training progresses to 5,000 steps. This graph was generated using WandB with a smoothing parameter of 0.7, using an exponential running average. The solid blue line represents the smoothed data, while the faded gray line shows the actual data points. Data was logged every 100 iterations

After fine-tuning Octo for 5000 iterations, the newly fine-tuned model, was then tested on the Franka robot positioned in the lab. The small dataset of 21 episodes proved to greatly improve the performance of the Octo, with the robot deliberately going in the direction of the yellow Lego brick. Even if the yellow Lego brick was moved during inference, the robot would immediately change trajectory towards the Lego brick. During some runs, the robot would go towards the general direction of the Lego brick, but go too far towards the right and go out of bounds. This would cause an error in the Polymetis server. These movements, while not doing as expected in every test-run, were still a large improvement over the model fine-tuned using the dummy data. This test using 21 episodes, were done as part of additional verification of the Octo system. When fine-tuning Octo on a new setup, such as the one in the report, one should aim to record around 50-100 episodes as per the authors of Octo [54]. Additionally, when doing advanced tasks or with a lack of training data, then the random start position of the Lego brick is also not ideal. A fixed position would be easier to fine-tune for. While 21 episodes are a fair bit below 100 episodes, the time constraints of this project made it difficult to record 100 episodes for a lot of different tasks.

While the robot's movements often did not take very long, saving the episode after completion would take at least 2 minutes, depending on the length of the episode.

These 2 minutes were completely idle from the operator's point of view, requiring the operator to watch the computer console until the message "Episode x is ready" appeared.

Additionally, the output of the VR controller was inconsistent and often mirrored when reinitializing the dataset recorder. This inconsistency caused movements that should have made the robot move in a positive direction along the X-axis to instead make it move in a negative direction. This issue also affected the Y axis but never the Z axis. When this error occurred, the headset had to be turned 180 degrees. There were also instances where the input was rotated by 90 degrees, causing movements along the X axis to be registered as movements along the Y axis. This was resolved by rotating the headset 90 degrees. These inconsistencies and necessary adjustments added to the time required for recording data, as the VR controls often needed the headset's position to be changed with each new dataset being recorded. As previously mentioned, 7 initial datasets were recorded with 3 episodes each.

After the second proof of concept using the 21 episodes, additional episodes were recorded. These episodes were still the same kind of task, with the task being a pick-up task. However, multiple bricks with different colors were introduced. When recording this additional data, each dataset contained 5 episodes, as to cut down on time we spent on re-orientating the VR headset per episode. An even mix were recorded using the new colored bricks, with separate natural language instruction, e.g., "pick up the red brick" or "pick up the gray brick". Of these new colored brick episodes, 83 new were recorded, with a dataset size of 5. However we introduced a new functionality which allowed us to discard specific episodes in a dataset when recording, this feature is also described in section 4.2.4. These new dataset were combined with the 7 initial datasets that was used in the proof of concept. In total the new combined dataset contained a little over 100 episodes with the same task, although with some variance in brick color. This single dataset containing 100 episodes were then used to fine-tune a pre-trained model. The new model was trained for 70.000 steps, which was then used for testing. Besides from recording more data, additional configurations of the model was also tried, with different action heads and history window was also introduced later on. What was the same through all fine-tunes was the model used, Octo Small. As Octo small already used around 42GB of VRAM out of 48GB available, it was unfortunately not possible to fine-tune the larger model. We later found out that the high usage of VRAM used was partially due to having a too large buffer size and doing a "full fine-tune". As mentioned in the Solution chapter, a full fine-tune means to fine-tune the transformer backbone as well as the input and output layers, instead of just the input and output layers. We decreased the buffer size between some of

the fine-tunes we did, but kept doing a "full fine-tune", as we only discovered the flag enabling it just before hand-in, after the testing.

Chapter 5

Testing

To find out whether or not we achieved the goal, that we set out for us in the problem statement each trained model that we have created was tested. For each model, the performance is described in relation to some of the key metrics of the specific model and training distribution used.

5.1 Testing configuration

When we tested the performance of the Octo system, the physical setup was the same as described in the previous chapter. We used three differently colored plastic bricks and placed the cameras in the same locations as during the data recording. We tried many different configurations of Octo fine-tunes, along with various amounts of training data. These different configurations and the corresponding amounts of training data are shown in Table 5.1.

Model ID	Action Head Type	Prediction Horizon	History Window	Iters	MSE	Model	Epi
H0	L1	4	0	0	N/a	Octo Small	0
H1	L1	4	0	5,000	0.571	Octo Small	21
H2	L1	4	0	30,000	0.648	Octo Small	104
H2X	L1	4	0	70,000	0.612	Octo Small	104
H3	Diffusion	4	1	50,000	0.563	Octo Small	104
H3X	Diffusion	4	1	100k	0.314	Octo Small	104
H4	Diffusion	4	1	100k	0.592	Octo Small	124
H5	L1	4	1	100k	0.266	Octo Small	124

Table 5.1: This table shows the different configurations of the trained models. Five models were trained, and their respective MSE can be seen at the end of training. "Iters" is short for iterations and "Epi" is short for episodes. An ID ending in X indicates that the model is the same as the one without an X, but has been trained more. All of the fine-tunes are "full fine-tunes". All fine-tunes used two images and proprioception was disabled, otherwise all values not mentioned in this table have been left at default

While the types of action heads were changed, the observation space also changed when we switched from L1 action heads to diffusion. A history window with a size of 1 was added to the observation space. This was not done in the initial tests due to issues with memory management, which we resolved when we switched to diffusion action heads.

5.2 Tests of different models

All the tests we conducted were designed to closely mimic the training data. This involved maintaining the same camera placement, using the same color of bricks, and using the same natural language commands.

5.2.1 H1 test

The first fine-tune of Octo we created was trained on only 21 episodes where the robot moved a yellow Lego brick, comprised of 8 smaller Lego bricks. This fine-

tune was never meant to actually perform the tasks but merely serve as a proof of concept. The H0 model, the pre-trained Octo model from *HuggingFace*, had already been tested. As mentioned in the papers and on GitHub by the authors, it cannot be expected to perform on a new setup without any fine-tuning. H0 moved erratically, and the behavior did not seem to respond to the task or the environment. As a proof of concept, we recorded 21 episodes and fine-tuned the model with 5,000 steps, creating the fine-tune with the ID H1.

H1 seemed to better control the Franka arm, with less erratic movements, and it often steered towards the Lego brick. It never closed the gripper, but it proved that the entire Octo pipeline was working, marking a tremendous success for what it was.

5.2.2 H2 test

H2 was the second fine-tuned model we created for this project. Since H1 was successful in its own right, it was clear that we needed more training data for better performance. According to the authors, a completely new model should have 50-100 episodes of new training data to fine-tune properly to a new environment [26]. With this knowledge, we recorded an additional 83 episodes and trained a pre-trained Octo model for 30,000 iterations. When testing this in the physical environment, the performance was significantly better than H1. The robot moved more deliberately towards the Lego brick, and since multiple colors of Lego bricks were in the training set, it could also move towards specific colors. It did manage to close the gripper, but never around a Lego brick. While closing the gripper was a sign of improvement, Octo still seemed far from the performance shown in the paper.

5.2.3 H2X test

As H2 did not achieve the envisioned performance, we trained the same model for an additional 40,000 iterations. The Mean Squared Error (MSE) decreased slightly from 0.648 to 0.612, indicating that the model had nearly stopped converging towards zero. However, the Franka robot controlled by Octo managed to grab a Lego brick once and lift it slightly before dropping it. Out of 10 tests, the robot managed to lift the brick once but approached the brick in 7 out of 10 tests.

5.2.4 H3 test

While H2X successfully grabbed a Lego brick, it struggled to do so consistently. Both H1 and H2 used an L1 action head, which is expected to have lower performance than their diffusion counterparts[26]. We then switched to a diffusion action head, which should result in better actions. Additionally, we resolved a

memory allocation issue caused by the large training-data buffer and enabled a history window to further improve performance. H3 was trained for 50,000 iterations, and the MSE was lower than H1, H2, and H2X, indicating potentially better performance. However, the model moved erratically and did not perform much better than H0.

5.2.5 H3X test

H3X is essentially the same model as H3, but trained for an additional 50,000 iterations. It did not perform any better than the base H3, despite having the lowest MSE of all models at the time. The movements remained erratic, and it did not seem to respond much to the environment.

5.2.6 H4 test

As the desired performance of the models had not yet been achieved, we theorized that the mixed colors of the bricks in the dataset might be the issue. We then recorded an additional 20 episodes of the robot lifting the black brick, resulting in a dataset with a 35/34/55 distribution of red, yellow, and black bricks, respectively. This ensured that the black brick task met the soft requirement of 50 episodes. We then trained a new model with all 124 episodes for 100,000 iterations, using a diffusion action head.

The performance did not improve over the previous diffusion head tests.

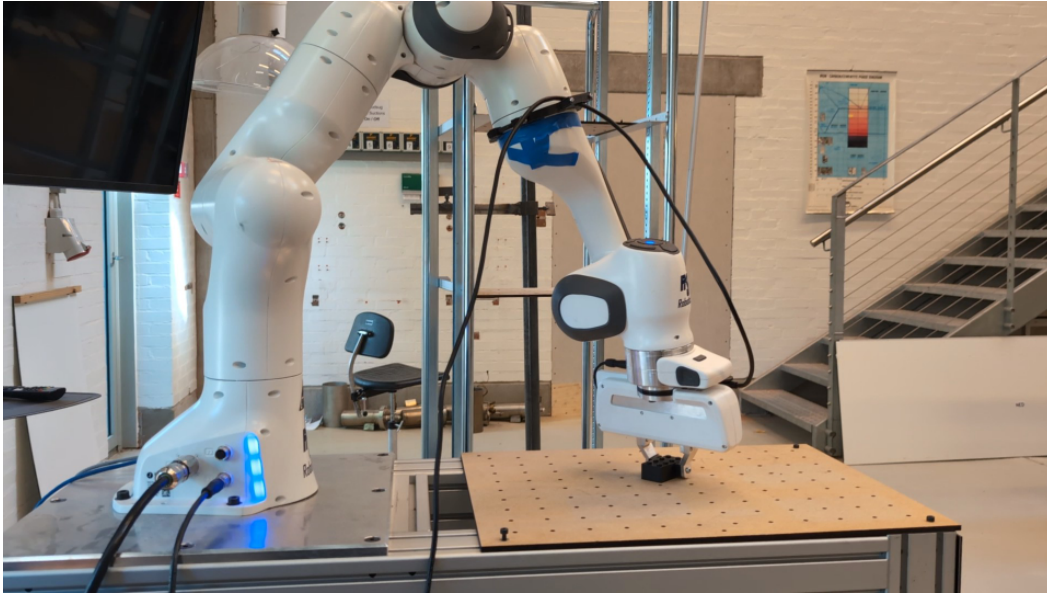


Figure 5.1: The Franka Emika robot at the end of the first trial run using the H5 fine-tune. It is in the process of grabbing the brick for a second time, as it stopped due to having used all 200 steps permissible

5.2.7 H5 test

As the diffusion heads did not seem to work for us, we trained a new model with the L1 action head. This model was the first L1-based fine-tune with a history window. Additionally, it was the most trained model of all the L1-based fine-tunes. It also used all 124 episodes and had an MSE of 0.266, the lowest of all the models.

When testing H5, only the black brick was used, and the others were not present on the table. Due to time constraints, only five test runs could be performed with this fine-tuned model. However, it managed to grab the black brick twice in the first run, lifting it up, dropping it, and then lifting it again. In the last run, it grabbed the brick again. The end-effector moved deliberately towards the black brick in 4 out of 5 runs, making it the best performing fine-tune compared to the rest of the fine-tunes we had created.

Chapter 6

Discussion

During the process of making this project we encountered several difficulties and experienced various levels of success. While implementing Octo did not seem to be too difficult at first, we quickly discovered how large of a pipeline we had to make first. This entailed everything from implementing a real time robot control system (Polymetis) to implementing a custom VR policy for data recording using a custom data recording script. While implementing the entire pipeline by ourselves provided us with great understanding and control of the entire system, the amount of time it took was greatly underestimated, leaving a very small amount of time for the actual training and testing of our Octo implementation. While we never got the Franka Emika to satisfactorily grab the bricks every time, we managed to fully fine-tune Octo and see a tremendous amount of progress in Octo's capability in grabbing. We observed a clear improvement in Octo's performance as we increased the number of episodes in the training data and training iterations. The improvement from 0 to 21 episodes, made the change from random movements to the robot clearly responding to the environment at specific test runs. Going from 21 to 104 episodes and from 5,000 iterations to 30,000, made the movements more deliberate and it responded even clearer to the environment. Using the same amount of episodes and simply training more further enhanced the performance of Octo. While the diffusion heads did not seem to work for us, adding a history window and 20 more episodes for the last fine-tuned model also proved to help tremendously, with the robot now grabbing the black brick in 2/5 runs; proving that Octo can do low-level control of a Franka manipulator using only a text command and two camera streams.

6.1 Key thoughts and findings

In this section, we describe some of our personal theories and observations regarding Octo's performance in relation to key metrics.

Amount of training data

According to the authors of Octo, fine-tuning Octo to run on a new environment requires 50-100 episodes of new training data for fine-tuning. We did see a clear improvement from going from 0 to 21 episodes and also a massive improvement going from 21 to 104 episodes. Going to 124 episodes also seemed to help a lot, but a lot of other factors also changed in this configuration. Unfortunately, due to time constraints it was not possible to do further testing of configurations to figure out what exactly made the improvement for the H5 fine-tune. Additionally, adding more training data to the diffusion head did not seem to make any difference to the performance.

Quality of training data

Once again, due to time constraints, the quality of training data may leave some parts to be desired. The system was prone to communication errors between the robot and the Polymetis system. This would often give big delays as systems would sometimes have to be restarted. Additionally, recording an episode with RLDS would take around 2-3 minutes of processing time, where the operator would have to wait for the system to finish processing. These problem points resulted in many non-optimal episodes being used for fine-tuning data, as recording more was not feasible in the time frame. This included episodes with non-optimal movements, episodes where the robot would jerk violently due to communication issues. Additionally, the VR controls also got inverted 180 degrees in the beginning of some episodes. This would sometimes result in minute movements in the wrong direction. The operator would then turn the headset 180 degrees, and continue the recording.

These problem affects maybe a bit less than 1/4 episodes. However, even the episodes with some jerky movements were for the most part sufficient except for a few steps out of up to 500 steps in each episode. We also randomized the starting position of the robot and the brick in each episode, which strengthens the variability of the training data, which should in theory make the system better at generalizing the task.

Significance of iterations and MSE

The amount of iterations seemed to make somewhat of a large difference in performance. The difference from H2 and H2X, while somewhat small, was still noticeable. When going towards the brick, the robot moved in straighter lines and it even managed to grab the brick once with H2X. H5 used 100,000 iterations, and was by far the best model. However, as previously mentioned, multiple factors are probably part of this performance. Training the models for longer also helped in lowering the MSE of the model. However, MSE did not prove to be a great indicator of the performance of the model. While H5 had the lowest MSE and best performance, the second lowest MSE was H3X's. This model performed as bad as H0 did. The second lowest MSE of the L1 based models was H1. It also archived this low MSE on only 5,000 iterations. Coupled with the fact that it only had 21 episodes for fine-tuning, over-fitting was possibly the reason behind the second lowest L1 MSE but the worst L1 performance. When more episodes were added to the mix, making the dataset more versatile, the lower MSE in H2X vs H2 did seem to fall in line with slightly better performance. H5's MSE was more than 50% lower than any other L1 based model, despite only being fine-tuned for 30,000 iterations more than H2X. At the same amount of iterations as H2X, H5 had a MSE of 3.77 which is still significantly lower. We believe that the addition of the history window and the even more versatile dataset helped it generalize the task better between the different colors.

Model configuration

The configuration of the Octo model varied slightly between H1, H3 and H5. They all used the same prediction horizon, 4, which was the recommended size. They all used Octo Small as that was the only one we could train due to constraints in VRAM. H1, H2 and H5 used a L1 action head, while H3 and H4 used diffusion. We never got diffusion to work, even though it supposedly was the best action head according to the paper. However, in the GitHub issues, people seem to be having problems with the diffusion head's performance. Models H0-H2 had 0 history window, even though it is recommended to use a history window of at least 1. This was due to VRAM constraints, introduced because the buffer size was too large. This was fixed in H3, but as previously mentioned, H3 and H4 did not seem to work. However, the history window might have had a part in the improved performance of the H5 model. Fine-tuning the Octo Small model used around 42GB of VRAM, with 48GB being available. This kept us from fine-tuning the Octo Base model, which might have performed significantly better. However, we discovered after finishing all testing, that we had performed a full fine-tune by accident every time. This means the transformer backbone of Octo was also fine-tuned. As mentioned in the Solution chapter, Octo is designed to be partially fine-

tuned where the transformer backbone's weights are frozen. This is designed this way to cut down on VRAM usage, allowing Octo Base to be trained on consumer level hardware. We did not think too much into our high usage of VRAM, as the authors on GitHub mentioned a bug causing too high VRAM consumption.

Doing a small fine-tune might have also cut down on the required amount of iterations and possibly also the amount of training data needed, as only the tokenizers and action head would be fine-tuned. Using Octo Base might also have helped reducing the amount of training data needed, as bigger transformer models are generally better at generalizing. This is why many transformer based models are in the billions of parameters, while Octo Base has 93 million and Octo small only having 27 million parameters. However, when doing a full-fine tune, a larger network may require more data, as can also be seen in the world of LLM's where the biggest models train on the most data.

Final observations

When we tested a lot of the models, the environment were also substantially different as these test were done at around 23:00 in the evening. The training data was created in the timespan from 08:00 till 17:00 usually. While segmenting the observation images should help with this problem, it might still be a cause of instability. Having more time to record data would also have allowed us to become better at using the VR controls for recording data. When reviewing the older datasets and the newer, it is clear that the operator has gotten more used to controlling the robot which results in higher quality training data as the movements are more precise. This might also contribute a lot to the performance gain in H5 vs H2.

Chapter 7

Conclusion

For this project, we set out to integrate Octo and fine-tune it on a custom dataset to help advance the field of robotics. We successfully created an entire pipeline, starting from the Polymetis server controlling a Franka Emika manipulator to running Octo inference using a model fine-tuned on our dataset for a new task.

We developed tools for recording data required for fine-tuning, including writing our own VR controller to effectively control the robot during the data collection. This data was recorded in a standardized format, making it ready for submission to the Open X Embodiment Dataset. This submission aims to support further research in robotics by addressing the current lack of standardized training data. Before submitting, we plan to enhance the data quality further by applying camera calibration to the images, as we have already performed a full calibration on both cameras, although it is not yet applied to the images.

We trained several variations of the Octo model, experimenting with different observation spaces and action heads. Our results varied, with the diffusion action heads not performing as well as expected, while the L1-based action heads showed clear success. Ultimately, we trained Octo to use two cameras and a text command to pick up a black plastic brick from random locations on a work table. This was achieved without any programming specifically related to the task, demonstrating the capability of a generalized Multi-Modal Foundation Model to perform the task with only 124 episodes of training data.

Initially we set a problem statement: *How can Octo be used to perform low-level control of a collaborative manipulator in an unknown environment?* Using this, we have managed to implement an entire pipeline of data capturing and training to allow us to successfully use Octo for low level control of a Franka Emika manipulator.

Bibliography

- [1] Rishi Bommasani et al. “On the opportunities and risks of foundation models”. In: *arXiv preprint arXiv:2108.07258* (2021).
- [2] Yafei Hu et al. “Toward general-purpose robots via foundation models: A survey and meta-analysis”. In: *arXiv preprint arXiv:2312.08782* (2023).
- [3] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706 . 03762 [cs.CL].
- [4] Fanlong Zeng et al. *Large Language Models for Robotics: A Survey*. 2023. arXiv: 2311.07226 [cs.R0].
- [5] Michael Ahn et al. *Do As I Can, Not As I Say: Grounding Language in Robotic Affordances*. 2022. arXiv: 2204.01691 [cs.R0].
- [6] Wenlong Huang et al. *Inner Monologue: Embodied Reasoning through Planning with Language Models*. 2022. arXiv: 2207.05608 [cs.R0].
- [7] Xiuye Gu et al. *Open-vocabulary Object Detection via Vision and Language Knowledge Distillation*. 2022. arXiv: 2104.13921 [cs.CV].
- [8] Aishwarya Kamath et al. *MDETR – Modulated Detection for End-to-End Multi-Modal Understanding*. 2021. arXiv: 2104.12763 [cs.CV].
- [9] Jacky Liang et al. “Code as Policies: Language Model Programs for Embodied Control”. In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*. 2023, pp. 9493–9500. DOI: 10.1109/ICRA48891.2023.10160591.
- [10] Allen Z. Ren et al. “Robots That Ask For Help: Uncertainty Alignment for Large Language Model Planners”. In: *Proceedings of the Conference on Robot Learning (CoRL)*. 2023.
- [11] Lihan Zha et al. *Distilling and Retrieving Generalizable Knowledge for Robot Manipulation via Language Corrections*. 2024. arXiv: 2311.10678 [cs.R0].
- [12] Teyun Kwon, Norman Di Palo, and Edward Johns. *Language Models as Zero-Shot Trajectory Generators*. 2023. arXiv: 2310.11604 [cs.R0].
- [13] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: 2303.08774 [cs.CL].

- [14] *GitHub - luca-medeiros/lang-segment-anything: SAM with text prompt — github.com.* <https://github.com/luca-medeiros/lang-segment-anything>. [Accessed 14-05-2024].
- [15] <https://openai.com/contributions/gpt-4v/>. [Accessed 16-05-2024].
- [16] Yingdong Hu et al. *Look Before You Leap: Unveiling the Power of GPT-4V in Robotic Vision-Language Planning*. 2023. arXiv: 2311.17842 [cs.R0].
- [17] Wenlong Huang et al. *Grounded Decoding: Guiding Text Generation with Grounded Models for Embodied Agents*. 2023. arXiv: 2303.00855 [cs.R0].
- [18] Fangchen Liu et al. *MOKA: Open-Vocabulary Robotic Manipulation through Mark-Based Visual Prompting*. 2024. arXiv: 2403.03174 [cs.R0].
- [19] Tianhe Ren et al. *Grounded SAM: Assembling Open-World Models for Diverse Visual Tasks*. 2024. arXiv: 2401.14159 [cs.CV].
- [20] Anthony Brohan et al. *RT-1: Robotics Transformer for Real-World Control at Scale*. 2023. arXiv: 2212.06817 [cs.R0].
- [21] Ethan Perez et al. “Film: Visual reasoning with a general conditioning layer”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. 1. 2018.
- [22] Mingxing Tan and Quoc Le. “Efficientnet: Rethinking model scaling for convolutional neural networks”. In: *International conference on machine learning*. PMLR. 2019, pp. 6105–6114.
- [23] Michael Ryoo et al. “Tokenlearner: Adaptive space-time tokenization for videos”. In: *Advances in neural information processing systems* 34 (2021), pp. 12786–12797.
- [24] Daniel Cer et al. “Universal sentence encoder”. In: *arXiv preprint arXiv:1803.11175* (2018).
- [25] Danny Driess et al. *PaLM-E: An Embodied Multimodal Language Model*. 2023. arXiv: 2303.03378 [cs.LG].
- [26] Octo Model Team et al. *Octo: An Open-Source Generalist Robot Policy*. <https://octo-models.github.io>. 2023.
- [27] *Home | Everyday Robots — everydayrobots.com.* <https://everydayrobots.com/>. [Accessed 20-05-2024].
- [28] Dmitry Kalashnikov et al. “Scalable deep reinforcement learning for vision-based robotic manipulation”. In: *Conference on robot learning*. PMLR. 2018, pp. 651–673.
- [29] Mostafa Dehghani et al. *Scaling Vision Transformers to 22 Billion Parameters*. 2023. arXiv: 2302.05442 [cs.CV].
- [30] Aakanksha Chowdhery et al. *PaLM: Scaling Language Modeling with Pathways*. 2022. arXiv: 2204.02311 [cs.CL].

- [31] Anthony Brohan et al. *RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control*. 2023. arXiv: 2307.15818 [cs.R0].
- [32] Abdul Rehman Abhiram Maddukuri Abhishek Gupta Ajay Mandlekar Ben Burgess-Limerick Chelsea Finn Coline Devin Deepak Pathak Fei Xia Jeanette Bohg Ken Goldberg Li Fei-Fei Pieter Abbeel Sergey Levine Trevor Darrell Wolfram Burgard Yuke Zhu et al. *Embodiment Collaboration Abby O'Neill. Open X-Embodiment: Robotic Learning Datasets and RT-X Models*. 2024. arXiv: 2310.08864 [cs.R0].
- [33] Homepage — *franka.de*. <https://franka.de/>. [Accessed 23-05-2024].
- [34] Sabela Ramos et al. *RLDS: an Ecosystem to Generate, Share and Use Datasets in Reinforcement Learning*. 2021. arXiv: 2111.02767 [cs.LG].
- [35] Alexander Khazatsky et al. "DROID: A Large-Scale In-The-Wild Robot Manipulation Dataset". In: (2024).
- [36] 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.
- [37] seann999. *GitHub Issue 43*. 2024. URL: <https://github.com/octo-models/octo/issues/43>.
- [38] Eric Hofesmann. *How to work with object detection datasets in COCO format*. <https://towardsdatascience.com/how-to-work-with-object-detection-datasets-in-coco-format-9bf4fb5848a4>. 2024.
- [39] LFranka Robotics GmbH. *Franka Control Interface Documentation*. <https://frankaemika.github.io/docs/requirements.html>. 2023.
- [40] Yixin Lin et al. *Polymetis*. <https://facebookresearch.github.io/fairo/polymetis/>. 2021.
- [41] zerorpc. *zerorpcs*. <http://www.zerorpc.io/>. 2024.
- [42] Joey Hejna "jhejna". *robot-lightning*. URL: <https://github.com/jhejna/robot-lightning>.
- [43] *GitHub - AGI-Labs/manimo: A Modular interface for robotic manipulation*. <https://github.com/AGI-Labs/manimo>. [Accessed 24-05-2024].
- [44] *Pinocchio: fast forward and inverse dynamics for poly-articulated systems*. <https://stack-of-tasks.github.io/pinocchio>.
- [45] *GitHub - dm_control: Google DeepMind Infrastructure for Physics-Based Simulation*. https://github.com/google-deepmind/dm_control. [Accessed 28-05-2024].
- [46] *SciPy* - — *scipy.org*. <https://scipy.org/>. [Accessed 31-05-2024].

- [47] *Cartesian Impedance Controller FREQUENTLY Dies*. <https://github.com/facebookresearch/fairo/issues/1190>. [Accessed 27-05-2024].
- [48] *GitHub - hengyuan-hu/monometis* — *github.com*. <https://github.com/hengyuan-hu/monometis>. [Accessed 24-05-2024].
- [49] Hanxiao Jiang et al. “RoboEXP: Action-Conditioned Scene Graph via Interactive Exploration for Robotic Manipulation”. In: *arXiv preprint arXiv:2402.15487* (2024).
- [50] Mark Towers et al. *Gymnasium*. Mar. 2023. DOI: 10.5281/zenodo.8127026. URL: <https://zenodo.org/record/8127025> (visited on 07/08/2023).
- [51] *GitHub - google-deepmind/envlogger: A tool for recording RL trajectories*. <https://github.com/google-deepmind/envlogger>. [Accessed 30-05-2024].
- [52] *GitHub - rail-berkeley/oxe_envlogger: Robot Env logger for open-x-embodiment* — *github.com*. https://github.com/rail-berkeley/oxe_envlogger. [Accessed 30-05-2024].
- [53] RAIL Berkeley. *Morphological Transformations*. URL: https://github.com/rail-berkeley/oculus_reader.
- [54] apirrone. *Random behavior after fine tuning octo on new robot 29*. <https://github.com/octo-models/octo/issues/29>. 2024.