Integration of Vision and Tactile Sensing for Robotic Insertion Tasks using Deep Reinforcement Learning

Integration von visueller und taktiler Sensortechnologie für robotergestützte Einfügeaufgaben mithilfe von Reinforcement Learning
Master thesis in the field of “Mechatronics” by Janis Lenz
Date of submission: May 26, 2025

1. Review: Prof. Jan Peters, Ph.D., Intelligent Autonomuous Systems Group
2. Review: Prof. Dr. rer. nat. Debora Clever, Institute for Mechatronic Systems
3. Review: Tim Schneider, M.Sc., Intelligent Autonomuous Systems Group
4. Review: Theo Gruner, M.Sc., Intelligent Autonomuous Systems Group
5. Review: Daniel Palenicek, M.Sc., Intelligent Autonomuous Systems Group
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

IMS

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

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

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

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

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

Darmstadt, 26. Mai 2025

J. Lenz

Abstract

Insertion tasks are a fundamental capability for robots when dealing with assembly operations, and other automation issues in an industrial environment. Humans naturally combine vision and tactile feedback to perform such tasks effectively, but integrating multiple modalities in robotics remains a significant challenge. Prioritizing and combining data from vision and touch often proves difficult, as these modalities typically produce high-dimensional and complex information that isn't easy to interpret even for humans. This thesis addresses these challenges by leveraging a Deep Reinforcement Learning algorithm called Dreamer to solve the classical peg-in-hole insertion task in a real-world environment. A modular and enhanced experimental setup, based on previous work [47], is introduced to increase the complexity of the task and evaluate Dreamer's ability to adapt to new conditions. The results demonstrate that for a hole depth of $4\mathrm{cm}$ and a tolerance of $0.5\mathrm{mm}$ between the cylinder and hole, training a policy from scratch using only visual feedback fails to solve the task. These findings underscore the critical role of tactile feedback in learning and successfully performing peg-in-hole insertion tasks with tight tolerances. Moreover, the real-world setup, developed in this work, can be reused to test other algorithms and methods to solve the task with Dreamer's results as a strong baseline for comparisons.

Kurzfassung

Das Ausführen von Einfügeaufgaben ist eine grundlegende Fähigkeit für Roboter bei Montagearbeiten und anderen Automatisierungsaufgaben in industriellen Umgebungen. Um solche Aufgaben zu lösen, kombinieren Menschen ihre visuellen und taktilen Sinne auf natürliche Weise. Die Integration mehrerer Modalitäten in der Robotik bleibt jedoch eine große Herausforderung. Die Priorisierung und Kombination von Daten aus visuellen und taktile Quellen erweist sich oft als schwierig, da diese Modalitäten typischerweise hochdimensionale und komplexe Informationen liefern, die selbst für Menschen schwer zu interpretieren sind. Diese Arbeit löst diese Herausforderungen, indem sie einen Deep Reinforcement Learning-Algorithmus namens Dreamer einsetzt, um eine klassische Einfügeaufgabe zu lösen und das in der realen Welt. Ein modularer und weiterentwickelter Versuchstand, basierend auf einer früheren Arbeiten [47], wurde aufgebaut, um die Komplexität der Aufgabe zu erhöhen und Dreamers Fähigkeit zur Anpassung an neue Bedingungen zu evaluieren. Die Ergebnisse zeigen, dass bei einer Lochtiefe von $4\mathrm{cm}$ und einer Toleranz von $0.5\mathrm{mm}$ zwischen Zylinder und Loch es nicht möglich ist, eine Policy ausschließlich mit visuellem Informationen zu trainieren und die Aufgabe zu lösen. Diese Erkenntnisse hebt die entscheidende Rolle taktiler Sensorik beim Erlernen und erfolgreichen Ausführen von Einfügeaufgaben mit engen Toleranzen hervor. Darüber hinaus kann der in dieser Arbeit entwickelte Versuchsstand genutzt werden, um andere Algorithmen und Methoden zur Lösung dieser Aufgabe zu testen. Die Ergebnisse von Dreamer können dabei als starke Vergleichsbasis dienen.

Acknowledgments

First, I want to thank my thesis advisors, Tim Schneider, Theo Gruner, and Daniel Palenicek, for giving me the opportunity to work on this fascinating topic. They always gave me great advice when needed. Additionally, we had several interesting discussions giving me new insights and impulses that I would never have thought of myself.

Next, I want to thank Prof. Dr. Jan Peters and the whole "Intelligent Autonomous Systems" group for providing the excellent infrastructure and laboratories. Without it, this work would not have been possible. I would also like to to thank Prof. Dr. Debora Clever for taking on the supervision by the Mechanical Engineering department.

Finally, I would like to thank my parents and family for their support over the years of my studies. I am very grateful that I had this opportunity.

Contents

Figures and Tables

List of Figures

List of Tables

Abbreviations and Symbols

List of Abbreviations

RL Reinforcement Learning

MBRL Model-Based Reinforcement Learning

VAE Variational Autoencoder

RSSM Recurrent State Space Model

ELBO Evidence Lower Bound

KL divergence Kullback-Leibler divergence

MDP Markov Decision Process

POMDP Partially Observable Markov Decision Process

PlaNet Deep Planning Network

TD-MPC Temporal Difference Learning for Model Predictive Control

RNN Recurrent Neural Network

SSM State Space Model

CNN Convolutional Neural Network

MLP Multi-Layered Perceptron

TD Temporal Difference

ML Machine Learning

DNN Deep Neural Network

List of Symbols

Symbol	Description
P	Probability distribution
$\mathcal{N}$	Normal distribution
$\boldsymbol{z}$	Latent or hidden state
$\boldsymbol{h}$	Deterministic latent state or recurrent state
$\boldsymbol{s}$	Stochastic (latent) state
$\boldsymbol{x}$	Input vector
$\boldsymbol{o}$	Observation vector
$\boldsymbol{a}$	Action vector
$\phi, \boldsymbol{\theta}$	Parameters of a neural network
$\boldsymbol{\mu}$	Vector of means
$\boldsymbol{\Sigma}$	Covariance matrix
ϵ	Noise variable
S	Set of (stochastic) states
A	Set of actions
r	Reward
γ	Discount factor
π	Policy
V	State value function
Q	State action value function
$\boldsymbol{p}$	Position vector

1 Introduction

Humans effortlessly combine multiple senses to solve tasks encountered in everyday life. For example, we rely on both vision and touch when plugging in a cable or inserting a key into a lock. While these tasks may seem simple to us, they present a significant challenge for machines and robots. Interpreting visual and tactile data and deriving suitable actions from it is a complex problem that requires advanced algorithms. In the following work, recent advancements in technology are utilized to address this issue, making a meaningful contribution to the ongoing research in this field.

Recent advancements in tactile sensing technology offer exciting opportunities to enhance robot performance in contact-rich tasks [17, 35, 57, 60]. High-resolution tactile sensors now provide robots with the ability to sense detailed surface interactions, such as contact forces [27], slippage [8], and surface textures [6]. This tactile feedback is invaluable for tasks like insertion [13], where visual input alone is insufficient due to occlusions or ambiguities in depth perception. However, this approach has the drawback that algorithms must be capable of processing and interpreting high-dimensional data. While visual feedback from cameras can address this challenge using computer vision techniques, interpreting tactile data adds an additional layer of complexity since even humans can't directly interpret it.

Here comes Dreamer [24] into play. Dreamer is a Model-Based Reinforcement Learning (MBRL) algorithm that learns a latent dynamics model of the environment and uses it to train a reinforcement-learning-driven policy. It shows that Reinforcement Learning (RL) is a powerful framework for training agents to solve complex tasks through interaction with their environment. By continuously learning from trial and error, RL agents develop policies that maximize cumulative rewards, making them highly effective in domains like robotics. In robotics, RL is applied to a range of applications, including locomotion [58], grasping [37], and high-precision tasks [32], enabling robots to perform tasks that traditionally require a lot of engineering effort and expert knowledge.

One often used task in robotics and industrial automation is the peg-in-hole insertion problem, a fundamental operation in assembly and manufacturing processes [53]. Peg-in-hole tasks serve as the cornerstone for many assembly procedures, such as fitting components into tight tolerances or assembling complex mechanical systems. Success in this task directly translates to improved efficiency in production lines. Given its widespread relevance, the peg-in-hole insertion task is often used as a benchmark for evaluating robotic manipulation capabilities. It involves aligning and inserting a peg of any shape into a corresponding hole, often under varying conditions and constraints. Several challenges make this task particularly demanding. Precision is crucial, as even small deviations in alignment can lead to failure. Integrating the multi-modal feedback into a cohesive framework is another significant challenge coming with this task, as the robot must interpret data from multiple inputs to guide its actions. Moreover, real-world complexities such as variability in component dimensions, misalignments, and material properties introduce further difficulty, requiring robots to adapt dynamically to new and unforeseen conditions. Addressing these challenges is not only crucial for advancing robotic manipulation but also for unlocking new possibilities in industrial automation, where reliability and adaptability are key.

Lastly, this work utilizes Shapley values [54] to evaluate the significance of the different modalities in influencing the outcomes of Dreamer's policy. Originating from cooperative game theory, Shapley values provide a method for quantifying the contribution of individual components to a final result. In this context, they are used to determine how the visual and tactile inputs contribute to the actions Dreamer produces during the task. The assumption is that the Shapley values will show that the policy relies on different modalities in specific situations throughout a successful episode, e.g. while inserting, the tactile feedback should be more relevant.

To summarize, this work makes three key contributions to the field of robotic manipulation and Reinforcement Learning:

1. **Highlighting the Importance of Tactile Sensing for Dexterous Manipulation:** This study demonstrates that tactile sensing is essential for solving complex manipulation tasks, particularly those involving tight tolerances in real-world environments. By deploying a peg-in-hole insertion task with tolerances of $0.5\mathrm{mm}$, it underscores the critical role of tactile feedback in achieving the necessary precision. This contribution reinforces the importance of tactile sensing for advancing robotic manipulation in scenarios where visual feedback alone is insufficient.

2. **Showcasing Dreamer's Performance in the Real World:** The work highlights the effectiveness of the Dreamer algorithm in solving diverse tasks in the real world with multi-modal inputs. Dreamer demonstrates impressive sample efficiency, solving tasks with relatively fewer interactions while maintaining a fixed set of hyperparameters. This underscores not only the high performance of Dreamer but also its flexibility in adapting to different scenarios, making it a valuable tool for reinforcement learning research and real-world applications.
3. **Introducing a Versatile and Flexible Testing Platform:** The work introduces a versatile platform for testing machine learning algorithms in the real world, which is based on previous work [47]. This platform enables researchers to deploy and evaluate other algorithms in a physical environment with a range of challenging configurations. The results produced by Dreamer on this platform also establish a strong baseline for future algorithms, encouraging further research and innovation in robotic manipulation and tactile integration.

2 Related Work

This chapter begins with an introduction to Dreamer and the concept of world models, followed by a section on multi-modal sensing. Next, typical peg-in-hole insertion tasks and the application of Reinforcement Learning (RL) in robotic manipulation are discussed.

2.1 Dreamer and World Models

The main machine learning framework utilized in this work is Dreamer, a Model-Based Reinforcement Learning (MBRL) approach developed by Hafner et al. [24]. Dreamer stands out for its promise of sample efficiency, meaning that it needs fewer interactions with the real environment, a critical point for experiments where hardware resources and time are limited. Therefore, it learns a world model that captures the dynamics of the task environment. By leveraging this model to predict and optimize future trajectories in a compact latent space, Dreamer reduces the need for extensive interaction with the real environment. Moreover, it is well-suited for tasks where only partial information about the environment is available.

The development of Dreamer has been an iterative process, with three key versions released to date. In the following, they are referred to as DreamerV1 [21], DreamerV2 [23], and DreamerV3 [24] respectively. Each version introduced advancements in architecture and performance, demonstrating the continuous improvement of this approach. In addition to these core versions, the framework's history includes two related papers by the same authors. The following section will introduce the papers with their core concepts in chronological order.

In the first paper, "Learning Latent Dynamics for Planning from Pixels" [22], the authors introduce the Deep Planning Network (PlaNet). PlaNet is designed specifically for planning tasks involving image-based inputs, a significant step forward in enabling reinforcement

learning agents to handle high-dimensional visual data. In this context, the authors introduce the concept of Recurrent State Space Models (RSSMs), which play a central role in learning latent dynamic models of the world. RSSMs enable the agent to operate entirely in the latent space, abstracting away the complexity of high-dimensional observations. This approach combines deterministic and stochastic transition components, ensuring the model can capture both predictable and uncertain aspects of the environment. A more detailed description of RSSMs will be provided in Section 3.2 of this work, as they constitute the world model that underpins Dreamer's planning and decision-making capabilities.

DreamerV1 [21] is the first version of the Dreamer framework to integrate a learned world model with an agent based on RL. Its world model is realized by using a RSSM from the previously described paper. A key innovation in DreamerV1 is its use of imagined rollouts generated by the world model. The agent learns its behavior solely through these imagined trajectories, that only rely on the learned world model. By using an initial real-world observation to seed the latent state, the model generates a complete trajectory within the latent space. This approach not only improves sample efficiency but also allows DreamerV1 to optimize policies in a scalable and computationally efficient manner.

DreamerV2 [23] introduces another important improvement. The stochastic component of the latent state is represented as a vector of multiple categorical variables. This discrete formulation differs from the continuous stochastic state used in both PlaNet and DreamerV1. By combining this discrete representation with deterministic transitions, DreamerV2 improves its ability to model complex dynamics as human-level performance on the Atari benchmark shows.

Next, the DayDreamer paper [58] demonstrates that the capabilities of Dreamer extend beyond simulated environments to solving complex tasks in the real world. In this work, the authors employ Dreamer to train policies on four different physical robots, showcasing its adaptability and efficiency in real-world scenarios. One of the most striking examples involves a quadruped robot, which successfully learns to perform a series of challenging tasks, including rolling off its back, standing up, and walking. Remarkably, the robot accomplishes this entirely from scratch and without resets, completing the training in just one hour of real-world interaction. A key aspect of these experiments is the consistency of the approach, every experiment is conducted using the same hyperparameters, demonstrating the robustness and generalizability of the Dreamer framework. The success of DayDreamer highlights the potential of MBRL for practical applications in robotics and beyond. Therefore, it is an important inspiration for this thesis.

The most recent iteration, DreamerV3 [24], introduces further improvements to the framework, solidifying its position as a state-of-the-art reinforcement learning algorithm. One improvement is the use of *symlog predictions* as loss functions for reconstruction and predicting rewards and other values. The challenge is that their scale varies across domains. Symlog predictions promise to make this issue manageable and have a stable training process. Among its notable achievements, DreamerV3 is the first algorithm to collect diamonds in Minecraft from scratch, accomplishing it without relying on human-provided data or predefined curricula. This breakthrough demonstrates the framework's ability to solve highly complex, long-horizon tasks with minimal external guidance, showcasing its potential for tackling challenging problems in both simulated and real-world environments.

A fascinating approach leveraging DreamerV3 is presented in Hu et al. [29]. The authors draw a motivating analogy to the process of learning to tie shoelaces. Initially, humans must look closely at the task throughout the process, but as proficiency increases, visual attention becomes unnecessary, and tying shoelaces can be done without looking. This example illustrates how additional sensory input during the learning phase can scaffold skill acquisition. Building on this idea, the authors propose that similar principles can be applied to training machine learning models. Specifically, they introduce privileged data, additional observations that are accessible only during training but not during deployment. By incorporating this privileged information, the model gains a richer understanding of the environment, enabling the learning of more robust and effective policies. The results of this approach show that policies trained with privileged data during the learning process outperform those trained only with the target data.

While Dreamer's model-based approach shows great promise, alternative methods also offer innovative solutions to RL challenges. One such approach is Temporal Difference Learning for Model Predictive Control (TD-MPC) [26], where the authors propose a method that tries to combine the strengths of model-based and model-free techniques. The core idea behind TD-MPC is to optimize local trajectories generated by a learned dynamics model over short horizons while simultaneously accounting for long-term returns using a learned terminal value function. TD-MPC also introduces a different strategy for learning the latent dynamics model. While Dreamer aims to learn a comprehensive representation of the entire environment, TD-MPC focuses on modeling only those aspects of the environment that are directly relevant for predicting the reward. This targeted approach reduces the complexity of the world model, making it more efficient and less resource-intensive than broader representations. By combining shorter rollouts with a reward-focused latent dynamics model, TD-MPC demonstrates an effective hybrid of model-based and model-free RL, offering a promising alternative to Dreamer's paradigm.

2.2 Multi-Modal Sensing in Robotic Manipulation

Successfully performing arbitrary tasks with a robot requires meaningful data to guide it and make intelligent decisions. To address the complexities of robotic tasks, many researchers have adopted multi-modal approaches, combining data from various sensing modalities to enhance performance and robustness of their solutions. Commonly used modalities include visual, tactile, force/torque, and depth information, providing complementary information for decision-making.

The first important modality in robotic manipulation is vision provided by a camera. Vision serves as a primary source of information for understanding the environment, identifying objects, and guiding actions during complex tasks. Cameras can be positioned in two primary ways within robotic setups. The first approach places the camera outside the scene, where it provides a static, global view of the environment [8, 32, 36, 52]. This setup allows for a broad perspective of the workspace, which is particularly useful for monitoring interactions between the robot and multiple objects or maintaining situational awareness. Alternatively, the camera can be mounted directly on the robot's end-effector [7, 16, 25, 30, 37], offering a localized view that focuses on the specific task being performed, and the environment near the end-effector.

In addition to standard visual data, many cameras are equipped to capture depth information, providing a 3D understanding of the environment. Depth data can significantly enhance robotic policies by enabling accurate spatial reasoning, improving object interaction, and handling complex geometries. Therefore, this modality is often used when it is available through the camera [13, 36, 38].

In recent years, the technology and availability of tactile sensors have advanced significantly, enabling more sophisticated and effective robotic manipulation. These sensors are crucial for providing feedback on physical interactions, especially when performing contact-rich tasks that require precision, such as peg-in-hole insertion [12, 59] or grasping objects [8, 25]. There are two main types of tactile sensors commonly used in robotics: vision-based and texel-based sensors.

Vision-based tactile sensors, such as the GelSight Mini [18], rely on an optical system to capture detailed surface deformations during contact. These sensors operate by using a coated gel that deforms when pressed by objects. Behind the gel, LEDs of different colors illuminate the deformation, and a camera captures the changes in the gel's surface. This setup allows estimating forces and surface textures with high resolution to algorithms,

providing rich, detailed feedback for dexterous manipulation tasks [7, 8, 12, 16, 25, 34, 46, 59].

Texel-based tactile sensors, like the uSkin sensors by XELA Robotics [49], consist of an array of distinct measuring units called texels. A texel is the tactile equivalent of a pixel in vision systems and can measure forces along different directions. Texels are capable of measuring normal forces in the surface's normal direction and shear forces in the tangent plane of the surface. Compared to vision-based tactile sensors, texel sensors help reduce the complexity of tactile feedback and improve data efficiency. Therefore, they are also commonly used in research papers [19, 30, 32, 37, 52].

Vision-based tactile sensors are increasingly favored over texel-based sensors for dexterous applications due to their ability to provide high-resolution, detailed feedback about the interaction between the sensor and the environment. Recent papers use the rich deformation data captured by vision-based tactile sensors to reconstruct both normal and shear force distributions between the sensor and the pressed object [27, 52]. By leveraging these methods, texel-based sensors can be replaced with vision-based ones making them a universal alternative.

Incorporating force and torque sensors in robotic manipulation tasks remains a common approach [7, 12, 32, 34, 36, 37], despite the potential of tactile sensors to provide richer and more informative data. For example, Dong et al. [12] show that policies trained with force and torque sensors tend to learn more efficiently, while tactile feedback, particularly from vision-based sensors results in better generalization to new scenarios. This phenomenon can be attributed to the complexity of the data provided by vision-based tactile sensors. This high-dimensional, detailed information about the interaction between the robot and its environment requires policies to first learn how to interpret this data effectively and then derive appropriate actions. In contrast, force and torque sensors provide more direct and lower-dimensional feedback, simplifying the learning process and enabling policies to converge faster. However, the work also shows, that the richer tactile data enables policies to generalize better.

Less commonly used modalities like audio [13] can also help solve robotic tasks. For example, audio input can be used to reduce noise by introducing the objective to minimize the amplitudes of the audio signal. Indeed, Feng et al. [13] show that audio has higher attention scores in some stages of the task than vision and touch when pouring water from one container to the other.

Recently, research has begun incorporating transformer architectures for robotic manipulation tasks [25]. This architecture, initially popularized by advancements in natural

language processing, gained widespread attention with the development of large language models [45]. Transformers have proven effective in processing and integrating diverse data modalities due to their ability to handle sequential and high-dimensional inputs. For example, Han et al. [25] employ a transformer model to process both visual and tactile inputs to tackle the challenge of grasping deformable objects such as fruits.

2.3 Peg-in-Hole Insertion Task in Robotics

Classical peg-in-hole insertion tasks are a cornerstone in robotic manipulation research, serving as a benchmark for testing and comparing different models. These tasks typically involve pegs of various shapes and dimensions, such as cylindrical [59], rectangular [12], or more complex geometries, paired with corresponding holes designed to match the pegs. The primary goal is straightforward: to insert the peg into the hole in a straight, smooth movement, demonstrating the model's ability to perform accurate and efficient alignment and insertion.

Jin et al. [32] use square, pentagonal, triangular, and cylindrical shapes to test their approach. This diversity in geometry introduces unique challenges for alignment and insertion, providing a robust evaluation of the proposed method. Their approach relies on multi-modal input for feature extraction, such as determining the offset between the peg and the hole. Additionally, the study explores which combinations of sensory modalities yield the best performance. By systematically examining the contributions of different inputs, the authors identify the most effective configurations for accurate and efficient peg insertion.

Lee et al. [36] also evaluate their method using pegs of various shapes, such as squares, triangles, hexagons, and semicircles, to assess its performance. Notably, they incorporate VAEs, similar to those used in Dreamer. However, rather than having the decoder reconstruct the original input data, their approach uses the decoder to predict alternative self-supervised learning objectives, such as the next end-effector position. This strategy aims to create a latent state that focuses solely on task-relevant information, omitting unnecessary details and improving task efficiency. A potential downside of this approach is that the self-supervised learning objectives must be measurable and may sometimes be incomplete, limiting the representation's ability to capture all relevant environment states.

Beyond generic insertion tasks with simple geometric shapes, more specialized scenarios closely related to real-world applications are explored. For example, Fu et al. [16] address the insertion of a USB cable, a task that requires both precise alignment and robust handling of the connectors. In the context of laboratory automation, researchers apply tactile sensing to tasks such as placing vials on racks [7] and positioning well-plates onto holders [46]. Warehouse automation presents another domain where tactile data is leveraged effectively to perform box-packing tasks. Liang et al. [37] combine vision and tactile data to grasp and push objects into the correct positions within a box. In contrast, Dong and Rodriguez [11] rely solely on tactile input to address challenges such as detecting slip for maintaining a stable grasp and estimating object positions to minimize disturbances to the surrounding environment. These real-world use cases illustrate the versatility and importance of tactile sensing in solving complex manipulation problems across diverse domains.

Another use of tactile sensing in real-world scenarios is presented by Higuera et al. [28], where the focus is on estimating extrinsic contact between objects, such as mugs, bowls, and bottles, and tactile sensors. This estimation is achieved using a specialized network architecture called Neural Contact Fields (NCF), which combines the 3D model of the object with tactile data to infer precise contact points. The detailed contact information derived from this approach is then utilized to learn robot insertion policies.

In standard insertion tasks, where a single straight movement is sufficient to achieve successful insertion, visual input alone is often adequate for solving the task. However, to highlight the importance of multi-modal feedback, Feng et al. [13] propose a more complex, multi-stage insertion task requiring rotation around the z-axis for completion. In this setup, the first stage involves inserting the peg into a larger hole, allowing for partial insertion. In the second stage, the hole perfectly matches the peg's shape but is occluded from the camera's view. Here, rotation is necessary to find the correct orientation for full insertion. With this work, the authors demonstrate that the relative importance of different sensory modalities varies across the stages of the task. While visual input is sufficient for the initial insertion, tactile feedback becomes critical in the second stage to determine the peg's orientation and achieve precise alignment.

The concept of dividing a task into distinct stages with varying sensory priorities is explored in several other studies. Feng et al. [13] propose a method that dynamically adjusts the priority of different sensory modalities based on the current stage of the task, allowing the system to leverage the most relevant information at each step. Similarly, Fu et al. [16] divides the task into two phases: first, aligning the connectors of two USB cables using tactile feedback by estimating the plug's rotation, and second, completing the

insertion using visual input for precise guidance. In Lu et al. [38], the task is also split into two stages but employs two separate neural networks. The first network handles coarse alignment, ensuring the connectors are positioned roughly correctly, while the second network performs fine insertion with greater precision. Another related approach is presented in Kamijo et al. [33], which introduces a dual-policy architecture. This system separates the alignment and insertion into distinct policies, each optimized for its respective stage of the task. All these methods highlight the effectiveness of stage-specific strategies and multi-modal sensing in tackling dexterous manipulation challenges.

Another often-used approach in the reviewed papers is to estimate the in-hand position and rotation. Gibbons, Albini, and Maiolino [19] use texel-based tactile sensors to gain this information and use it in a handcrafted search policy to find the hole which has only $0.1\mathrm{mm}$ clearance to the peg. A similar approach can be found by Sferrazza et al. [52] and Pai et al. [46].

Besides the peg-in-hole insertions tasks addressed in this section, there are other tasks where tactile feedback can be helpful. For instance, in Ichiwara et al. [30], tactile feedback is employed to tackle the task of opening a flexible bag with a zipper, demonstrating how fine-grained touch data can guide precise and adaptive actions. Similarly, Calandra et al. [8] use the combination of vision and tactile input to grasp a diverse set of 65 different objects in real-world environments. Additionally, Han et al. [25] leverage tactile feedback to grasp deformable objects such as fruits, showcasing its effectiveness in dealing with soft and fragile items where precise force control is essential.

Tactile feedback has proven to significantly enhance the performance and generalization ability of policies for solving complex robot manipulation tasks. Vision-based tactile sensors, in particular, are gaining popularity due to their ability to provide rich and detailed data. However, algorithms that rely on vision-based tactile feedback face the challenge of processing the highly complex and high-dimensional observations these sensors generate. To address this, many existing approaches focus on deriving interpretable values from the raw input data or splitting tasks into distinct stages that leverage different modalities or even separate models. While effective, these methods often result in highly application-specific solutions requiring considerable engineering effort.

This work takes a different approach by employing the Dreamer framework, which directly consumes real-world observations captured by the sensors and outputs actions, leaving the intermediate processes as a black box. This significantly reduces the engineering effort needed for task-specific tuning but raises questions about the model's decision-making process and the importance of different data modalities for its performance. To address these concerns, traditional ablation studies will be employed. Additionally, this work will

explore the use of Shapley values [54] in a later section as another approach to analyze and interpret the model's behavior, shedding light on its decision-making process.

2.4 Reinforcement Learning in Robotic Manipulation

RL has emerged as a widely used approach for solving peg-in-hole insertion tasks and other robotic manipulation challenges. Its ability to learn from interactions with the environment makes it particularly well-suited for tasks requiring adaptability and precision. Notably, Dong et al. [12] demonstrate that RL outperforms supervised learning approaches for these tasks. It is largely due to RL's capacity to explore and optimize non-greedy policies, allowing it to find more efficient and robust solutions than those derived from supervised methods, which often rely on fixed datasets and are prone to overfitting.

Various RL algorithms have been employed across different studies to tackle such tasks. Prominent methods include Proximal Policy Optimization (PPO) [32, 52], Twin Delayed Deep Deterministic Policy Gradient (TD3) [16, 34], and Soft Actor-Critic (SAC) [59]. Each of these algorithms offers unique advantages and trade-offs, making them suitable for specific problem settings and requirements. The choice of algorithm often depends on factors such as the complexity of the task, the available computational resources, and the desired balance between exploration and exploitation during training.

3 Technical Background

The following section outlines essential concepts for understanding the Dreamer framework. It begins with an introduction to Variational Autoencoders (VAEs), Recurrent State Space Models (RSSMs), and Reinforcement Learning (RL), which together provide the foundation for Dreamer's model-based approach. Next, the integration of these elements within Dreamer is described, highlighting its use of latent space planning and imagined rollouts to learn effective policies directly from high-dimensional inputs. Finally, Shapley values are introduced as a method to make arbitrary models interpretable.

3.1 Variational Autoencoders

Dreamer is specialized in dealing with high-dimensional input data, like images. Therefore it uses Variational Autoencoders (VAEs) to reduce the dimensionality of this data and map all important information into the so-called *hidden* or *latent space*.

The basic idea behind VAEs is to learn a distribution $P(\boldsymbol{x})$ over the input data x. That makes it possible to draw (or sample) new data points that are completely imagined but very similar to the original input. Therefore VAEs are called *probabilistic generative models*, and can also be used for other tasks like denoising or anomaly detection.

The explanation of VAEs in the following section is based on Prince [48, Chapter 17]. Unless explicitly stated otherwise, all statements are based on this source.

3.1.1 Nonlinear Latent Variable Models

Latent variable models are a class of probabilistic models that describe the observed data $P(\boldsymbol{x})$ indirectly by introducing a joint distribution $P(\boldsymbol{x}, \boldsymbol{z})$, where $\boldsymbol{z}$ represents the hidden or latent state. This state captures underlying structures or factors in the data that are

not directly observable. The observed probability $P(\boldsymbol{x})$ is obtained by marginalizing over the latent variable $\boldsymbol{z}$, which means integrating over all possible values of $\boldsymbol{z}$:

$$P(\boldsymbol{x}) = \int P(\boldsymbol{x}, \boldsymbol{z})\, d\boldsymbol{z}. \tag{3.1}$$

To construct $P(\boldsymbol{x}, \boldsymbol{z})$, the joint probability is expressed using the rules of conditional probability which leads to

$$P(\boldsymbol{x}) = \int P\left(\boldsymbol{x} \mid \boldsymbol{z}\right) P(\boldsymbol{z})\, d\boldsymbol{z}. \tag{3.2}$$

Here, $P(\boldsymbol{z})$ is the prior distribution over the latent variables, and $P\left(\boldsymbol{x} \mid \boldsymbol{z}\right)$ is the likelihood, representing the probability of observing $\boldsymbol{x}$ given a specific value of $\boldsymbol{z}$. This decomposition provides a structured way to model the complex data distribution $P(\boldsymbol{x})$ by leveraging the hidden structure encoded in the latent states.

In the next step, both of these distributions are specified. For a nonlinear latent variable model, the observed data $\boldsymbol{x}$ and the latent variables $\boldsymbol{z}$ are continuous and multivariate. The prior distribution over the latent variable $P(\boldsymbol{z})$ is typically chosen to be a standard multivariate normal distribution:

$$P(\boldsymbol{z}) = \mathcal{N}(\mathbf{0}, \boldsymbol{I}).$$

The likelihood $P\left(\boldsymbol{x} \mid \boldsymbol{z}, \boldsymbol{\phi}\right)$ is also modeled as a normal distribution

$$P\left(\boldsymbol{x} \mid \boldsymbol{z}, \boldsymbol{\phi}\right) = \mathcal{N}\left(\boldsymbol{x} \mid \boldsymbol{f}(\boldsymbol{z}, \boldsymbol{\phi}), \boldsymbol{\sigma}^2 \boldsymbol{I}\right),$$

where the mean is represented by a nonlinear function $\boldsymbol{f}(\boldsymbol{z}, \boldsymbol{\phi})$, and the covariance $\boldsymbol{\sigma}^2 \boldsymbol{I}$ is spherical with constant variance for each dimension. The function is realized by a Deep Neural Network (DNN) with parameters $\boldsymbol{\phi}$ that allows the model to capture complex, nonlinear relationships between the latent variables and the observed data. Next, both distributions can be plugged into Equation (3.2) to get the data probability

$$P\left(\boldsymbol{x} \mid \boldsymbol{\phi}\right) = \int \mathcal{N}\left(\boldsymbol{x} \mid \boldsymbol{f}(\boldsymbol{z}, \boldsymbol{\phi}), \boldsymbol{\sigma}^2 \boldsymbol{I}\right) \mathcal{N}\left(\boldsymbol{z} \mid \mathbf{0}, \boldsymbol{I}\right)\, d\boldsymbol{z}. \tag{3.3}$$

with a given set of parameters $\boldsymbol{\phi}$. This results in an infinite weighted sum of spherical Gaussians that is able to capture complex, nonlinear relationships between the latent variable $\boldsymbol{z}$ and the input data $\boldsymbol{x}$.

3.1.2 Training

The objective in order to train the model is to maximize the log-likelihood

$$\hat{\boldsymbol{\phi}} = \underset{\boldsymbol{\phi}}{\operatorname{argmax}} \left[\sum_{i=1}^{N} \log \left(P \left(\boldsymbol{x}_i \mid \boldsymbol{\phi} \right) \right) \right]$$

for a given training dataset $\{\boldsymbol{x}_i\}_{i=1}^{N}$ with respect to the model parameters, where $P\left(\boldsymbol{x}_i \mid \boldsymbol{\phi}\right)$ corresponds to Equation (3.3). Since the objective is generally computationally intractable, another way to solve the optimization problem must be found. The solution is to find a lower bound on the log-likelihood. Therefore, Jensen's inequality is used:

$$g(\mathbb{E}(y)) \geq \mathbb{E}(g(y)).$$

Here, g is a concave function. With this prerequisite, the inequality says that g of the expectation of a random variable y is greater or equal to the expectation of $g(y)$. Now, we can plug in the formula for the expectation of the random variable with the logarithm as the concave function g, and get

$$\log \left(\int P(y) y \, dy \right) \geq \int P(y) \log(y) \, dy. \tag{3.4}$$

Next, we rewrite the log-likelihood using Equation (3.1), and extend the equation by multiplying and dividing with an auxiliary probability distribution $q(\boldsymbol{z})$ depending on the latent variable:

$$\begin{aligned} \log(P\left(\boldsymbol{x} \mid \boldsymbol{\phi}\right)) &= \log \left(\int P\left(\boldsymbol{x}, \boldsymbol{z} \mid \boldsymbol{\phi}\right) \, d\boldsymbol{z} \right) \\ &= \log \left(\int q(\boldsymbol{z}) \frac{P\left(\boldsymbol{x}, \boldsymbol{z} \mid \boldsymbol{\phi}\right)}{q(\boldsymbol{z})} \, d\boldsymbol{z} \right). \end{aligned}$$

Finally, Jensen's inequality (Equation (3.4)) is applied to get a lower bound:

$$\log \left(\int q(\boldsymbol{z}) \frac{P\left(\boldsymbol{x}, \boldsymbol{z} \mid \boldsymbol{\phi}\right)}{q(\boldsymbol{z})} \, d\boldsymbol{z} \right) \geq \int q(\boldsymbol{z}) \log \left(\frac{P\left(\boldsymbol{x}, \boldsymbol{z} \mid \boldsymbol{\phi}\right)}{q(\boldsymbol{z})} \right) \, d\boldsymbol{z}.$$

The right-hand side of this equation is called the Evidence Lower Bound (ELBO). In practice, q is a distribution that is approximated by another DNN and therefore depends on the parameters $\boldsymbol{\theta}$ which leads to:

$$\mathrm{ELBO}(\boldsymbol{\phi}, \boldsymbol{\theta}) = \int q\left(\boldsymbol{z} \mid \boldsymbol{\theta}\right) \log \left(\frac{P\left(\boldsymbol{x}, \boldsymbol{z} \mid \boldsymbol{\phi}\right)}{q\left(\boldsymbol{z} \mid \boldsymbol{\theta}\right)} \right) \, d\boldsymbol{z}. \tag{3.5}$$

Now, Equation (3.5) can be rewritten to better understand the implications of the equation. Therefore, the numerator is separated using the rules of conditional probability. Then, the integral is split into two using the arithmetic rules of the logarithm.

$$\begin{aligned}
\mathrm{ELBO}(\boldsymbol{\phi}, \boldsymbol{\theta}) &= \int q\left(\boldsymbol{z} \mid \boldsymbol{\theta}\right) \log\left(\frac{P\left(\boldsymbol{x}, \boldsymbol{z} \mid \boldsymbol{\phi}\right)}{q\left(\boldsymbol{z} \mid \boldsymbol{\theta}\right)}\right) d\boldsymbol{z} \\
&= \int q\left(\boldsymbol{z} \mid \boldsymbol{\theta}\right) \log\left(\frac{P\left(\boldsymbol{x} \mid \boldsymbol{z}, \boldsymbol{\phi}\right) P(\boldsymbol{z})}{q\left(\boldsymbol{z} \mid \boldsymbol{\theta}\right)}\right) d\boldsymbol{z} \\
&= \int q\left(\boldsymbol{z} \mid \boldsymbol{\theta}\right) \log(P\left(\boldsymbol{x} \mid \boldsymbol{z}, \boldsymbol{\phi}\right)) \, d\boldsymbol{z} + \int q\left(\boldsymbol{z} \mid \boldsymbol{\theta}\right) \log\left(\frac{P(\boldsymbol{z})}{q\left(\boldsymbol{z} \mid \boldsymbol{\theta}\right)}\right) d\boldsymbol{z} \\
&= \int q\left(\boldsymbol{z} \mid \boldsymbol{\theta}\right) \log(P\left(\boldsymbol{x} \mid \boldsymbol{z}, \boldsymbol{\phi}\right)) \, d\boldsymbol{z} - D_{\mathrm{KL}}\left(q\left(\boldsymbol{z} \mid \boldsymbol{\theta}\right) || P(\boldsymbol{z})\right)
\end{aligned}$$

The first term, the reconstruction loss, ensures that the decoder accurately reconstructs the input data from the latent representation. The second term is the Kullback-Leibler divergence (KL divergence) which is always greater or equal to zero and measures the distance between two probability distributions. In this case, it makes sure that the approximate posterior distribution $q\left(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{\theta}\right)$ matches the prior distribution $P(\boldsymbol{z})$ as closely as possible, encouraging the latent space to follow a structured distribution. To get a simple approximation, $q\left(\boldsymbol{z} \mid \boldsymbol{\theta}\right)$ is parameterized as a multivariate normal distribution with a mean $\boldsymbol{\mu}$ and a diagonal covariance matrix $\boldsymbol{\Sigma}$, both parameterized by a deep neural network:

$$q\left(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{\theta}\right) = \mathcal{N}\left(\boldsymbol{z} \mid g_{\boldsymbol{\mu}}(\boldsymbol{x}, \boldsymbol{\theta}), g_{\boldsymbol{\Sigma}}(\boldsymbol{x}, \boldsymbol{\theta})\right).$$

Finally, we get the equation for the ELBO:

$$\mathrm{ELBO}(\boldsymbol{\phi}, \boldsymbol{\theta}) = \int q\left(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{\theta}\right) \log(P\left(\boldsymbol{x} \mid \boldsymbol{z}, \boldsymbol{\phi}\right)) \, d\boldsymbol{z} - D_{\mathrm{KL}}\left(q\left(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{\theta}\right) || P(\boldsymbol{z})\right). \tag{3.6}$$

The problem remains that the first integral is still intractable. But since it calculates an expectation, it can be replaced by using the Monte Carlo estimate:

$$\mathbb{E}_{\boldsymbol{z}}(\log(P\left(\boldsymbol{x} \mid \boldsymbol{z}, \boldsymbol{\phi}\right))) = \int \log(P\left(\boldsymbol{x} \mid \boldsymbol{z}, \boldsymbol{\phi}\right)) q\left(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{\theta}\right) \, d\boldsymbol{z} \approx \frac{1}{N} \sum_{n=1}^{N} \log(P\left(\boldsymbol{z}_n^* \mid \boldsymbol{z}, \boldsymbol{\phi}\right)).$$

Here, $\boldsymbol{z}_n^*$ is the n-th sample from $q\left(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{\theta}\right)$. We get the following approximation for the ELBO:

$$\mathrm{ELBO}(\boldsymbol{\phi}, \boldsymbol{\theta}) \approx \log(P\left(\boldsymbol{x} \mid \boldsymbol{z}^*, \boldsymbol{\phi}\right)) - D_{\mathrm{KL}}\left(q\left(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{\theta}\right) || P(\boldsymbol{z})\right). \tag{3.7}$$

In the special case with two normal distributions where one is a standard normal distribution, the second term of Equation (3.6), the KL divergence, can be calculated as

$$D_{\mathrm{KL}}\left(q\left(\boldsymbol{z} \mid \boldsymbol{x}, \theta\right) || P(\boldsymbol{z})\right) = \frac{1}{2}\left(Tr(\boldsymbol{\Sigma}) + \boldsymbol{\mu}^T\boldsymbol{\mu} - D_{\boldsymbol{z}} - \log(\det(\boldsymbol{\Sigma}))\right),$$

where $D_{\boldsymbol{z}}$ is the dimensionality of the latent space.

Now we can compute the ELBO with the following steps:

1. We take a data point $\boldsymbol{x}$ and insert it into $\boldsymbol{g}(\boldsymbol{x}, \boldsymbol{\theta})$ to get the mean and variance of the approximate posterior distribution $q\left(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{\theta}\right)$.
2. We sample a proposal latent variable $\boldsymbol{z}^*$ from this distribution.
3. Finally, we use $\boldsymbol{z}^*$ to evaluate the ELBO using Equation (3.7).

One remaining challenge in training the models is that the process involves sampling from a distribution, which introduces a stochastic component. This stochasticity prevents the computation of gradients, making it impossible to directly optimize $\boldsymbol{g}(\boldsymbol{x}, \boldsymbol{\theta})$. To address this problem, the reparameterization trick is used. It reformulates the sampling process to make it differentiable. Instead of sampling $\boldsymbol{z}$ directly from $q\left(\boldsymbol{z} \mid \boldsymbol{x}, \boldsymbol{\theta}\right)$, $\boldsymbol{z}$ is expressed as a deterministic function of a noise variable $\boldsymbol{\epsilon}$ (sampled from a standard normal distribution), and the parameters of the original distribution. In this specific case, the distribution is reparameterized as

$$\boldsymbol{z}^* = \boldsymbol{\mu} + \boldsymbol{\Sigma}^{1/2}\boldsymbol{\epsilon}, \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}).$$

This reformulation separates the stochasticity from parameters $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$, allowing gradients to flow through the deterministic parts. By enabling differentiation through the sampling process, the reparameterization trick makes it possible to optimize the model using gradient-based methods.

3.1.3 Architecture

Figure 3.1 shows the architecture of a VAE. This name is composed as follows: It is called *variational* because it approximates the posterior distribution of the latent variables with a simpler distribution, typically a Gaussian. It is called *autoencoder* because it encodes high-dimensional input data $\boldsymbol{x}$ into a lower-dimensional latent representation $\boldsymbol{z}$ and then reconstructs the original data as closely as possible from this latent vector.

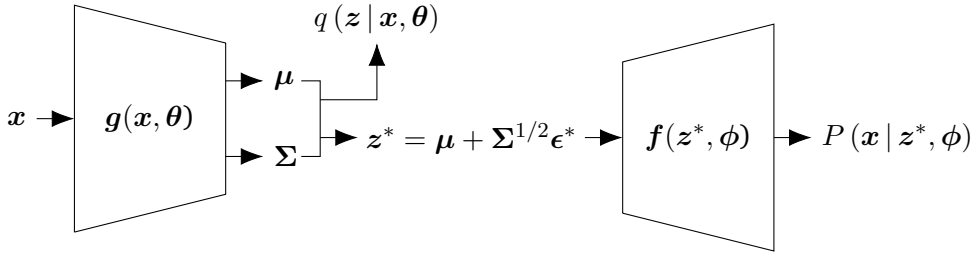
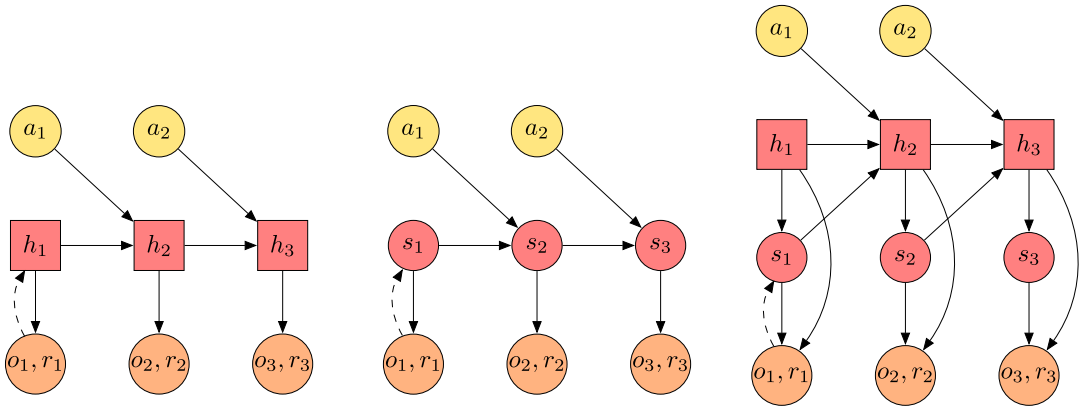


Figure 3.1: Architecture of a Variational Autoencoder (VAE): The high-dimensional input $\boldsymbol{x}$ passes through the encoder network $\boldsymbol{g}$ to generate low dimensional latent representation $\boldsymbol{z}$. The reparameterization trick is used to express $\boldsymbol{z}$ as a deterministic function depending on the noise variable ϵ. This step enables gradients to pass through the network. The original input data is then reconstructed as close as possible by passing the latent representation through the decoder network $\boldsymbol{f}$. (Figure based on Prince [48, Figure 17.9 and Figure 17.11])

Training a VAE involves optimizing the Evidence Lower Bound (ELBO), which balances the quality of reconstructions and the alignment of the latent distribution with the prior. The parameters $\boldsymbol{\phi}$ and $\boldsymbol{\theta}$ are updated iteratively using mini-batches of data and an optimization algorithm such as Stochastic Gradient Descent (SGD) or Adam.

3.2 Recurrent State Space Models

While Variational Autoencoders (VAEs) are an essential building block in mapping high-dimensional, complex data into a low-dimensional latent space, they represent just one part of Dreamer's world model. The full world model relies on a more advanced concept, the Recurrent State Space Model (RSSM), introduced in the PlaNet paper [22]. The RSSM extends the concept of VAEs from one distinct observation to a sequence of observations, creating what is referred to as a sequential VAE. It can also be interpreted as a non-linear Kalman filter, where the latent state evolves over time based on observed dynamics. A defining feature of the RSSM architecture is its division of the latent state into stochastic and deterministic components. This separation enables the model to capture both the uncertainty and consistency in the environment's dynamics, making it a critical element for successful planning and decision-making in reinforcement learning tasks.



(a) Deterministic model (b) Stochastic model (c) RSSM

Figure 3.2: Latent dynamics models: This figure shows three different designs for generative latent dynamics models. Circles indicate stochastic variables and squares deterministic variables. The solid lines denote generative processes and the dashed lines the encoding of the first real observation. The colors make it possible to distinguish between actions (yellow), latent states (red), and observations/rewards (orange). (a) Deterministic transition model using a Recurrent Neural Network (RNN). (b) The stochastic approach using a State Space Model (SSM). (c) The combination of both methods enables the model to remember information from all previous states and simultaneously incorporate uncertainties from the real environment. It is called Recurrent State Space Model (RSSM). (Figure from Hafner et al. [22] with a view adjustments)

Figure 3.2a and Figure 3.2b show the deterministic and stochastic approach for a generative dynamics model, respectively. Both capture the generative process of observations (e.g. images) and rewards, by leveraging a learned model and a first real observation to give a starting point. The key feature of the deterministic model is that it can pass information through the complete sequence of states $\{\boldsymbol{h}_t\}_{t=1}^{T}$, but can't handle uncertainties. On the other hand, the latent state-space model is fully stochastic and can handle uncertainties but loses information on previous states when dealing with longer sequences $\{\boldsymbol{s}_t\}_{t=1}^{T}$.

The RSSM architecture combines both concepts to get the best of both worlds. It is described by

$$
\begin{aligned}
&\text{Deterministic state model:} && \boldsymbol{h}_t = \boldsymbol{f}_\phi(\boldsymbol{h}_{t-1}, \boldsymbol{s}_{t-1}, \boldsymbol{a}_{t-1}) \\
&\text{Stochastic state model:} && \boldsymbol{s}_t \sim p\left(\boldsymbol{s}_t \,|\, \boldsymbol{h}_t\right) \\
&\text{Observation model:} && \boldsymbol{o}_t \sim p\left(\boldsymbol{o}_t \,|\, \boldsymbol{h}_t, \boldsymbol{s}_t\right) \\
&\text{Reward model:} && \boldsymbol{r}_t \sim p\left(\boldsymbol{r}_t \,|\, \boldsymbol{h}_t, \boldsymbol{s}_t\right),
\end{aligned}
$$

where $\boldsymbol{f}_\phi(\boldsymbol{h}_{t-1}, \boldsymbol{s}_{t-1}, \boldsymbol{a}_{t-1})$ is implemented as Recurrent Neural Network (RNN) with parameters ϕ realizing the deterministic path of the model. The RSSM is illustrated in Figure 3.2c and introduces a key distinction by splitting the latent state into a stochastic component $\boldsymbol{s}_t$ and a deterministic component $\boldsymbol{h}_t$. Together, these two states build the latent state $\boldsymbol{z}_t$. The inclusion of the deterministic part $\boldsymbol{h}_t$ is crucial for enabling the model to retain information over multiple time steps. It allows the model to deterministically access all previous states, providing a more robust mechanism for capturing long-term dependencies and improving the predictive capabilities of the generative process. The stochastic part $\boldsymbol{s}_t$ captures the inherent uncertainty and variability in the environment, which is essential for modeling complex, high-dimensional data and making robust predictions under uncertain conditions.

3.3 Reinforcement Learning

Reinforcement Learning (RL) [55] is a framework for sequential decision-making, where an agent interacts with an environment by performing actions to maximize the rewards it receives over time. The agent learns a policy (a strategy for choosing actions) that leads to the best long-term outcomes. For example, in a robotics context, a robot (agent) performs movements (actions) in the real world (environment) to complete a task, such as assembling components, and earn rewards based on its success.

RL faces several key challenges. Rewards can often be sparse. For instance, in a game of chess, the outcome (win, lose, or draw) is only determined at the end, and it is difficult to assign rewards to individual moves during the game. This difficulty is linked to the *temporal credit assignment problem*, where there may be a significant delay between a critical action and its associated reward. Additionally, environments are frequently stochastic. Sticking to the chess example, an opponent may make different moves in the same situation, introducing unpredictability. Another fundamental issue is the exploration-exploitation trade-off: the agent must balance exploring new actions to discover potentially

better strategies with exploiting known actions that yield high rewards. Addressing these challenges is central to designing effective RL algorithms. All information in this section comes from Prince [48, Chapter 19]. Unless explicitly stated otherwise, all statements are based on this source.

3.3.1 Markov Decision Process

A Markov Decision Process (MDP) is a mathematical framework used in RL to model decision-making in environments where outcomes are partly random and partly under the control of an agent. An MDP is the tuple (S, A, P, r, γ), where S is the set of states, A is the set of actions, $P\left(\boldsymbol{s}_{t+1} \mid \boldsymbol{s}_t, \boldsymbol{a}_t\right)$ is the transition probability of moving to state $\boldsymbol{s}_{t+1}$ from state $\boldsymbol{s}_t$ after taking action $\boldsymbol{a}_t$, $r(\boldsymbol{s}_t, \boldsymbol{a}_t)$ is the reward received for taking action $\boldsymbol{a}_t$ in state $\boldsymbol{s}_t$, and $\gamma \in (0, 1]$ is the discount factor, which balances immediate and future rewards. The cumulative reward starting from the state $\boldsymbol{s}_t$ is given by

$$R_t = \sum_{k=0}^{\infty} \gamma^k r(\boldsymbol{s}_k, \boldsymbol{a}_k).$$

MDPs satisfy the Markov property, meaning the future state depends only on the current state and action, not on the history of past states. The goal in an MDP is to find a policy, a mapping from states to actions, that maximizes the expected cumulative reward over time.

3.3.2 Partially Observable Markov Decision Process

A Partially Observable Markov Decision Process (POMDP) extends the MDP to scenarios where the agent does not have full access to the environment's true state. In a POMDP, the agent receives observations drawn from the observation model $\Omega\left(\boldsymbol{o}_t \mid \boldsymbol{s}_t\right)$ that provide partial information about the current state. Because the true state is hidden, the agent must maintain a belief, a probability distribution over possible states, based on past actions and observations.

In this specific case, the concept of a POMDP is crucial. Solving a peg-in-hole insertion task using visual and tactile input causes that the robot does not have full knowledge of the environment's precise state. Factors such as sensor noise, occlusions in visual data, or variability in the tactile feedback from the GelSight sensor create partial observability. By modeling the task as a POMDP, the model can maintain a belief over the possible

states of the environment, integrating uncertain observations from both visual and tactile modalities.

3.3.3 Policy

The policy is the rule that determines the agent's actions based on the current state of the environment. It defines the agent's behavior and can be either deterministic or stochastic. A deterministic policy maps a state directly to a specific action, represented as $\pi(\boldsymbol{s}) = \boldsymbol{a}$. In contrast, a stochastic policy assigns probabilities to actions for each state, represented as $\pi\left(\boldsymbol{a} \mid \boldsymbol{s}\right)$, allowing the agent to explore different actions in uncertain environments.

3.3.4 Value Functions

In reinforcement learning, evaluating the quality of states and actions under a given policy is crucial. Therefore, the state value function and the state action value function are introduced.

The state value function,

$$V^{\pi}(\boldsymbol{s}_t) = \mathbb{E}\left(R_t \mid \boldsymbol{s}_t, \pi\right),$$

represents the expected return (cumulative reward) starting from state $\boldsymbol{s}_t$ and following a policy π thereafter. It captures how good it is for the agent to be in a particular state, considering the long-term rewards achievable from that point.

The state action value function,

$$Q^{\pi}(\boldsymbol{s}, \boldsymbol{a}) = \mathbb{E}\left(R_t \mid \boldsymbol{s}_t, \boldsymbol{a}_t, \pi\right),$$

extends this concept by evaluating the expected return for taking a specific action $\boldsymbol{a}_t$ in state $\boldsymbol{s}_t$, and then following the policy π for the remaining trajectory. This function is crucial for connecting future rewards to current actions, addressing the temporal credit assignment problem by quantifying how an action at a given moment influences future rewards.

3.3.5 Policy Gradient Methods

Policy gradient methods are a class of reinforcement learning algorithms that directly learn a parameterized stochastic policy $\pi(\boldsymbol{a}_t \mid \boldsymbol{s}_t, \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ represents the trainable parameters of the policy. Instead of relying on a fixed mapping between states and actions, these methods produce a probability distribution $P(\boldsymbol{a}_t \mid \boldsymbol{s}_t)$ over the possible actions, from which actions can be sampled. The sampling step inherently introduces exploration into the agent's behavior, allowing it to discover new strategies and avoid prematurely converging to suboptimal solutions.

To formalize the optimization process, policy gradient algorithms operate over trajectories of state-action pairs $\boldsymbol{\tau} = \{\boldsymbol{s}_1, \boldsymbol{a}_1, \boldsymbol{s}_2, \boldsymbol{a}_2, ..., \boldsymbol{s}_T, \boldsymbol{a}_T\}$ generated by interacting with the environment. The goal is to maximize the expected return $r(\boldsymbol{\tau})$, which is the cumulative reward obtained across a trajectory. This is achieved by iteratively updating the policy parameters $\boldsymbol{\theta}$ to increase the probability of trajectories yielding higher rewards.

The parameter update is expressed as:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \frac{1}{I} \sum_{i=1}^{I} \frac{1}{P(\boldsymbol{\tau}_i \mid \boldsymbol{\theta})} \frac{\partial P(\boldsymbol{\tau}_i \mid \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} r(\boldsymbol{\tau}_i), \tag{3.8}$$

where α is the learning rate and I is the number of sampled trajectories or episodes. $P(\boldsymbol{\tau}_i \mid \boldsymbol{\theta})$ represents the likelihood to observe a trajectory $\boldsymbol{\tau}_i$ under the current policy.

The parameter update equation for policy gradient methods can be understood as a reward-weighted likelihood adjustment. It modifies the policy parameters $\boldsymbol{\theta}$ to increase the likelihood $P(\boldsymbol{\tau}_i \mid \boldsymbol{\theta})$ of an observed trajectory $\boldsymbol{\tau}_i$ in proportion to its reward $r(\boldsymbol{\tau}_i)$, ensuring that higher-reward trajectories have a greater influence on shaping the policy. Additionally, the probability of observing a given trajectory is normalized within the update to account for the fact that some trajectories are naturally more likely to occur, independent of their reward. This normalization prevents the policy from disproportionately prioritizing frequent but suboptimal trajectories. For trajectories that are already common under the current policy and yield high rewards, the gradient update introduces relatively small changes. This reflects the intuition that these trajectories suggest the policy is already effective in those areas of the environment. Conversely, the largest updates arise from rare trajectories that produce significant rewards. These updates encourage the policy to explore and exploit underutilized areas of the state space, facilitating the discovery of new and potentially more effective strategies.

Finally, Equation (3.8) can be rewritten and simplified to get its final form:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \cdot \frac{1}{I} \sum_{i=1}^{I} \sum_{t=1}^{T} \frac{\partial \log \left[\pi \left(\boldsymbol{a}_{it} \mid \boldsymbol{s}_{it}, \boldsymbol{\theta}\right)\right]}{\partial \boldsymbol{\theta}} \sum_{k=t}^{T} r(\boldsymbol{\tau}_{ik}), \tag{3.9}$$

where t is the time and i the episode. For a complete derivation of Equation (3.9) see Prince [48, pp. 389–391].

One challenge associated with policy gradient methods is the issue of high variance in the gradient estimates. This high variance arises because the updates rely on the stochastic sampling of trajectories, and the variability in trajectory rewards can make it difficult to achieve consistent and stable parameter updates. Consequently, many episodes are often required to obtain reliable gradient estimates, which can slow down the learning process.

A common solution to this problem is to subtract a baseline b from the trajectory returns $r(\boldsymbol{\tau}_i)$ during the parameter update. This baseline serves to reduce the variance of the gradient estimates without introducing bias. One effective choice for the baseline is to make it dependent on the current state $\boldsymbol{s}_{it}$, such as using the state value function $V^{\pi}(\boldsymbol{s}_{it})$ estimated by a neural network. The state value function provides a context-aware baseline that accounts for the inherent value of the state. By using this baseline, the policy gradient focuses on optimizing actions relative to the expected return of the current state, rather than being influenced by the absolute magnitude of the rewards.

This adjustment leads to the parameter update

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \frac{1}{I} \sum_{i=1}^{I} \sum_{t=1}^{T} \frac{\partial \log \left[\pi \left(\boldsymbol{a}_{it} \mid \boldsymbol{s}_{it}, \boldsymbol{\theta}\right)\right]}{\partial \boldsymbol{\theta}} (r(\boldsymbol{\tau}_{it}) - V^{\pi}(\boldsymbol{s}_{it})), \tag{3.10}$$

where $r(\boldsymbol{\tau}_{it})$ is the discounted return for the partial trajectory $\boldsymbol{\tau}_{it}$ that starts at time t:

$$r(\boldsymbol{\tau}_{it}) = \sum_{k=t+1}^{T} \gamma^{k-t-1} r_{ik}.$$

3.3.6 Actor-Critic Methods

Actor-critic methods fall under the category of Temporal Difference (TD) policy gradient algorithms, where the policy is updated based on feedback provided by a state value function. Instead of relying on the full sum of future rewards, actor-critic methods approximate this sum by using the observed reward from the current step and the discounted value of the next state:

$$r(\boldsymbol{\tau}_{it}) \approx r_{it} + \gamma \cdot V^{\pi}(\boldsymbol{s}_{i,t+1}, \boldsymbol{\phi}). \tag{3.11}$$

The $\boldsymbol{\phi}$ indicates that the state value is estimated by a second neural network. By applying the approximation from Equation (3.11) and substituting it into Equation (3.10), we derive the following relation:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \frac{1}{I} \sum_{i=1}^{I} \sum_{t=1}^{T} \frac{\partial \log \left[\pi \left(\boldsymbol{a}_{it} \mid \boldsymbol{s}_{it}, \boldsymbol{\theta}\right)\right]}{\partial \boldsymbol{\theta}} (r_{it} + \gamma V^{\pi}(\boldsymbol{s}_{i,t+1}, \boldsymbol{\phi}) - V^{\pi}(\boldsymbol{s}_{i,t}, \boldsymbol{\phi})). \tag{3.12}$$

This equation defines the update rule for the parameters of the policy, commonly referred to as the *actor*. The expression within the brackets of Equation (3.12) is known as the TD error. This term evaluates the consistency between the estimated state value $V^{\pi}(\boldsymbol{s}_{i,t}, \boldsymbol{\phi})$ and the estimate obtained after a single step, $r_{it} + \gamma V^{\pi}(\boldsymbol{s}_{i,t+1}, \boldsymbol{\phi})$. In theory, the TD error enables this approach to update the policy at every single step, allowing for a continuous learning process. However, in practice, this step-by-step updating is barely used. Instead, the agent typically collects a batch of experience, which is then utilized to compute updates, balancing computational efficiency with learning stability.

The parameters of the second neural network, which estimates the state value, are updated by minimizing the squared TD error:

$$L(\boldsymbol{\phi}) = \sum_{i=1}^{I} \sum_{t=1}^{T} \left(r_{it} + \gamma V^{\pi}(\boldsymbol{s}_{i,t+1}, \boldsymbol{\phi}) - V^{\pi}(\boldsymbol{s}_{i,t}, \boldsymbol{\phi})\right)^2.$$

The second network is also called the *critic* because it evaluates the quality of the actions chosen by the actor. In practice, the two networks are often realized as one network with two sets of outputs, one for the policy and one for the state values.

3.4 The Dreamer Framework

This work uses DreamerV3 [24] to learn a multi-modal policy that solves a peg-in-hole insertion task. It learns behavior by rolling out complete imagined trajectories in a compact latent space. These trajectories are then used to train an actor-critic network. Dreamer promises to work across various domains with fixed hyperparameters. This section will motivate the theoretical background by describing Dreamer's underlying idea and architecture.

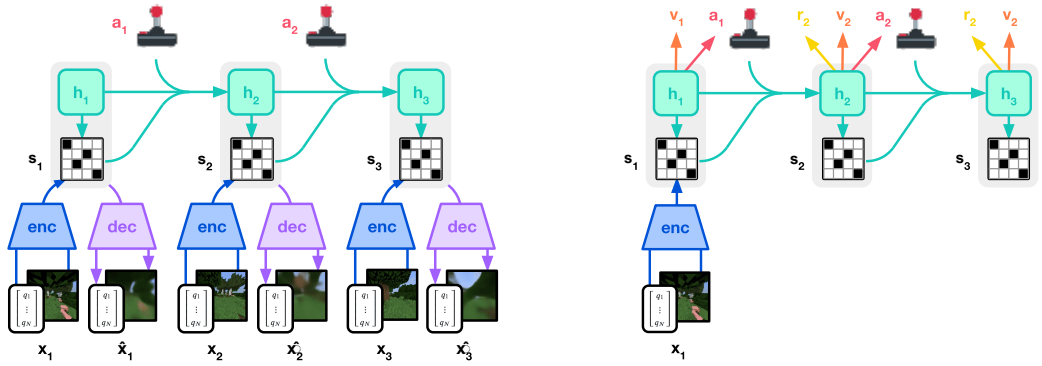
The fundamental element of Dreamer is its world model based on a RSSM described in Section 3.2 (see Figure 3.3a):

$$
\text{RSSM} \begin{cases} \text{Sequence model:} & \boldsymbol{h}_t = \boldsymbol{f}_{\boldsymbol{\phi}}(\boldsymbol{h}_{t-1}, \boldsymbol{s}_{t-1}, \boldsymbol{a}_{t-1}) \\ \text{Encoder:} & \boldsymbol{s}_t \sim (\boldsymbol{s}_t \,|\, \boldsymbol{h}_t, \boldsymbol{x}_t) \\ \text{Dynamics predictor:} & \hat{\boldsymbol{s}}_t \sim (\hat{\boldsymbol{s}}_t \,|\, \boldsymbol{h}_t) \\ \text{Reward predictor:} & \hat{r}_t \sim (\hat{r}_t \,|\, \boldsymbol{h}_t, \boldsymbol{s}_t) \\ \text{Continue predictor:} & \hat{c}_t \sim (\hat{c}_t \,|\, \boldsymbol{h}_t, \boldsymbol{s}_t) \\ \text{Decoder:} & \hat{x}_t \sim (\hat{x}_t \,|\, \boldsymbol{h}_t, \boldsymbol{s}_t) \end{cases}
$$

In the first step, the sequence model computes a recurrent or latent state $\boldsymbol{h}_t$ using the previous recurrent state $\boldsymbol{h}_{t-1}$, the prior stochastic representation $\boldsymbol{s}_{t-1}$ of the environment, and the previous action $\boldsymbol{a}_{t-1}$. The used model corresponds to the deterministic state model in Section 3.2. Next, the input $\boldsymbol{x}_t$ and the recurrent state $\boldsymbol{h}_t$ are encoded to a stochastic representation $\boldsymbol{s}_t$ capturing the current state of the environment. Together the recurrent state and the stochastic representation form the latent state $\boldsymbol{z}_t$ of the model. In comparison to the previously described stochastic part of a RSSM, this model uses a discrete representation model indicated by the checkerboard pattern in Figure 3.3.

To enable imagined rollouts, the dynamics predictor, corresponding to the stochastic state model in Section 3.2, is trained to replace the encoder by relying solely on the recurrent state. This capability allows Dreamer to simulate trajectories using only one initial real-world observation to generate complete imagined rollouts, making it possible to optimize its policy entirely within the latent state. Besides these models, Dreamer also needs a predictor for the reward and whether the trajectory should continue or not. The last model, the decoder, tries to reconstruct the original observations to ensure that the recurrent state and the stochastic representation retain all necessary information about the environment. Its equivalent in Section 3.2 is the observation model.

The encoder and decoder are realized as Convolutional Neural Networks (CNNs) for high-dimensional (visual) inputs and as Multi-Layered Perceptrons (MLPs) for low-dimensional inputs. The other predictors are also implemented as MLPs and the sequence model as RNN.



(a) World model

(b) Actor critic learning

Figure 3.3: Dreamer's world model: (a) Dreamer learns a dynamics model combining a deterministic recurrent state h and a stochastic, discrete representation s of the observations. Together, these states build the latent state z. (b) With the learned dynamics model, Dreamer is able to to simulate complete trajectories needing only one real observation to start the process. It uses these imagined trajectories to optimize its actor-critic network. (Figure from Hafner et al. [24])

Once the world model is learned, Dreamer leverages it to simulate rollouts entirely in the latent space, requiring only one initial real-world observation to begin the process. These imagined trajectories enable Dreamer to optimize its actor-critic network by iteratively refining the policy and the state value estimation based on simulated interactions. The updated policy is then deployed in the real world to collect new data, potentially achieving higher rewards and providing new insights into the environment. This new data is subsequently incorporated to enhance the world model, creating a self-improving cycle that integrates learning, simulation, and real-world interaction.

3.5 Shapley Value Analysis

Shapley values [54], originating from cooperative game theory, provide a method to fairly distribute the value generated by a group of contributors based on their individual contributions. To illustrate, imagine three persons collaborating to develop a groundbreaking new technology and founding a company valued at €1 million. The question arises: how should the value, or company shares, be divided among them based on their contributions?

At first glance, an equal split might seem fair, as it is challenging to determine precisely how much each person contributed to the company's success. However, if time travel were possible, you could evaluate the value of the company under various scenarios: if only two of the three individuals collaborated, or even if only one worked on the project. These hypothetical evaluations could reveal whether certain contributors were more or less crucial to the final outcome. Shapley values formalize this approach by calculating the value of all possible subsets of contributors and determining each individual's exact contribution to the final value.

While this evaluation might be infeasible in real-world scenarios like the example above, the concept can be effectively applied to machine learning. In this context, Shapley values are used to evaluate the importance of specific features or inputs of a model. By repeatedly testing the model's performance with and without certain features, Shapley values quantify the unique contribution of each feature to the model's overall performance. This ability makes Shapley values a powerful tool for interpreting and understanding machine learning algorithms.

This idea is formalized by the equation

$$\Phi_i(\boldsymbol{x}) = |\mathcal{F}|^{-1} \sum_{\mathcal{S} \subset \mathcal{F} \setminus \{i\}} \begin{pmatrix} |\mathcal{F}| - 1 \\ |\mathcal{S}| \end{pmatrix} \left(f(\boldsymbol{x}_{\mathcal{S} \cup \{i\}}) - f(\boldsymbol{x}) \right) \tag{3.13}$$

to calculate the expected contribution or the Shapley value $\Phi_i(\boldsymbol{x})$ of feature i. $\mathcal{F}$ is the set of all features and $\mathcal{S} \subset \mathcal{F} \setminus \{i\}$ is the subset of features excluding feature i. The absolute value of a set is considered the total number of features inside this set. The last term of Equation (3.13) measures the marginal contribution of feature i. It compares the output of f when feature i is added to subset $\mathcal{S}$ versus when it is excluded.

Shapley values, while a powerful tool for understanding feature contributions, come with a significant limitation: their computational cost. As the number of features N increases, the number of possible feature combinations grows exponentially, scaling with 2^N. This

rapid increase in complexity makes the calculation of Shapley values computationally expensive, especially for models with a large number of features.

Despite this limitation, the Shapley value framework is sufficient for analyzing the model in this work due to the manageable number of features under consideration. However, for cases where the number of features becomes prohibitively large, alternative methods such as SHapley Additive exPlanations (SHAP) [41] can be utilized. SHAP offers an efficient approximation of Shapley values, making it suitable for high-dimensional models while retaining the interpretability benefits of the original method.

4 Methodology

This chapter provides an overview of the hardware setup and offers a brief explanation of the software implementation. Following that, the integration of Shapley value analysis into Dreamer is discussed.

4.1 Experimental Setup

The experimental setup, illustrated in Figure 4.1, is based on the work of Palenicek et al. [47]. The primary objective is to insert the cylinder into the hole located at the center of the base plate. During each episode, the cylinder is held by specialized grippers that can mount GelSight mini sensors [18], positioned on the left and right sides of the cylinder.

To address scenarios where the cylinder rotates inside the gripper or is dropped, a self-resetting mechanism is incorporated, which was first introduced by Palenicek et al. [47]. A thin thread attaches the cylinder to the end-effector, allowing it to hang down when the gripper is opened. This setup enables the robot to reposition the cylinder consistently by moving with the open gripper above the reset box, lowering it to let the cylinder settle into a predefined position, and regrasping it in the same position and orientation every time. This automated reset mechanism ensures uninterrupted operation, enabling the robot to practice the insertion task continuously over extended periods without requiring human intervention.

In this work, the original design from Palenicek et al. [47] has been enhanced with two significant upgrades to improve the flexibility and complexity of the experimental setup. The first upgrade introduces a modular base plate. This allows the hole's diameter and depth to be easily adjusted by 3D printing different hole inserts and replacing them within the base plate. The second upgrade adds the ability for the hole to rotate around its vertical axis. This adjustment makes it possible to randomize the orientation of the hole,

particularly when the hole does not go straight down but has a specific angle. Before diving into the details of these two adjustments, the 3D printing process and the hardware are first described.

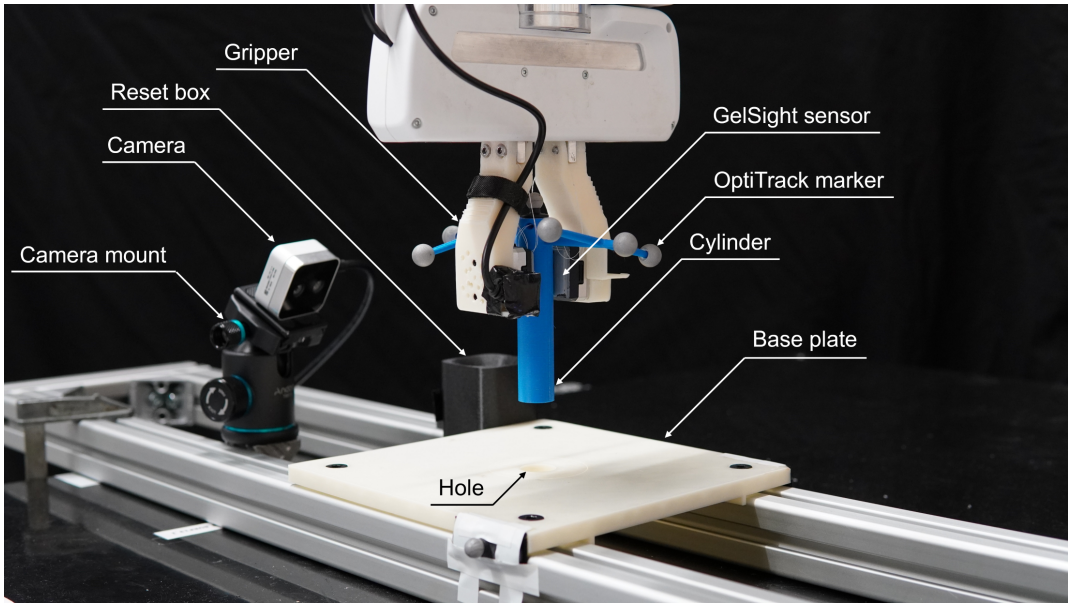


Figure 4.1: Experimental setup: The experimental setup features specialized grippers designed to mount the GelSight sensors, which securely hold the cylinder between them. To prevent the cylinder from being dropped completely, a thin thread is attached to its upper end, allowing it to hang down from the end-effector when the gripper is open. For consistent repositioning, the robot regrasps the cylinder in a predefined position by moving with an open gripper above a reset box, lowering down, and grasping the cylinder again. The objective of the experiment is to insert the cylinder into the barely visible hole located in the center of the base plate. Additionally, the camera films the scene, providing important observations (besides the GelSight images) to solve the task.

4.1.1 3D printing

An essential aspect of constructing the experimental setup is the use of 3D printing technology, which provides the flexibility to rapidly prototype and adjust components as

needed. Several parts of the setup are 3D printed, including the gripper for mounting the GelSight sensors, the base plate, modular holes with varying diameters, the reset box, and the cylinder equipped with arms to hold OptiTrack markers.

The 3D printing process was carried out using the Prusa i3 MK3S [1], a reliable and versatile 3D printer well-suited for creating high-precision parts. For 3D modeling, Autodesk Fusion [31] was used to design the components, while the PrusaSlicer [2] was deployed to convert the 3D models into G-code files, which are instructions for the printer.

The advantages of 3D printing were particularly evident in this project. It enabled rapid prototyping, allowing for iterative design and testing. Additionally, it facilitated quick adjustments to the experimental setup without waiting for new parts to be manufactured and delivered.

4.1.2 Hardware

The central part of the experimental setup is the Franka Research 3 robot [20], a state-of-the-art robotic arm with 7 degrees of freedom, which provides high precision and flexibility for manipulation tasks. This robot can be controlled via the Frank Control Interface (FCI) at a frequency of $1\,\mathrm{kHz}$, enabling real-time control and communication for complex tasks.

Depending on the observation space configuration, the hardware is capable of incorporating tactile feedback, visual feedback, and the end-effector position. The robot's internal measurement system is used to track the end-effector's position. Tactile sensing is provided by the GelSight Mini [18], a high-resolution sensor capable of capturing detailed surface contact information. Visual feedback is provided by an Intel RealSense camera [10], which can offer depth and color images for additional context during the task.

OptiTrack [44] is used for precise tracking of objects in the environment, providing critical spatial information for the experiment. Specifically, OptiTrack is used to monitor the position of the base, which is necessary for calculating the location of the hole and the reset box. Additionally, OptiTrack tracks the position and orientation of the cylinder, allowing for accurate reward calculation and ensuring that the cylinder remains within a defined workspace. The tracking also helps to limit the angle of the cylinder relative to the end-effector's orientation, which is crucial for avoiding damage to the cylinder and other parts of the setup.

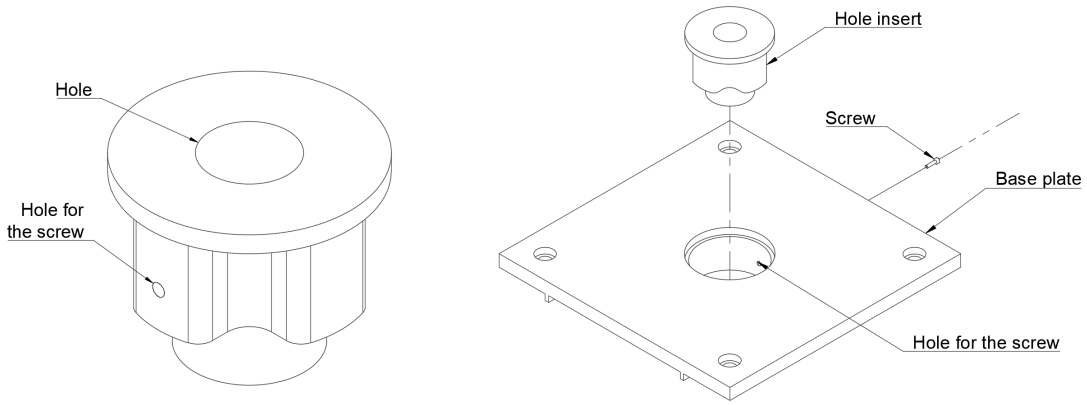
4.1.3 Original setup

The original experimental setup from the previous work [47] had some significant limitations. The hole diameter was fixed at $23\,\mathrm{mm}$ and the hole depth was $17\,\mathrm{mm}$, which restricted the range of possible experiments. But in the beginning, it was used to perform a proof of concept, since visual data was never utilized in this setup before.

Initially, the camera was mounted directly onto the end-effector to provide highly dynamic visual feedback during the task. However, this setup presented a number of challenges. The camera often covered the OptiTrack markers, making it difficult to track the position of the cylinder in the environment. Furthermore, the movement of the camera during the task caused problems with the cables, which would often become wiggled or disconnected, leading to a termination of the training process. To resolve these issues, the camera was repositioned to a fixed perspective, overlooking the entire scene.

4.1.4 Upgrade 1

In the first iteration of improving the experimental setup (Figure 4.2), the system was made modular to increase its flexibility. The base plate was redesigned to include a $5\,\mathrm{cm}$ hole at its center, which allows for quick and easy replacement of the hole insert (see Figure 4.2a). This modular design enables the system to accommodate holes with varying diameters and depths by simply swapping out the insert. The hole inserts are securely attached to the base plate using a screw, ensuring that the robot cannot accidentally pull them out during operation. This robust attachment method still allows rapid adjustments. With the modular approach, printing a new hole insert takes only 2-3 hours, a significant improvement compared to the more than 10 hours required to print a completely new base plate. This not only reduces the time required for deploying new hole configurations but also minimizes material usage, making the setup more resource-efficient and adaptable to different experimental requirements.



(a) Hole insert (b) Exploded view

Figure 4.2: Drawing of upgrade 1: This modular upgrade to the setup allows to quickly change the parameters of the hole by simply replacing the insert in the center of the base plate. The insert is secured by a screw, avoiding that the robot accidentally pulls it out. The lines of long and short dashes resemble symmetry lines. (a) shows the design for the insert with a hole for the screw. (b) shows the exploded view of the setup providing a visualization of how it is assembled.

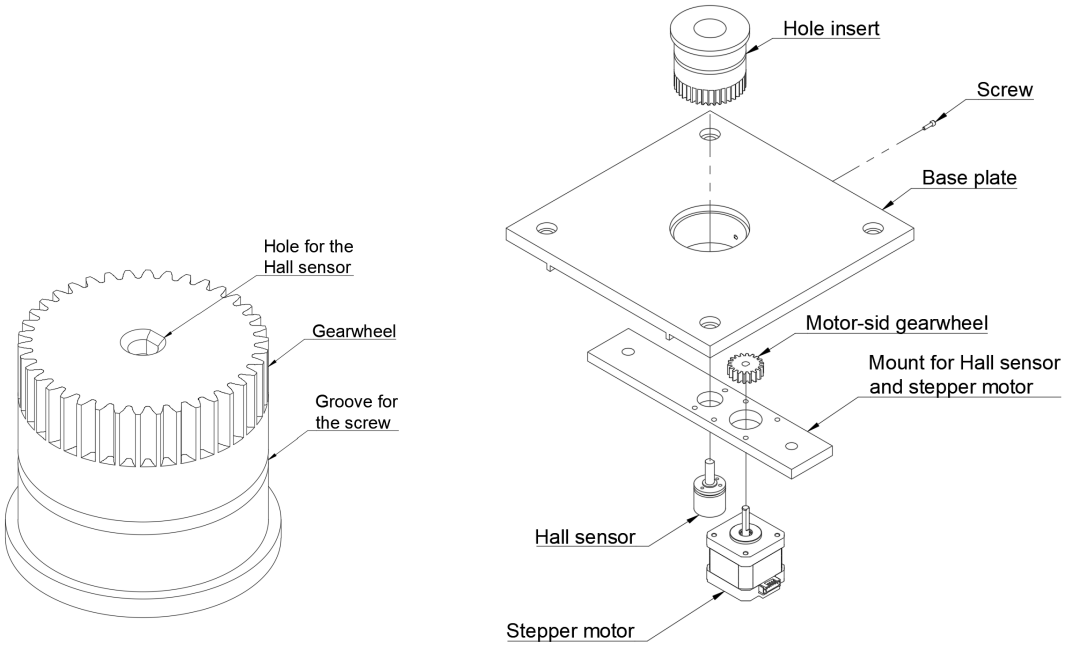
4.1.5 Upgrade 2

The second upgrade of the setup, shown in Figure 4.3, builds upon the modular design established in the first iteration, leveraging the rotational symmetry of the hole inserts. Each hole insert (Figure 4.3a) is now equipped with a gear at its bottom, enabling controlled rotation. To prevent any vertical displacement during operation, the inserts are again secured by a screw that engages with a rotationally symmetrical groove on the insert. A smaller motor-side gearwheel, attached to a stepper motor, drives the rotation of the hole insert. This precise motion is monitored by a Hall sensor, which is directly linked to the hole insert and provides accurate information about its rotation angle. Both the Hall sensor and the stepper motor are mounted securely on a dedicated frame that holds the components in place, ensuring stability and reliable operation.

The stepper motor is controlled, and the Hall sensor outputs are processed, by an Arduino Uno [4]. This microcontroller acts as the interface for managing the rotation mechanism of

the hole insert. Commands can be sent to the Arduino via serial communication, allowing precise control over the stepper motor. Specifically, the Arduino can receive instructions to rotate the stepper motor by a specified number of steps, either in the positive or negative direction. Additionally, it provides feedback on the current angle of the hole insert by measuring the output voltage of the Hall sensor.

This experimental setup was specifically developed to increase the complexity of the task, pushing the boundaries of what the model can learn and accomplish. One significant enhancement is that the holes can now be oriented at angles of up to 15°, introducing an additional challenge for the model. Therefore, the end-effector is now capable of adjusting its orientation, adding another layer of difficulty to the insertion task. With these adjustments, the hope is that Dreamer will need to rely more heavily on its sense of touch. While the camera can provide a clear view of the hole's position, it cannot determine the hole's precise orientation. This limitation makes it essential for Dreamer to "feel" its way through the task using tactile feedback.



(a) Hole insert (b) Exploded view

Figure 4.3: Drawing of upgrade 2: In this upgrade, the hole insert can now rotate around its vertical axis. This is achieved by equipping the insert with a gear that is driven by a smaller gearwheel connected to a stepper motor. A rotary encoder (Hall sensor) is directly linked to the hole insert, providing feedback on its current rotation angle. To prevent vertical displacement, the insert is secured by a screw, which engages with a rotationally symmetrical groove. A mount for the Hall sensor and the stepper motor holds everything in place. The lines of long and short dashes resemble symmetry lines. (a) shows the hole insert turned upside down to see the hole where the Hall sensor gets linked to it. (b) shows the exploded view of the setup leaving out the screws for mounting the Hall sensor and the stepper motor to keep a better overview.

4.2 Software and Computation Hardware

Dreamer utilizes a model with a parameter count ranging from 8 to 200 million, depending on the specific configuration. This considerable model complexity naturally results in significant computational demands for executing and optimizing the model effectively and in real time. To accommodate these requirements, a high-performance PC was employed with the following specifications: an AMD Ryzen 9 7950X3D 16-Core Processor, 64 GB of RAM, and a GeForce RTX 4090 graphics card. This setup ensures that the computational load associated with training and executing Dreamer can be managed efficiently, providing the necessary resources for the model's intensive operations.

The software stack used in this project incorporates various tools to facilitate development, environment management, and experimental logging. Version control for the source code is managed with Git [9], ensuring a robust and collaborative workflow. Miniconda [3] is utilized to create and manage isolated environments, streamlining dependency handling and ensuring reproducibility. Development is carried out using Visual Studio Code (VS Code) [42] as the integrated development environment (IDE), enhanced with Python extensions to support debugging, linting, and code navigation. For processing data from the OptiTrack system, Motive [43] is employed. Experiment results are logged and visualized using Weights & Biases [5], enabling comprehensive tracking and analysis of experimental progress. The project runs on Python version 3.11.9, and the Python packages used for implementation and analysis are introduced incrementally in the subsequent sections, offering detailed insights into their roles and integration.

4.3 Implementation

This section outlines the implementation of the various hardware interfaces and explains how they are integrated into the environment class. It also details the structure of the reward function and describes how Dreamer is incorporated into the setup.

4.3.1 Hardware Interfaces

A key component of the implementation is the codebase that facilitates interaction with the hardware used in the experimental setup. It consists of various classes developed to serve as interfaces to the respective hardware systems.

The first class `PandaReal` enables interaction with the Franka Research 3 robot. It uses an existing Python package, franky [50], which itself builds upon the official Franka library [15]. It delivers smooth motions and leverages Franka's internal impedance controller for the execution of movements. The class provides several essential functions, including moving the robot to its neutral position, performing absolute and relative motions along a specified axis, or executing movements across all three axes simultaneously. These motions can either maintain a default end-effector orientation or adjust it based on a quaternion passed to the function. Additionally, the class includes methods for retrieving the robot's state in various forms, such as its joint positions or the position and orientation of the end effector.

Two additional classes handle input from the `GelSight` Mini and the Intel `RealSense` cameras. Despite having similar structures and naming conventions, the classes are implemented separately due to the use of distinct Python packages, each tailored to its respective hardware. Attempts to merge the functionality into a single class resulted in errors related to hardware or implementation incompatibilities, making separation necessary.

Another interface class is dedicated to receiving data from the `OptiTrack` system. This class is built using the Python NatNet Client package by Schneider [51]. It provides a straightforward way to access and process the position and orientation data of tracked objects, such as the cylinder.

Finally, a class was implemented to manage serial communication with the `Arduino`, which controls the hole insert's orientation. The core function of this class is the `new_angle`-method, which sends a command to the Arduino to rotate the hole insert by a randomly determined amount. After waiting for the rotation process to complete, the method sends another command to retrieve the current rotation angle of the hole.

4.3.2 The Environment Class

All hardware interface classes are integrated into a central environment class, built using the Gymnasium framework [14]. This environment class serves as the foundation for the interaction between the system and the real-world setup, coordinating all hardware components.

Key functionalities of the environment class include the step and reset methods. The step function defines how the system executes an action, capturing the resulting observations, rewards, and termination signals. The reset function ensures the system is ready for a new

episode by reinitializing the hardware setup, such as repositioning the robot, resetting the cylinder, and adjusting the hole orientation. These are the key methods to interact with the real environment and collect data for Dreamer.

Reset function

The reset function is primarily adapted from the methodology outlined by Palenicek et al. [47], with a few adjustments to accommodate the changes made to the experimental setup. The reset procedure begins by pulling the cylinder out of the hole if the last episode was successful. First, the gripper is opened and then the cylinder is pulled out to protect the thin thread from unnecessary stress. If the cylinder was not successfully inserted during the previous episode, the end-effector instead raises to a predefined height and releases the cylinder, ensuring that the thread does not become entangled with the OptiTrack markers. Afterward, the robot moves the end-effector to a neutral position, defined via joint positions, to prevent the robot from entering singularities over extended operations. The reset process then continues by positioning the end-effector above the reset box. The robot lowers the end-effector, grasps the cylinder securely, lifts it back up, and moves to the random initial position of the next episode. Throughout this sequence, each step is carefully monitored using OptiTrack to verify the cylinder's orientation and placement. If any issues arise, such as the cylinder being twisted or misaligned during the insertion into the reset box, the procedure attempts to recover multiple times. If these recovery attempts fail after several retries, the entire training process is halted to avoid continued errors. In cases where the cylinder remains correctly aligned and untwisted at the end of an episode, the next episode can commence immediately without a reset. However, this is limited to a maximum of five consecutive episodes to ensure stability. After this threshold, the system automatically resets the cylinder without verifying its orientation.

With the incorporation of the second upgrade to the experimental setup, the reset function requires further adjustments to accommodate the rotational capabilities of the end-effector. If the cylinder was successfully inserted into the hole during the previous episode, the end-effector must now maintain its orientation while pulling the cylinder out. It is ensured by a straight extraction aligned with the cylinder's current orientation and the gripper closed, preventing unnecessary stress on the thin thread or damage to the experimental components. Only after the cylinder has been fully removed from the hole, the end-effector can adjust to its neutral orientation. The measurement of the cylinder's orientation has also been updated. It is now determined relative to the orientation of the end-effector when deciding whether the next episode can proceed without a reset. Additionally, the

orientation of the hole must be randomized before each new episode, regardless of whether a full reset procedure is needed. Importantly, this randomization can only occur after the cylinder has been fully extracted from the hole, ensuring that the reset procedure and the task preparation remain sequential and organized.

Step function

The step function in the environment can be divided into three distinct parts:

1. Performing the action
2. Maintaining the action frequency
3. Calculating the reward and returning values

The first part of the step function focuses on executing the action provided by Dreamer. When using the first upgrade of the setup, this process is straightforward. The action vector $\boldsymbol{a} = [\Delta x, \Delta y, \Delta z]^\intercal$ specifies a relative movement along the three axes. The robot performs this motion asynchronously, meaning the function does not wait for the movement to complete before returning. This approach ensures continuous and smooth operation, avoiding interruptions between actions. With the second upgrade of the setup, which introduces rotational capabilities for the end-effector, the action vector expands to five components: $\boldsymbol{a} = [\Delta x, \Delta y, \Delta z, \theta_1, \theta_2]^\intercal$. The first three values still govern the relative translational movement, while the additional two control the rotation of the end-effector. For a detailed explanation of how the orientation is modified using these values, refer to Section 6.1. The key concept to understand here is that the rotation design ensures the gripper consistently faces the same direction avoiding to turn the cylinder around the symmetry axis.

The second part of the step function ensures that the action frequency of $10\,\mathrm{Hz}$ is maintained throughout all experiments. To achieve this, the function waits for an appropriate amount of time after starting the action. However, simply waiting for a fixed duration is not sufficient, as the execution of the code within the step function, along with Dreamer's processing time to compute the next action, also takes up noticeable time. To address this issue, the function estimates the duration of these intermediate computations and adjusts the waiting time accordingly. This dynamic adjustment ensures that the overall timing aligns with the desired action frequency, maintaining a consistent and predictable interaction with the environment.

In the third part of the step function, the reward is calculated based on the current state of the environment after taking the agent's actions. Details about the reward computation are provided in the next section. Additionally, the function checks whether the episode has reached its termination conditions. An episode is considered finished under several circumstances:

- The goal is successfully achieved, meaning the cylinder is inserted into the hole.
- The cylinder leaves the defined workspace.
- The cylinder tilts beyond a predefined threshold, indicating excessive rotation.
- The maximum number of steps $N = 400$ for the episode is reached, at which point the episode is also truncated.

When using the second upgrade of the setup, the orientation of the end-effector introduces an additional termination condition. If the end-effector rotates too far beyond the allowable limits, the episode ends to prevent undesired behaviors.

At the end of each episode, the function also fills an info dictionary with relevant data about the episode, such as whether it was successful, the insertion time for successful attempts, and the individual reward shares. This information is valuable for analysis and debugging purposes. Finally, the step function returns all the required values, including the current state, the reward, whether the episode is finished, and the info dictionary.

Reward

The reward function, much like the experimental setup, underwent numerous adjustments throughout this work. To provide precise mathematical expressions for the reward function, the following vectors are defined: $\boldsymbol{p}_c \in \mathbb{R}^3$ represents the position of the lower tip of the cylinder, $\boldsymbol{p}_g \in \mathbb{R}^3$ represents the goal position, $\boldsymbol{p}_h \in \mathbb{R}^3$ represents the position of the hole on the base plate, and β denotes the rotation of the cylinder within the gripper, described as a rotation around a single axis. Initially, it included several components designed to guide the robot's learning process effectively. A significant final reward of $+100$ was granted for successful task completion. Conversely, substantial penalties were assigned to discourage undesirable outcomes: a final penalty of -100 was applied if the cylinder left the workspace, and another penalty of -50 was given if the cylinder's absolute rotation exceeded 10° leading to the end of that episode. These terminal rewards and penalties are summarized in the following equation:

$$r_t = \underbrace{100 \cdot \mathbb{1}_{\{\mathcal{G}\}}(\boldsymbol{p}_c)}_{\text{terminal reward}} \underbrace{-100 \cdot \mathbb{1}_{\{\mathcal{W}\}}(\boldsymbol{p}_c)}_{\text{workspace penalty}} \underbrace{-50 \cdot \mathbb{1}_{\{\mathcal{R}\}}(\beta)}_{\text{terminal rotation penalty}},$$

where $\mathcal{G} = \{\boldsymbol{x} \in \mathbb{R}^3 : |\boldsymbol{p}_g - \boldsymbol{x}| < (5, 5, 5)[\mathrm{mm}]\}$ and $\mathbb{1}_{\{\mathcal{G}\}}(\boldsymbol{p}_c)$ is 1 when $\boldsymbol{p}_c$ lies inside the set $\mathcal{G}$, otherwise its value is 0. The workspace is described in relation to the position of the hole $\boldsymbol{p}_h$ by the set $\mathcal{W} = \{\boldsymbol{x} \in \mathbb{R}^3 : (-3, -3, -5)[\mathrm{cm}] < \boldsymbol{x} - \boldsymbol{p}_h < (3, 3, 6)[\mathrm{cm}]\}$ leading to cubic workspace with the edge length $6[\mathrm{cm}]$ on the surface of the base plate. Notice that the workspace also goes underneath the surface of the base plate, avoiding that the cylinder goes out of the workspace when it is inserted into the hole. The set of all possible rotations is described by $\mathcal{R} = \{\phi \in \mathbb{R}^3 : \phi < 10°\}$.

Additionally, intermediate penalties were included to fine-tune the robot's behavior. A linear penalty was introduced for rotations of the cylinder between 4° and 10°, encouraging the system to minimize misalignments. Another linear penalty discouraged all distances from the goal, incentivizing closer alignment with the target position. Finally, an action penalty was implemented to encourage efficient use of actions, discouraging unnecessary or overly large movements. These continuous rewards and penalties are summarized in this equation:

$$r_c = \underbrace{-5 \cdot |\boldsymbol{p}_g - \boldsymbol{p}_c|}_{\text{linear penalty}} \underbrace{-0.1 \cdot \max\{(\beta - 4°), 0\}}_{\text{rotation penalty}} \underbrace{-10^{-3} \cdot |\boldsymbol{a}|}_{\text{action penalty}},$$

resulting in six different reward terms when adding up continuous and terminal reward shares.

While this reward function was designed to account for a variety of factors influencing the robot's performance, it quickly became apparent that its complexity posed significant challenges to the learning process. Although the original setup allowed the agent to learn the task with this reward function, introducing the first upgrade revealed some serious downsides. Specifically, the agent frequently adopted a counterproductive strategy: it learned to exit the workspace as quickly as possible. This approach minimized the linear penalty for distance from the goal and avoided penalties related to cylinder rotation. Attempts to address these issues led to further modifications of the reward function. However, these adjustments only increased the complexity of the reward structure and often yielded limited improvements. The challenge lies in creating a reward system that effectively guides the agent toward successful task completion without encouraging unintended behaviors or adding unnecessary computational overhead. These experiences

underscored the importance of balancing simplicity and effectiveness in reward function design.

After extensive trial and error, the reward function was completely revised. The new approach replaced the linear penalty, which was originally designed to attract the cylinder to the goal by minimizing the penalty near the goal, with a linear reward. This new term is nearly zero at the edges of the workspace and reaches its maximum value at the goal position, directly encouraging the agent to move closer to the goal. Introducing this linear reward significantly simplified the reward function by making many of its components redundant. For example, the final penalty for leaving the workspace became unnecessary. Since leaving the workspace early reduces the agent's opportunity to accumulate the linear reward, it implicitly discourages the agent from exiting the workspace. Similarly, the final penalty for the cylinder rotation was also removed. Tilting the cylinder still ends the episode prematurely, reducing the linear reward the agent can earn and naturally penalizing this behavior without requiring an explicit term. The revised reward function was stripped down to just three essential terms:

1. The linear reward encourages proximity to the goal by increasing as the cylinder gets closer.
2. A final reward, granted upon successful completion of the task.
3. An action penalty, to prevent excessive or unnecessary movements.

It is expressed using the following mathematical equation:

$$r = \underbrace{5 \cdot (0.1 - |\boldsymbol{p}_g - \boldsymbol{p}_e|)}_{\text{proximity to the goal}} \underbrace{+500 \cdot \mathbb{1}_{\{\mathcal{G}\}}(\boldsymbol{p}_g)}_{\text{terminal reward}} \underbrace{-10^{-3} \cdot |\boldsymbol{a}|}_{\text{action penalty}},$$

using the same set for $\mathcal{G}$ as described above. This new reward structure means a real simplification compared to the previous one and additionally guides the agent successfully.

4.3.3 Integration of Dreamer

With the environment fully implemented, the next step is integrating Dreamer to train and evaluate the model. Dreamer provides various scripts to execute its model, and for this work, the `parallel` script is utilized. This script enables the parallelization of data collection by interacting with the environment and the simultaneous training of the model

using the collected data. This parallel execution is crucial because, without it, the training process would interrupt the flow of an episode, causing breaks that make it impossible to maintain the required action frequency. Such interruptions would result in episodes unusable for training. Additionally, the `eval_only` script from Dreamer is employed to evaluate the model's performance. This script is also used to collect replay data, which is later utilized to perform Shapley value analysis.

To execute Dreamer, the environment provides observations in the form of visual and tactile inputs. Specifically, the visual observations are denoted as $\boldsymbol{o}^{\text{vis}} \in \mathbb{R}^{64\times64\times3}$, and the tactile observations as $\boldsymbol{o}^{\text{tac}} \in \mathbb{R}^{64\times64\times3}$. Both types of observations are scaled down to these specified dimensions to match the input requirements of Dreamer. Although these dimensions might seem small compared to typical high-resolution inputs, they have been found sufficient in this work for Dreamer to successfully learn the task. Dreamer processes these inputs and determines the next action, which is then passed back to the environment for execution. The action returned by Dreamer is either a three-dimensional or five-dimensional vector, depending on the setup as described in Section 4.3.2. A visual summary of the integration of Dreamer and the environment class is shown in Figure 3.3.

Adjustments to the Default Dreamer Implementation

During the integration of Dreamer, several adjustments were made to the framework to enhance its compatibility and usability. One of the key changes was adding support for Gymnasium environments. Originally, Dreamer only supported the legacy gym environments, but this limitation was overcome by incorporating a ready-to-use class from Toyer [56], enabling seamless integration with Gymnasium.

Dreamer also manages the logging of important metrics to Weights & Biases, which is crucial for tracking and evaluating the training process. To log additional data, developers traditionally needed to extend the observation space by introducing so-called log variables (the keys of these variables start with the string "`log_`"). While these variables are not visible to the policy, they are recorded during the logging process. However, this approach proved inconvenient during development, as it required modifications to the observation space whenever new data needed to be logged. To address this issue, the implementation was revised to pass the info dictionary through Dreamer and log all relevant data from it. This adjustment not only simplifies the logging process but also keeps the observation space clean. The new approach provides the flexibility to log new data effortlessly, improving the overall development workflow.

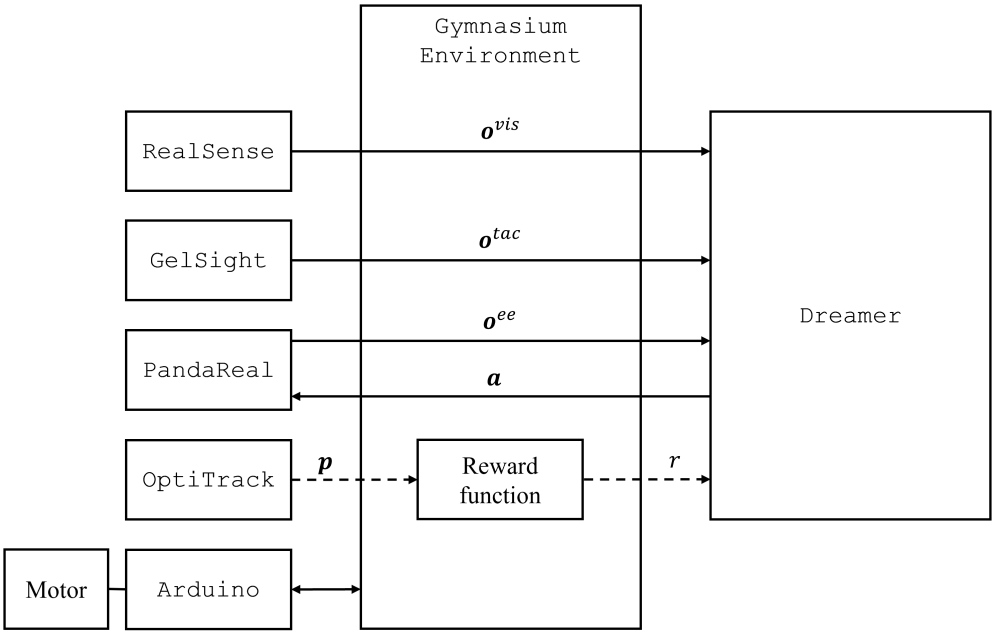


Figure 4.4: Integration of Dreamer and the environment class: The `RealSense`, `GelSight`, or `PandaReal` classes deliver the observations, depending on the configuration of the observation space. Dreamer sends back actions a through the environment to the robot. The dashed lines visualize the flow of data to calculate the reward. The `OptiTrack` class sends information about the current positions p of the cylinder and the goal. The reward function then calculates the reward r. In the newest upgrade of the setup, the `Arduino` class is used to adjust and measure the orientation of the hole. This graphic is not intended to be exhaustive but illustrates the most important data flows between the environment and Dreamer.

While using the `parallel` script to integrate Dreamer, a significant issue was discovered related to the saving of checkpoints during the parallelized processes. Many functions within the script rely on a `should_save` variable to determine when to save the current checkpoint. However, this variable is not initialized simultaneously across all parallel processes. As a result, there were inconsistencies in the timing of checkpoint saves, leading to relatively large and unpredictable time spans between saves. To address this issue, a centralized `should_save` variable was introduced. This central variable is shared among all processes that need to check its state, ensuring synchronized checkpoint saving across all parallel tasks. The functionality for this shared variable was implemented using Python's `BaseManager` class from the multiprocessing package.

In the original implementation of Dreamer, visual input modalities were processed by stacking the images on top of each other to form a single $64 \times 64 \times 6$ vector. This combined input was then passed through a single CNN in the encoder. While this method is effective for multiple cameras capturing the same scene from different angles, it is not an ideal solution when dealing with distinct visual modalities, such as visual and tactile inputs, even though tests showed that it remains functional when using the original implementation. To improve upon this, the encoder was restructured to use separate CNNs for each high-dimensional input modality. This change not only allows for more tailored processing of each input type but also introduces the flexibility to handle inputs with differing dimensions.

Hyperparameters

The hyperparameters of Dreamer remained fixed throughout all experiments to ensure consistency in the algorithm's performance. However, the experimental setup introduced additional hyperparameters that required careful consideration to optimize the learning process. Examples of these hyperparameters include the dimensions of the workspace, the maximum length of an episode, and the operational velocity during an episode. A detailed list of these hyperparameters can be found in Section 6.2 for reference.

To simplify the manipulation of these experimental hyperparameters, a YAML configuration file was introduced. This file not only allows easy adjustment of hyperparameters but also contains other essential configuration values. For instance, it specifies how frequently Dreamer should log videos to Weights & Biases, helping to avoid excessive logging.

4.4 Integration of Shapley Value Analysis

To conduct Shapley value analysis, it is necessary to load both a checkpoint of the current model and the corresponding replays of real trajectories. The most reliable way to obtain these checkpoints and replays is by utilizing data from an evaluation phase. During evaluation, the model remains static and does not update its parameters, ensuring consistent and reproducible results. The evaluation process is designed to facilitate this analysis by saving the specific checkpoint used and its associated replays in a separate folder.

The first crucial question in performing Shapley value analysis is determining which input features to include in the computation. Initial experiments explored separating the images from visual and tactile feedback into 128 equally sized sections or features, resembling a checkerboard pattern. However, this approach resulted in highly noisy Shapley values, making it challenging to draw meaningful conclusions. One potential explanation for this behavior is the sheer number of feature combinations involved in this setup. With 128 features, the possible combinations scale exponentially to 2^{128}, making the estimation of Shapley values computationally expensive and less reliable due to the limited sampling possible in practical scenarios. An alternative approach that yielded better results was to define four distinct features: the hidden state, the previous action, the visual input as a whole, and the tactile input as a whole. This reduction in feature count improved the results of the Shapley value analysis, providing clearer insights into the contributions of these key components to the model's decision-making process.

The next important consideration in Shapley value analysis is determining how to appropriately mask the modalities when they are absent. This masking process is critical to ensure that the model's behavior under different feature combinations accurately reflects the impact of the omitted inputs. For the hidden state, the masking is handled using a built-in Dreamer function. This function initializes the hidden state in the same way it does at the start of a new episode when the hidden state carries no information other than the initialization itself. The previous action is masked by simply replacing it with zeros. This straightforward approach ensures that the policy does not receive any actionable information when the previous action is considered absent. For the visual and tactile inputs, both image modalities are masked by replacing them with zero-filled arrays, effectively creating completely black images. These black images are then passed into the policy when an image modality is absent.

The implementation of Shapley value analysis in this work leverages an existing SHAP framework [40]. Specifically, the Exact Explainer [39] is utilized, as it is well-suited for

cases with fewer than 15 input features and as the name suggests calculates exact Shapley values without estimation. To use the Exact Explainer, its constructor requires two key inputs: a policy function and a mask for the data. A common approach to define this mask is to use arrays of zeros and ones, where an array of 4 zeros indicates the absence of all features, and an array of 4 ones signals the presence of all features. This masking convention allows the explainer to systematically generate and test all combinations of feature presence and absence. For each combination, the explainer passes the mask to the policy function. To handle these masks, a custom policy function needs to be implemented. This function is designed to process the masked input, apply the required transformations (e.g., masking the hidden state, previous action, visual input, or tactile input), and then execute Dreamer's policy using the modified input data. The remaining steps are handled by the SHAP framework. The explainer computes and returns the Shapley values, showing how each input feature contributed to every action value. For example, with 4 input features and 3 action values, the framework produces 12 Shapley values in total. These values provide a detailed insight into the importance of each feature in determining the model's actions, forming a basis for analyzing and interpreting the model's decision-making process.

5 Results

This section presents the results of two separate studies. The first study involves experiments using the original setup, incorporating both tactile and visual feedback to serve as a proof of concept. During this phase, an ablation study is conducted to evaluate the impact of removing tactile feedback. In the second study, experiments are performed using an upgraded, modular setup (upgrade 1), which introduces varying hole diameters and deeper holes with a depth of $4\mathrm{cm}$. Similar to the first study, an ablation of tactile feedback is conducted to assess its influence on task performance. Following the training phase, the different policies are evaluated, and Shapley value analysis is applied to investigate the relative importance of each modality. Although a second upgrade to the setup was developed to further increase the task complexity, experiments with this version have not been conducted yet.

5.1 Experiments with the Original Setup

The experiments begin with the original setup and original reward function, serving as a proof of concept to validate the system's functionality and initial assumptions. In this configuration, the camera is mounted to the end-effector. The the cylinder has a diameter of $20\ \mathrm{mm}$, the hole has a diameter of $23\ \mathrm{mm}$ and a depth of $17\ \mathrm{mm}$. As part of the initial experimentation, an ablation study is performed to evaluate the importance of tactile feedback. By removing the tactile sensor input, the experiment assesses how the absence of this modality impacts task performance.

The experiments demonstrate that Dreamer is capable of solving the peg-in-hole insertion task using both visual and tactile data in the original setup within approximately $100,000$ steps. This performance is significantly better than all experiments conducted in previous work [47], where no visual input is utilized.

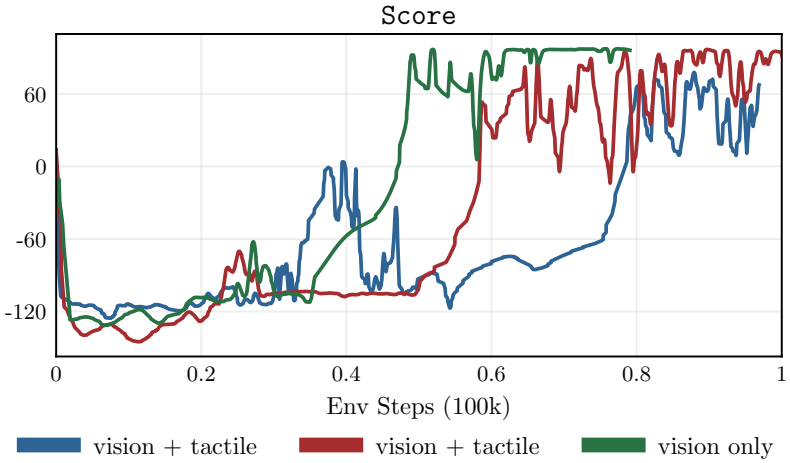


Figure 5.1: Training results with the original setup: In the first experiments consist of three runs in total, the first two including visual and tactile feedback, and the third with vision only. The results showed in this figure suggest that vision alone can handle this specific task better compared to the two runs including both modalities. However, this assertion is made with the three available runs, reducing its significance. At the same time, the confirms the assumption that this specific task is too simple, and tactile feedback has no impact on the training process.

As shown in Figure 5.1, the vision-only policy performs better than the previous runs that combined visual and tactile feedback. However, due to the limited amount of data available, this result does not necessarily indicate a general trend. It is possible that the vision-only run benefits from a degree of "luck" compared to the other two runs, for instance, by having more favorable random initial positions or more frequent goal-reaching events during the early exploration phase of training. On the other hand, it is also plausible that the inclusion of tactile feedback introduces noise, which could hinder the performance of the vision-tactile policies. Regardless of which explanation is accurate, the outcome of these experiments indicates that tactile feedback does not provide a significant advantage for solving this specific task. The peg-in-hole insertion task in this setup appears to be simple enough that visual input alone is sufficient to solve it.

5.2 Experiments with Upgrade 1

In the second phase of the experiments, the setup is upgraded to the first modular version, incorporating the simplified version of the reward function. Additionally, the camera is repositioned to provide a fixed view of the scene. One of the key advantages of this upgraded setup is the ability to easily change the hole inserts, enabling the use of different configurations for training.

5.2.1 Training

Two distinct hole inserts are employed for training, each designed with a hole depth of $40\ \mathrm{mm}$. The first insert has a hole diameter of $22\ \mathrm{mm}$, while the second has a diameter of $20.5\ \mathrm{mm}$. It is important to note that these dimensions are based on the digital 3D models and may differ slightly from the actual measurements of the physical, 3D-printed components. For each hole insert, two separate policies are trained: one using only visual input and another utilizing both visual and tactile input. In total, this creates four distinct experimental configurations. To ensure the results are statistically significant and to account for variability during training, multiple models (2–3) are trained for each configuration. This redundancy strengthens the reliability of the findings and helps identify consistent trends across experiments. The results of these experiments are summarized in Figure 5.2, showcasing the training progress of each policy under the respective conditions. In addition, Dreamer logs a lot of other valuable data. A selection of additional figures and graphs can be found in Section 6.3.

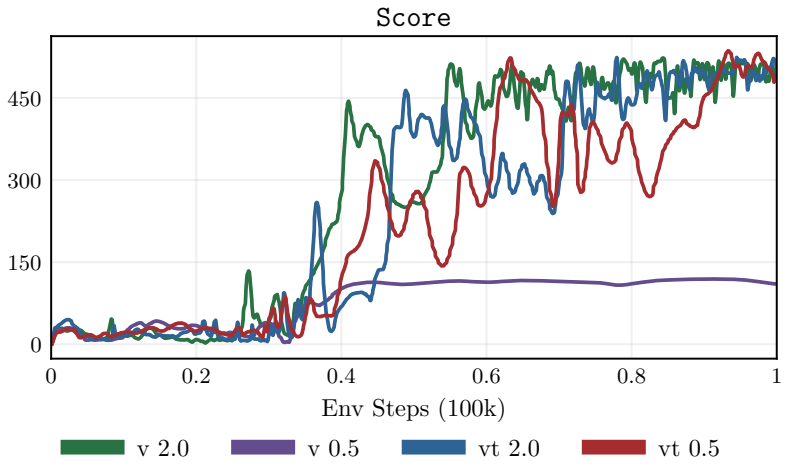


Figure 5.2: Training results with the first upgrade: Four different policies were trained with individual combinations of modalities and hole tolerances. The letters in the beginning of the label indicate the used modalities ("v" for visual, and "t" for tactile). The number afterward represents the tolerance between peg and hole in $[\mathrm{mm}]$. Every model was able to solve the task in nearly 100k steps, except the vision-only policy trained on the tighter hole.

It is striking that all experiments, except for one, learned the task in nearly the same amount of time, regardless of the tolerance or whether tactile input was included. The only exception was the visual-only policy trained with the 0.5mm tolerance. This policy failed to solve the task. During the corresponding experiment, it was observed that the cylinder often got jammed inside the hole. Without tactile feedback, the policy struggled to adjust the alignment and find a way to insert the cylinder deeper into the hole. To verify this result, an experiment was conducted, extending the training time to over 150k steps. However, the policy still failed to solve the task, further confirming that visual input alone was insufficient for successful training in such precise conditions.

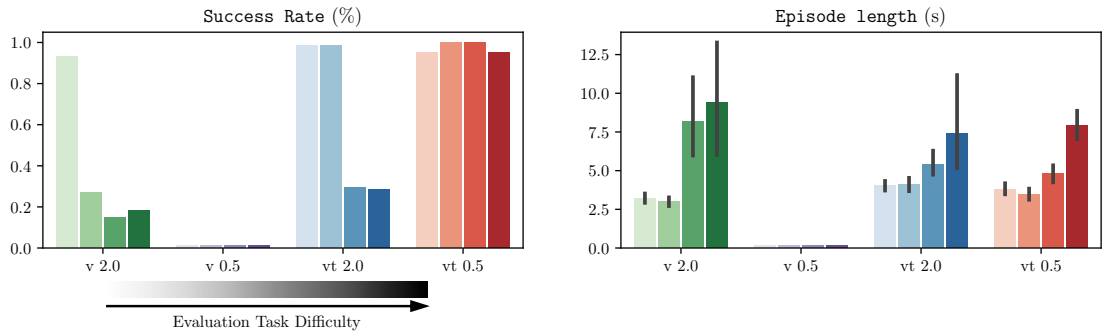
5.2.2 Evaluation

To thoroughly evaluate the performance of the trained models, various hole configurations are tested. This evaluation includes hole diameters that are smaller than those used during the original training, introducing additional challenges that require the models to adapt to tighter tolerances. Furthermore, the evaluation incorporates slightly angled holes, which test the models' ability to handle scenarios where the hole is not perfectly vertical, increasing the complexity of the insertion task. A detailed summary of all the hole configurations used for evaluation can be found in Table 5.1.

Configuration	Tolerance between peg and hole	Angle of the hole
1	$2\ \text{mm}$	$0°$
2	$1\ \text{mm}$	$4°$
3	$0.5\ \text{mm}$	$0°$
4	$0.5\ \text{mm}$	$4°$

Table 5.1: Hole configurations to evaluate the performance of the trained policies with increasing complexity.

With the trained policies from the first upgrade of the setup, their performance can now be evaluated using the evaluation configurations proposed in Table 5.1. These configurations test the policies' ability to generalize and adapt to new scenarios, including tighter tolerances and slightly angled holes. Figure 5.3 illustrates the success rate and the average insertion time for each of the four policies evaluated across the four configurations. Each evaluation was performed 20 times to ensure statistical significance and to account for variability in the results.



(a) Success rate

(b) Insertion time

Figure 5.3: Evaluation using the four proposed evaluation configurations with increasing complexity. (a) shows the percentage success rate of the policies over 20 evaluation episodes. The four bars, illustrated with every individual policy, display the performance using the four evaluation configurations. (b) shows the average insertion time over 20 evaluation episodes, when an episode was successful. The gray lines illustrate the 95% confidence intervals of the episode length.

First of all, it is clear from Figure 5.3 that the visual-only policy trained with 0.5mm tolerance (`v 0.5`) fails to solve any evaluation task. This outcome is expected, as the policy never experienced any successes during training. Consequently, it did not learn what to do even when it managed to reach the hole or insert the cylinder slightly. It lacked the understanding of how to complete the task.

On the other hand, both policies trained with a 2mm tolerance (`v 2.0` and `vt 2.0`) demonstrate some success in evaluation tasks with tighter tolerances. Particularly noteworthy is the fact that the visual-only policy (`v 2.0`) achieves success even with a 0.5mm tolerance. This is remarkable, given that training a policy from scratch with vision only and such tight tolerances shows to be infeasible in this work. This result suggests that tactile feedback during training plays an important role in guiding the alignment process and avoiding jamming. However, once the policy has a clear understanding of the task (locating the hole and pushing down) vision alone can solve tasks with tighter tolerances, even though with much lower success rates. Interestingly, the `vt 2.0` policy shows a significant performance improvement over `v 2.0` in the second evaluation task, where the tolerance is 1mm, and the hole is angled at $4°$. This indicates that tactile feedback becomes more critical in scenarios involving additional complexity, such as slightly angled holes,

where precise alignment is challenging using vision alone. Overall, the best-performing policy across all evaluation tasks is `vt 0.5`. It achieves the highest success rates and the shortest insertion times.

5.2.3 Shapley Value Analysis

In the next step, Shapley value analysis is employed to evaluate the contributions of each input to the model's final decision-making process. As previously outlined, the inputs under analysis are the previous action $\boldsymbol{a}_{t-1}$, the previous latent state $\boldsymbol{h}_{t-1}$, the current visual observation $\boldsymbol{o}_t^{vis}$, and the current tactile observation $\boldsymbol{o}_t^{tac}$. This analysis aims to quantify the relative importance of each modality and input source in guiding the policy's actions. Figure 5.4 illustrates the results of the Shapley value analysis for an example trajectory taken from the `vt 0.5` policy. This policy was evaluated on the most complex configuration in the evaluation set, which includes both a tight tolerance and a slightly angled hole.

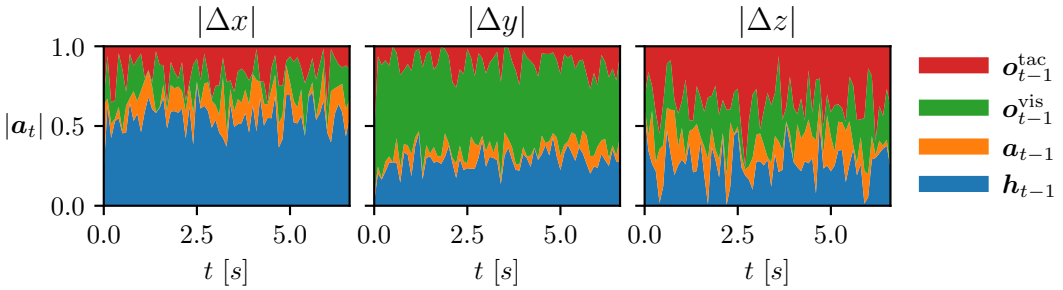


Figure 5.4: Shapley value analysis with an example trajectory from the `vt 0.5` policy evaluated the on most complex evaluation configuration. Each plot shows the percentage contribution of each model input to the action in this axis.

For the example trajectory, the Shapley value analysis shows clear differences in reliance on the different inputs depending on the axis of motion. For movements along the x-axis, the latent state emerges as the most important factor. This makes sense given that the x-axis corresponds to depth from the perspective of the camera. The policy relies on the latent state to estimate depth by leveraging temporal information about previous states in the trajectory. For movements along the y-axis, visual input plays the dominant role. This is expected, as the y-axis controls lateral motion (left-to-right) relative to the camera's viewpoint. In contrast, movements along the z-axis, which control vertical motion, are

influenced most strongly by tactile feedback. However, the dominance of tactile input in this axis is less pronounced compared to the reliance on visual input for y-axis movements or latent state information for the x-axis. This suggests that tactile sensing is primarily used to detect physical contact, such as identifying when the peg reaches the surface of the base plate or the hole. It plays a smaller role in fine-tuning alignment or guiding insertion, underlining the thesis from before that tactile feedback becomes less critical once the policy has learned the general strategy of locating the goal and pushing downward.

6 Discussion and Conclusion

The initial experiments with the original setup provided a successful proof of concept, demonstrating that a simple peg-in-hole insertion task could be solved using Dreamer with visual and tactile input. However, these results also revealed that the setup was too simple, as the task could be accomplished using only visual input without requiring tactile feedback. To introduce greater complexity, a modular setup was developed for the second phase of experiments, incorporating varying tolerances between peg and hole, ranging from 0.5mm to 2mm, as well as hole depths of 4cm. The results of these experiments indicated that training a policy from scratch with a 0.5mm tolerance required tactile feedback to successfully complete the task. At the same time, the evaluation phase showed that policies trained solely with visual input on a 2mm tolerance hole were still capable of solving the task with 0.5mm tolerances, even though with a significantly lower success rate. Further analysis using Shapley values provided insights into how different sensory modalities contribute to the learned policy. The results showed that visual feedback primarily guides the end-effector in the camera plane, while tactile feedback plays a crucial role in controlling vertical movement, where contact forces are most significant. However, it is important to highlight that these findings are based on a single trajectory. To further validate these conclusions, additional investigations and alternative explanation strategies are necessary to ensure the significance of the results.

The results from the first experiments using the original setup were very promising and demonstrated Dreamer's impressive capabilities. The system was able to successfully learn and solve the peg-in-hole task, even with complex multi-modal inputs. However, these results raised a significant question: why were the performance results in these experiments so much better compared to those observed in previous work [47], which also used tactile observations and the end-effector position? Intuitively, you would expect that the information provided by the end-effector position should be more useful for the task compared to visual input from the camera. After all, knowing the exact position of the end-effector should theoretically allow for more precise control over the peg's movements. However, this was not the case in the experiments. Despite the expectation,

the performance with visual input exceeded that of the model using the end-effector position. To investigate this discrepancy further, additional experiments were conducted that included a normalization strategy for the end-effector position. However, these efforts did not resolve the issue, and the question remains unsolved.

At the same time, the results of the first experiments showed that vision-only policies were able to solve the task just as effectively, if not better, than policies that combined both vision and tactile feedback. Therefore, the first upgrade to the setup was developed, making it more challenging by using deeper holes and arbitrary hole diameters. Subsequent experiments with very tight tolerances, specifically $0.5\mathrm{mm}$, demonstrated that tactile feedback became essential for training the model effectively on these tighter tolerances. However, it was still possible for vision-only policies to solve the task with $0.5\mathrm{mm}$ tolerances, although with lower success rates and longer episode lengths. One possible interpretation of this result is that tactile feedback serves as a form of scaffolding during the training process, assisting the model when it encounters challenges with tighter tolerances. This assumption aligns with the concept of Scaffolded Reinforcement Learning, as described in the related work by Hu et al. [29]. But to make a final assertion about the role of tactile feedback in this context, further investigations and experiments would be necessary.

The Shapley value analysis further supports the assumption that while tactile feedback does play a role in Dreamer's decision-making process, it is not a crucial one, at least not in the final policy. The analysis also revealed that the Shapley values do not show distinct stages within the specific tasks. This observation contrasts with several related studies [13, 16, 38], which explicitly divided their tasks into distinct stages. In these works, the distinct stages of the task were often supported by specific modalities. However, in Dreamer's case, there were no noticeable jumps in the Shapley values that would indicate clear transitions between stages of the task. To address this fact and hopefully make the task more reliant on tactile feedback, the second upgrade to the setup was developed.

Besides the already mentioned possibilities for future investigations, there are numerous other aspects for further research. First, the constructed platform provides an excellent opportunity to test different RL and Machine Learning (ML) algorithms. Therefore, the experiments with Dreamer can serve as a strong baseline against which other models can compete. Another possible direction for future work is the use of differently shaped pegs, where the end-effector would need to rotate around the z-axis. This would require the robot to utilize all 6 degrees of freedom, adding an additional layer of complexity to the task. A third potential direction for future research is to modify the setup to simulate a more real-world scenario, such as inserting a key into the corresponding lock. As shown, there are many possibilities for future work, and the setup developed in this and previous

work [47] can be reused for a wide variety of projects, making it a versatile platform for advancing research in robotic manipulation and multi-modal learning.

Bibliography

[1] Prusa Research a.s. *Prusa i3 MK3S*. URL: https://www.prusa3d.com/product/original-prusa-i3-mk3s-to-mk3-5-upgrade-kit-2/.

[2] Prusa Research a.s. *PrusaSlicer*. URL: https://www.prusa3d.com/de/page/prusaslicer_424/.

[3] Inc. Anaconda. *Miniconda*. URL: https://docs.anaconda.com/miniconda/.

[4] Arduino. *Arduino Uno*. URL: https://www.arduino.cc/en/hardware.

[5] Weights & Biases. *Weights & Biases*. URL: https://wandb.ai/site/.

[6] A. Boehm et al. "What Matters for Active Texture Recognition With Vision-Based Tactile Sensors". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. 2024. URL: https://www.ias.informatik.tu-darmstadt.de/uploads/Team/BorisBelousov/boehm24_tart.pdf.

[7] Aaron Butterworth et al. "Leveraging Multi-modal Sensing for Robotic Insertion Tasks in R&D Laboratories". In: *19th IEEE International Conference on Automation Science and Engineering, CASE 2023, Auckland, New Zealand, August 26-30, 2023*. IEEE, 2023, pp. 1–8.

[8] Roberto Calandra et al. "More Than a Feeling: Learning to Grasp and Regrasp Using Vision and Touch". In: *IEEE Robotics Autom. Lett.* 3.4 (2018), pp. 3300–3307.

[9] Scott Chacon. *Git*. URL: https://git-scm.com/.

[10] Intel Corporation. *RealSense D405*. URL: https://www.intelrealsense.com/depth-camera-d405/.

[11] Siyuan Dong and Alberto Rodriguez. "Tactile-Based Insertion for Dense Box-Packing". In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2019, Macau, SAR, China, November 3-8, 2019*. IEEE, 2019, pp. 7953–7960.

[12] Siyuan Dong et al. "Tactile-RL for Insertion: Generalization to Objects of Unknown Geometry". In: *IEEE International Conference on Robotics and Automation, ICRA 2021, Xi'an, China, May 30 - June 5, 2021*. IEEE, 2021, pp. 6437–6443.

[13] Ruoxuan Feng et al. “Play to the Score: Stage-Guided Dynamic Multi-Sensory Fusion for Robotic Manipulation”. In: *CoRR* abs/2408.01366 (2024).

[14] Farama Foundation. *Gymnasium*. URL: https://gymnasium.farama.org/index.html.

[15] frankaemika. *libfranka*. URL: https://github.com/frankaemika/libfranka.

[16] Letian Fu et al. “Safe Self-Supervised Learning in Real of Visuo-Tactile Feedback Policies for Industrial Insertion”. In: *IEEE International Conference on Robotics and Automation, ICRA 2023, London, UK, May 29 - June 2, 2023*. IEEE, 2023, pp. 10380–10386.

[17] Niklas Funk et al. “Evetac: An Event-Based Optical Tactile Sensor for Robotic Manipulation”. In: *IEEE Transactions on Robotics* 40 (2024), pp. 3812–3832.

[18] Inc. GelSight. *GelSight Mini*. URL: https://www.gelsight.com/gelsightmini/.

[19] Oliver Gibbons, Alessandro Albini, and Perla Maiolino. “A Tactile Feedback Insertion Strategy for Peg-in-Hole Tasks”. In: *ICRA*. IEEE, 2023, pp. 10415–10421.

[20] Franka Robotics GmbH. *Franka Research 3*. URL: https://franka.de/de/products.

[21] Danijar Hafner et al. “Dream to Control: Learning Behaviors by Latent Imagination”. In: *International Conference on Learning Representations*. 2020.

[22] Danijar Hafner et al. “Learning Latent Dynamics for Planning from Pixels”. In: *International Conference on Machine Learning*. 2019.

[23] Danijar Hafner et al. “Mastering Atari with Discrete World Models”. In: *ICLR*. OpenReview.net, 2021.

[24] Danijar Hafner et al. “Mastering Diverse Domains through World Models”. In: *arXiv preprint arXiv:2301.04104* (2023).

[25] Yunhai Han et al. “Learning Generalizable Vision-Tactile Robotic Grasping Strategy for Deformable Objects via Transformer”. In: *CoRR* abs/2112.06374 (2021).

[26] Nicklas Hansen, Hao Su, and Xiaolong Wang. “Temporal Difference Learning for Model Predictive Control”. In: *ICML*. Vol. 162. Proceedings of Machine Learning Research. PMLR, 2022, pp. 8387–8406.

[27] Erik Helmut et al. “Learning Force Distribution Estimation for the GelSight Mini Optical Tactile Sensor Based on Finite Element Analysis”. In: vol. abs/2411.03315. 2024.

[28] Carolina Higuera et al. “Perceiving Extrinsic Contacts from Touch Improves Learning Insertion Policies”. In: *CoRR* abs/2309.16652 (2023).

[29] Edward S. Hu et al. “Privileged Sensing Scaffolds Reinforcement Learning”. In: *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[30] Hideyuki Ichiwara et al. “Contact-Rich Manipulation of a Flexible Object based on Deep Predictive Learning using Vision and Tactility”. In: *2022 International Conference on Robotics and Automation, ICRA 2022, Philadelphia, PA, USA, May 23-27, 2022*. IEEE, 2022, pp. 5375–5381.

[31] Autodesk Inc. *Autodesk Fusion*. URL: https://www.autodesk.com/de/products/fusion-360.

[32] Piaopiao Jin et al. “Visual-Force-Tactile Fusion for Gentle Intricate Insertion Tasks”. In: *IEEE Robotics Autom. Lett.* 9.5 (2024), pp. 4830–4837.

[33] Tatsuya Kamijo et al. “Tactile-based Active Inference for Force-Controlled Peg-in-Hole Insertions”. In: *CoRR* abs/2309.15681 (2023).

[34] Sangwoon Kim and Alberto Rodriguez. “Active Extrinsic Contact Sensing: Application to General Peg-in-Hole Insertion”. In: *2022 International Conference on Robotics and Automation, ICRA 2022, Philadelphia, PA, USA, May 23-27, 2022*. IEEE, 2022, pp. 10241–10247.

[35] Mike Lambeta et al. “DIGIT: A Novel Design for a Low-Cost Compact High-Resolution Tactile Sensor With Application to In-Hand Manipulation”. In: *IEEE Robotics and Automation Letters* 5.3 (2020), pp. 3838–3845.

[36] Michelle A. Lee et al. “Making Sense of Vision and Touch: Learning Multimodal Representations for Contact-Rich Tasks”. In: *IEEE Trans. Robotics* 36.3 (2020), pp. 582–596.

[37] Wenyu Liang et al. “Visuo-Tactile Manipulation Planning Using Reinforcement Learning with Affordance Representation”. In: *CoRR* abs/2207.06608 (2022).

[38] Bo-Siang Lu et al. “CFVS: Coarse-to-Fine Visual Servoing for 6-DoF Object-Agnostic Peg-In-Hole Assembly”. In: *IEEE International Conference on Robotics and Automation, ICRA 2023, London, UK, May 29 - June 2, 2023*. IEEE, 2023, pp. 12402–12408.

[39] Scott Lundberg. *ExactExplaine API reference*. URL: https://shap.readthedocs.io/en/latest/generated/shap.ExactExplainer.html.

[40] Scott Lundberg. *SHAP documentation*. URL: https://shap.readthedocs.io/en/latest/index.html.

[41] Scott M Lundberg and Su-In Lee. "A Unified Approach to Interpreting Model Predictions". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 4765–4774. URL: http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf.

[42] Microsoft. *Visual Studio Code*. URL: https://code.visualstudio.com/.

[43] Inc. NaturalPoint. *Motion Capture Software*. URL: https://optitrack.com/software/motive/.

[44] Inc. NaturalPoint. *Motion Capture Systems*. URL: https://optitrack.com/.

[45] Long Ouyang et al. "Training language models to follow instructions with human feedback". In: *NeurIPS*. 2022.

[46] Sameer Pai et al. *Precise Well-plate Placing Utilizing Contact During Sliding with Tactile-based Pose Estimation for Laboratory Automation*. 2024. arXiv: 2309.16170 [cs.RO]. URL: https://arxiv.org/abs/2309.16170.

[47] Daniel Palenicek et al. "Learning Tactile Insertion in the Real World". In: *CoRR* abs/2405.00383 (2024).

[48] Simon J.D. Prince. *Understanding Deep Learning*. MIT Press, 2023. URL: http://udlbook.com.

[49] XELA Robotics. *uSkin Sensors*. URL: https://www.xelarobotics.com/tactile-sensors#models-section.

[50] Tim Schneider. *franky*. URL: https://github.com/TimSchneider42/franky.

[51] Tim Schneider. *Python NatNet Client*. URL: https://github.com/TimSchneider42/python-natnet-client/.

[52] Carmelo Sferrazza et al. "The Power of the Senses: Generalizable Manipulation from Vision and Touch through Masked Multimodal Learning". In: *CoRR* abs/2311.00924 (2023).

[53] Mohsen Shahinpoor and H. Zohoor. "Analysis of dynamic insertion type assembly for manufacturing automation". In: *ICRA*. IEEE Computer Society, 1991, pp. 2458–2464.

[54] Lloyd S Shapley. "A value for n-person games". In: *Contribution to the Theory of Games* 2 (1953).

[55] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[56] Sam Toyer. *Gymnasium wrapper*. URL: https://gist.github.com/qxcv/e8641342c102c2aa714c9caeca724101.

[57] Benjamin Ward-Cherrier et al. "The tactip family: Soft optical tactile sensors with 3d-printed biomimetic morphologies". In: *Soft Robotics* (2018).

[58] Philipp Wu et al. "DayDreamer: World Models for Physical Robot Learning". In: *CoRL*. Vol. 205. Proceedings of Machine Learning Research. PMLR, 2022, pp. 2226–2240.

[59] Kelin Yu et al. *MimicTouch: Leveraging Multi-modal Human Tactile Demonstrations for Contact-rich Manipulation*. 2024. arXiv: 2310.16917 [cs.RO]. URL: https://arxiv.org/abs/2310.16917.

[60] Wenzhen Yuan, Siyuan Dong, and Edward H. Adelson. "GelSight: High-Resolution Robot Tactile Sensors for Estimating Geometry and Force". In: *Sensors* 17.12 (2017), p. 2762.

Appendices

6.1 Appendix A: Changing the End-Effector Orientation

The rotation of an object can be described using various representations. In this work, the matrix representation is used, where each unit vector of the 3x3 matrix corresponds to an axis (x, y, and z) of the rotated coordinate system relative to the base system. In this case, the $\boldsymbol{z}$ vector of the rotation is of particular interest. When the end-effector is not rotated, the $\boldsymbol{z}$ vector points straight down, aligning with the symmetry line of the cylinder. The x-axis points away from the robot toward an observer standing directly in front of the setup, and to form a coordinate system with standard orientation, the y-axis must point to the left from the observer's perspective.

Now, the idea is to influence the orientation of the $\boldsymbol{z}$ vector by using the two additional components in the action vector to adjust the x and y components of the $\boldsymbol{z}$ vector, leading to

$$\boldsymbol{z}_n = \boldsymbol{z}_o + \begin{pmatrix} \theta_1 \\ \theta_2 \\ 0 \end{pmatrix},$$

where $\boldsymbol{z}_n$ is the new $\boldsymbol{z}$ vector, $\boldsymbol{z}_o$ is the old one, and θ_1 and θ_2 are the adjustments applied to the components. After this modification, the new $\boldsymbol{z}$ vector is normalized to a length of one to ensure it remains a unit vector.

Next, the new $\boldsymbol{x}$ vector has to be calculated. To ensure a unique vector, certain constraints are applied. The first constraint is that the $\boldsymbol{x}$ vector stays inside the x-z-plane of the base coordinate system, ensuring that the end-effector does not rotate around the z-axis. This constraint implies that the y component of the $\boldsymbol{x}$ vector remains zero. Additionally, the

angle between the $\boldsymbol{x}$ and $\boldsymbol{z}$ vectors must be $90°$, and the length of the $\boldsymbol{x}$ vector has to be one. These constraints result in the following equation system:

$$0 = \boldsymbol{z}_x \boldsymbol{x}_x + \boldsymbol{z}_z \boldsymbol{x}_z,$$

$$1 = \boldsymbol{x}_x^2 + \boldsymbol{x}_z^2,$$

where the indices determine the component of the according vector, leading to the following solution:

$$\boldsymbol{x}_x = \pm \sqrt{\frac{1}{1 + \frac{\boldsymbol{z}_x^2}{\boldsymbol{z}_z^2}}},$$

$$\boldsymbol{x}_z = -\frac{\boldsymbol{z}_x}{\boldsymbol{z}_z} \boldsymbol{x}_x,$$

and the vector $\boldsymbol{x} = [\boldsymbol{x}_x, 0, \boldsymbol{x}_z]^\intercal$. Since the $\boldsymbol{x}$ vector must still point toward the observer and away from the robot, the positive result for $\boldsymbol{x}_x$ is chosen. With the given $\boldsymbol{z}$ and $\boldsymbol{x}$ vectors, the $\boldsymbol{y}$ vector is calculated by forming the cross product

$$\boldsymbol{y} = \boldsymbol{z} \times \boldsymbol{x}.$$

The matrix formed by the three vectors $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$ defines the new rotation of the end-effector, which can be converted into a quaternion and passed to the robot's movement functions.

To limit the rotation, the angle between the current $\boldsymbol{z}$ vector and the neutral position vector $\boldsymbol{z}_0$ is calculated and constrained to a specified value.

6.2 Appendix B: Hyperparameters

Name	Value	Description
`tactile_shape`	`[64, 64, 3]`	Shape of the tactile observations
`visual_shape`	`[64, 64, 3]`	Shape of the visual observations
`max_steps_per_episode`	`400`	Maximum of steps per episode
`goal_reached_delta`	`0.005`	Absolute distance in all three axes between cylinder and goal to evaluate "success" (in m)
`max_cyl_to_ee_delta`	`0.005`	Absolute distance in all three axes between cylinder and end-effector to evaluate "cylinder lost" (in m)
`episode_velocity`	`0.05`	Relative velocity during the episode (value between 0 and 1)
`workspace_radius`	`0.03`	Radius of the workspace form the center of the base plate (in m)
`workspace_height`	`0.06`	Height of the workspace measured from the surface of the base plate (in m)
`workspace_depth`	`0.05`	Depth of the workspace measured from the surface of the base plate (in m)
`hole_depth`	`0.04`	Depth of the hole (in m)
`gripper_force`	`30`	Force of the gripper (in N)
`gripper_speed`	`0.05`	Speed of the gripper (in $\mathrm{m/s}$)

6.3 Appendix C: Additional Experimental Results and Graphs

This section introduces some interesting additional data that is logged by Dreamer or the environment class. All results refer to the experiments performed with the first upgrade of the setup (see Section 4.1.4). The labels of the different experiments are composed as follows: The letters in the beginning of the label indicate the used modalities ("v" for visual, and "t" for tactile). The number afterward represents the tolerance between peg and hole in [mm].

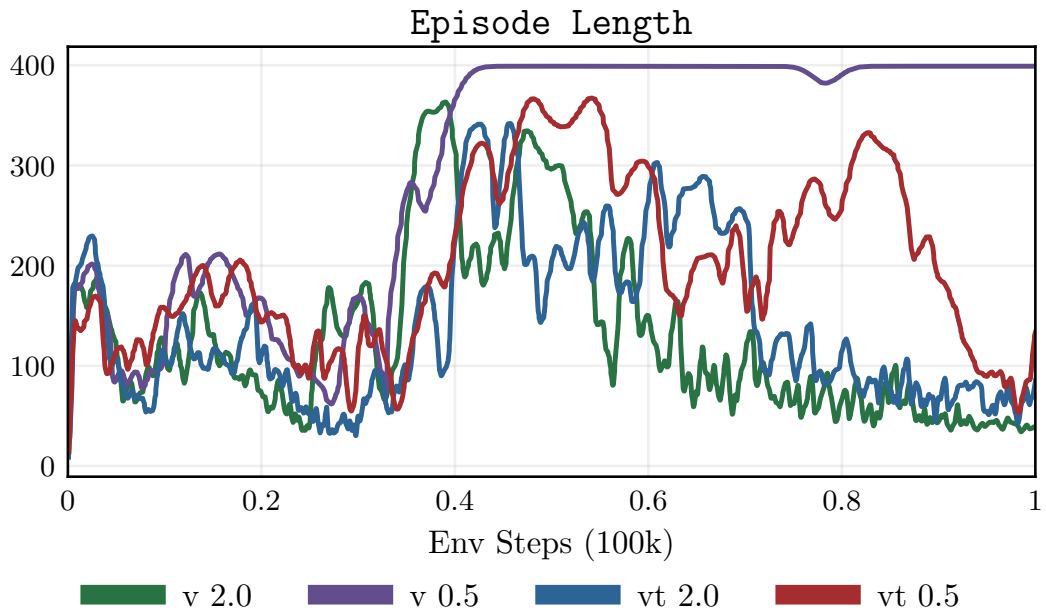


Figure 6.1: Episode length: This figure shows the progression of the episode length of different policies during training. The `v 0.5` policy fails to solve the task and instead learns to remain within the workspace near the hole to maximize its reward. After 400 steps the maximum number of steps is reached. In contrast, the other policies converge to a similar episode length clearly below the maximum.

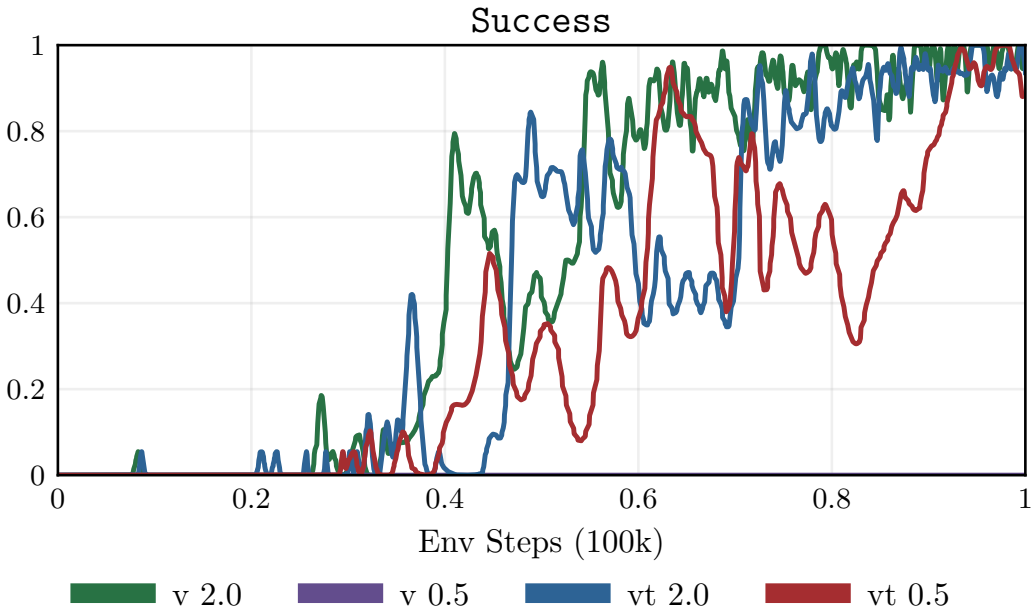


Figure 6.2: Success: The episode success is recorded as either 0 or 1. The curves are smoothed using a running average over the last seven values. It is clear that the `v 0.5` policy fails to solve the task, while the other policies follow similar trajectories, all achieving a success rate close to 1.

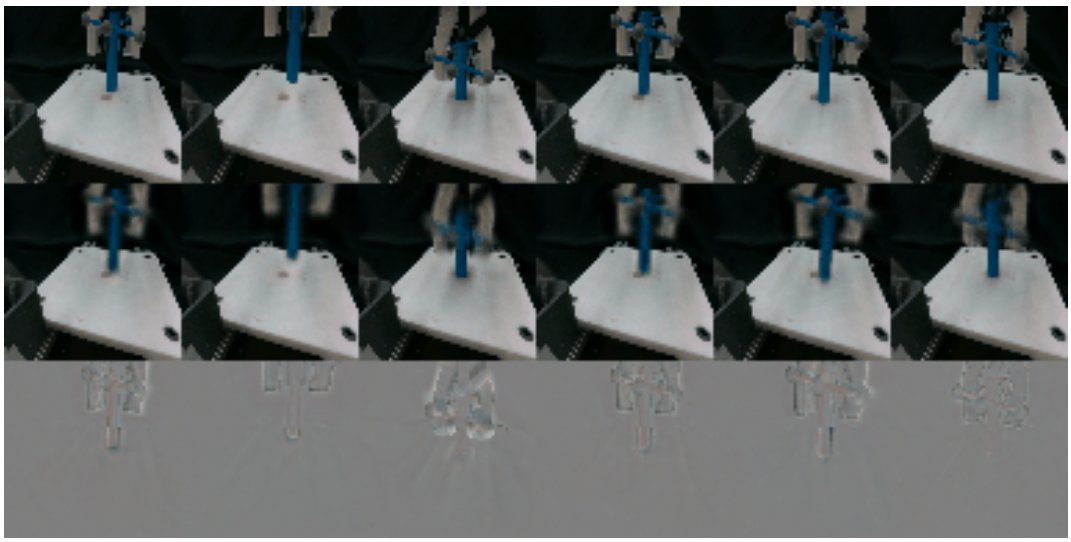


Figure 6.3: Reconstruction of the visual input: The first row presents the real-world observations, the second row displays the reconstructions generated by the decoder network, and the third row illustrates the differences between them. Due to the camera's fixed perspective, variations in the difference images are primarily caused by the movement of dynamic elements in the scene, in particular the cylinder and end-effector.

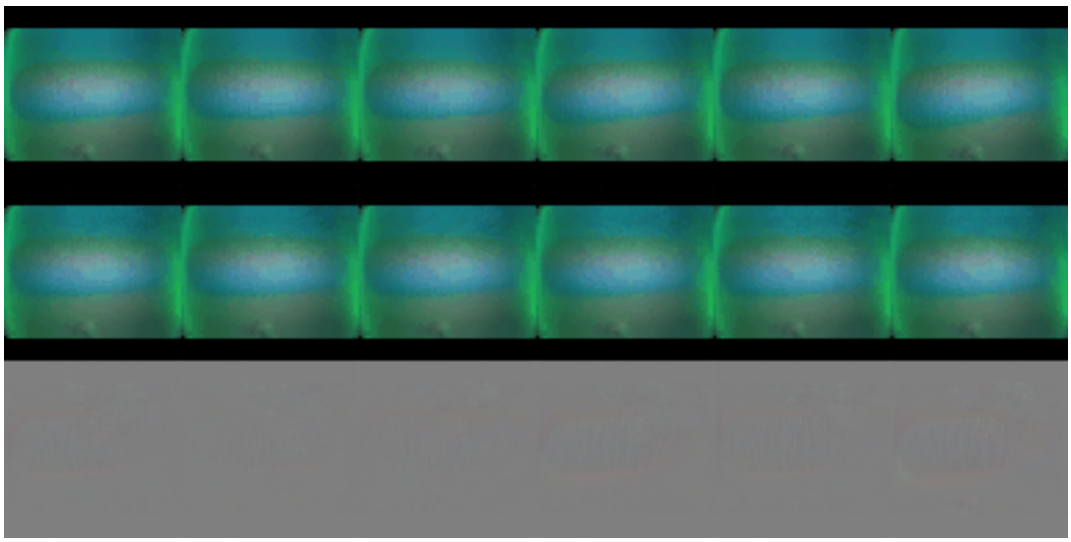


Figure 6.4: Reconstruction of the tactile input: The first row presents the real-world observations, the second row displays the reconstructions generated by the decoder network, and the third row illustrates the differences between them which are very small compared to the reconstruction of the visual input. This is due to the fact that the cylinder does not move much inside the gripper.