Jointly Learning Skill Execution and Domain Grounding for Robot Task and Motion Planning

Master Thesis in Computer Science
submitted to the
Chair of Machine Learning and Reasoning

by
Daniel Maximilian Swoboda

Department of Computer Science
Faculty of Mathematics, Computer Science and Natural Sciences
RWTH Aachen University

Master Thesis in Computer Science

Department of Computer Science
Faculty of Mathematics, Computer Science and Natural Sciences
RWTH Aachen University
Aachen, Germany

Submitted on: March 4th, 2024
Advisor: Dr. Till Hofmann
Examiners: Prof. Hector Geffner, PhD
Prof. Gerhard Lakemeyer, PhD

Abstract

Robotic manipulation in cluttered, dynamic environments is challenging due to the need for high-level reasoning and adaptive skill execution. This thesis explores the integration of task planning and imitation learning to address this challenge. Integrated Task- and Motion Planners (TAMP) have shown to be effective but often rely on extensive geometric modeling, limiting deployment in unknown environments. Conversely, imitation learning from low-level sensory input has made leaps but primarily focuses on short-term skills. Integrating these approaches could overcome their limitations. However, this requires solutions for domain grounding and action-conditioned imitation learning. We propose a transformer-based architecture to address these issues. We present two implementations, OOTAMP and PerTamp, and evaluate them on a custom dataset for symbolically annotated manipulation and domain grounding. Our results suggest that the proposed architectures do not yet meet the desired reliability to tackle long-horizon tasks.

Contents

List of Figures

List of Tables

We are more aware of simple processes that don't work well than of complex ones that work flawlessly.

– Marvin Minsky, 1986

Acknowledgements

My gratitude towards the many people that have been part of my journey to this point is beyond my ability to express myself, yet I shall attempt to do so in these lines. I want to thank Till Hofmann for advising me on my theses, for being open to my ideas, for giving honest feedback, and for his support; Professor Geffner for giving me opportunities to prove myself, for his support, and for asking the right questions; Professor Lakemeyer and the Knowledge-Based Systems Group for the opportunity to learn and grow over more than three years; Tarik Viehmann for his advice, memorable moments on the RCLL TC, and support throughout the years, the members of the Chair of Machine Learning and Reasoning for their questions, feedback, ideas, and jokes; and the former, current, and future members of Team Carologistics.

I want to thank my parents, Petra and Gerald, for supporting me in all my decisions, even though it was hard sometimes; my grandparents Christa and Johann, without whom I wouldn't have made it to university; Frithjof, Daniel, and Maxwell for showing me what camaraderie is; Kevin for his unwavering support; and the many other amazing people who I had the pleasure of sharing parts of my life with, no matter how long or short.

1

Introduction

Factories, kitchens, construction sites, and other real-world environments are cluttered, unstructured, and constantly changing. How can we enable a robot to autonomously and robustly solve long-horizon tasks in these environments? Instead of relying on single, well-defined skills, such a robot system needs to be able to reason about its environment on a task level to plan its actions and on a motion level to execute them.

A particular class of solvers, integrating both task planning and motion planning, can successfully solve long-horizon tasks in robotics. These solvers are integrated Task- and Motion

Planners (TAMP). Since task planning is often expressed logically, and motion planning is a geometric problem, it is usually intractable to solve either one problem using established methods for the other one. Existing TAMP solvers often rely on extensive geometric modeling of the world to compute grasping poses or motion-level constraints [18]. This extensive modeling, which often also requires a certain level of hand-crafting, often makes deploying TAMP solvers in unknown environments impractical.

On the other hand, learning-based approaches for robotic manipulation in reinforcement learning and imitation learning have made significant progress in recent years [38]. Learning-based approaches often directly use low-level sensory input like RGB-D camera streams instead of relying on 3D modeling and are thus able to solve parts of tasks as complicated as cooking meals or cleaning rooms [16]. However, imitation learning mainly focuses on learning to replicate short-term skills, not solving long-horizon tasks.

Considering the strengths and limitations of both task- and motion planning and imitation learning for robotics, it seems that one possible solution to their shortcomings is to integrate them into a single system. A task planner would then solve the long horizon problem purely on a task level, while an imitation learning policy executes the actions from the plan.

In this thesis, we want to work towards solving the problem of integrating imitation learning with task planning to solve long-horizon robot manipulation problems. This problem can be split into domain grounding and action-conditioned imitation learning. First, or a task planner to produce a coherent plan, it needs to be provided with an accurate description of the state of the environment. We refer to domain grounding as extracting this initial state from a low-level observation of the environment. Second, we need a policy conditioned on the plan's actions to translate a task plan into action. A multi-task imitation learning policy can be used instead of a motion planner to provide an approach for the robot manipulator.

While substantial prior work exists both for action-conditioned domain grounding and

the domain grounding problem, there is little focus on solving these adjacent problems in an integrated manner. We present a transformer-based neural network architecture that can be used to determine the truth value of ground predicates given an observation of the environment and to learn an imitation learning policy. Our architecture builds on insights from two recent approaches: in language-conditioned imitation learning [57]; and in predicate grounding [65]. Shridhar et al. [57] show that it is possible to learn a language-conditioned multi-task policy with a perceiver architecture [27]. Yuan et al. [65] extract the truth value of spatial predicates by visual observation of the environment using vision transformers [13]. We present two concrete implementations of our architecture and evaluate it on domain grounding and imitation learning tasks.

This thesis is structured as follows. In Chapter 2, we present related work in imitation learning for robot manipulation, vision-based domain grounding, and integrated task and motion planning. We give an overview of the theoretical background and the techniques we use in Chapter 3. There, we formally introduce task planning, motion planning, and TAMP. We also present the imitation learning problem and provide an overview of machine learning terminology relevant to this thesis and different transformer-based architectures for vision, domain grounding, and imitation learning. In Chapter 4, we analyse the problem of integrating imitation learning with task planning in-depth and introduce our transformer-based architecture, as well as the concrete implementations OOTamp and PerTamp, which we evaluate in depth in Chapter 5. Finally, we summarise the results and identify future work in Chapter 6.

2

Related Work

This thesis is situated within the fields of imitation learning for robotics and domain grounding learning. These adjacent fields have been well studied in previous works, providing a solid foundation for our approach. This chapter will give an overview of related work in imitation learning for robot manipulation and learning-based approaches for planning domain learning and grounding. Additionally, we will cover integrated task and motion planning as an alternative approach to solving longer horizon manipulation tasks.

2.1 Domain Grounding and Domain Learning

Grounding the semantic interpretations of symbols in *reality* is a problem that extends beyond symbolic AI in the form of the *symbol grounding problem* [22]. If we want to apply task planning to solve problem instances for robot manipulation, given the observations available, we must first determine the initial state of our planning domain. This generally implies two tasks: identifying objects and their types within the environment, and evaluating the truth value of the predicates of our domain with regard to these objects. Here, we focus primarily on the second task, which we will call *domain grounding*. We will give an overview of existing approaches for domain grounding in the following part of this section. Furthermore, we will briefly discuss the adjacent problem of learning to create symbolic planning domains from example traces (i.e., *domain learning*).

Domain Grounding

Domain grounding is the subject of active research, especially within the task and motion planning and robot manipulation learning communities [59, 34, 25, 10, 41, 46, 60, 65].

Srivastava et al. [60] and Silver et al. [59] rely on geometric information and a pre-defined mapping function to update their symbolic states. Similarly, Dearden & Burbridge [10] learn to map a geometric description of a state to a pre-defined set of predicates to ground a task-planning problem from an intermediary representation.

Lamanna et al. [41] propose an algorithm that iteratively initialises a PDDL planning domain with known actions and predicates but unknown objects and predicate groundings. Objects are detected based on visual observation, and a pre-trained predictor for each predicate is used to determine the true groundings for each object. Kase et al. [34] also predict the set of true predicates for a provided set of ground predicates from visual observation of the

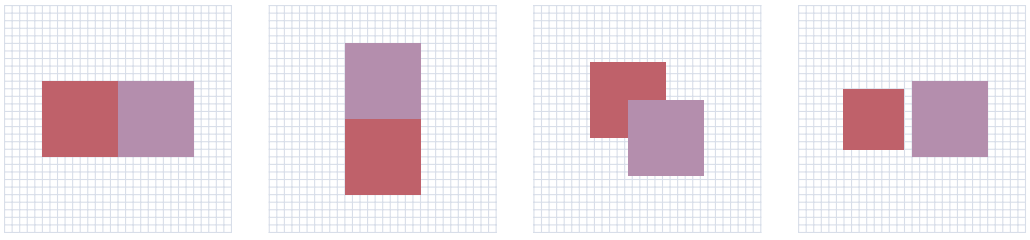


Figure 2.1: A graphical depiction of various O2D states over two objects (red and purple cube) as introduced by Occhipinti et al. [48]. Each O2D state represents an abstraction of the spatial relation between objects, that can be extracted from visual observations.

environment. Huang et al. [25] take this one step further by using a continuous relaxation of a given symbolic domain and learning to ground it from observation by training a classifier for each predicate. Migimatsu & Bohg [46] focus on reducing the training data size for predicate classifiers.

Finally, Yuan et al. [65] use canonical views, i.e., cropped images of objects, an RGB observation of the world, and a vision transformer-based encoder to determine the groundings of spatial predicates. We will introduce this approach in greater detail in Section 3.7.

Domain Learning and Predicate Requirements

Related to domain grounding is domain learning, the problem of creating a symbolic planning domain from observation. Approaches that learn planning domains (or parts of planning domains) from examples of interactions include [37, 40, 17, 3, 48, 2]. Results in domain learning are relevant to our problem insofar as they give us insights into which level of information about the environment is required to plan a task successfully. Since our approach relies on a learned model to determine the ground truth of predicates for planning, it is essential to know which predicates we need to consider. While we will not analyse this problem as part of this thesis, we can look into existing results for some clues: Occhipinti et al. [48] present a framework for learning an abstract planning domain from spatial observations of

a scene. Their goal is to learn feasible planning domains and a grounding function for the predicates in the domain, given a visual observation of the scene. For this purpose, they introduce an intermediary representation of the scene in the form of the language Objects in 2D Space (O2D). It consists of a set of unary and binary predicates describing object properties and spatial binary predicates (`left`, `below`, `overlap`, `smaller`, depicted in Figure 2.1), describing the geometric relationship between objects in the observed scene. Their successful results imply that basic abstract geometric information about the objects in an environment can be sufficient to build domains for planning, even for complex domains like video games.

2.2 Imitation Learning for Robot Manipulation

Learning to control robot manipulators from observation in an end-to-end manner directly, has the potential of being applied to a wide range of tasks [29]. Imitation learning was shown to produce executable policies for manipulation that are able to generalize to unseen tasks or objects [31, 57, 8, 15] by learning from expert demonstrations. This section will discuss various approaches to imitation learning for robot manipulation.

Overview

Early approaches to robot imitation learning focused on replicating single tasks (e.g., picking an object). Multiple learned policies could be combined to solve more complex tasks. Modern approaches are trained on multiple tasks and require some form of conditioning. One potential benefit of learning multiple tasks is that knowledge about object properties can be transferred between tasks [57].

The variety of concrete learning tasks is broad, even within the scope of imitation learning. Common to most approaches is that the input contains an encoding of the environment of

the manipulator (e.g., joint angles, poses, visual observations, etc.) and that the output is some control input for the manipulator (e.g., a goal pose, joint angles, etc.). If multiple skills are learned with the same model, some additional conditioning input might be required to execute the right task policy. Training data then contains examples of environment descriptions and desired output sampled from expert demonstrations.

The range of specific learning tasks within imitation learning is extensive. Tasks can range from simple pick and place operations [56] to things as complex as pan frying meat [16]. To achieve these tasks, the policy takes some observation of the scene as input. This may include information encoding the manipulator's environment (such as joint angles and poses), rich 3d poses of objects, or simple RGB observations. The output then consists of some control input for the manipulator (like goal poses or joint angles). In an imitation learning setup, the dataset $\mathcal{D}$ must use the same kind of perceptual data available to the robot during execution.

When a multi-task policy is represented by a deep neural network, additional input is necessary to execute the correct policy. We call adding this kind of input *conditioning*.

Early Approaches

In an early approach, Ratliff et al. [54] learn a proxy function that assigns scores to possible parameterisations of single actions like grasping based on expert demonstrations. This function is then used to pick actions during execution. Pastor et al. [50] learn differential equations that define the movement of a manipulation action directly from demonstrations.

Deep Learning Approaches

More modern approaches rely on deep neural networks to create action policies that can learn multiple tasks [67, 15, 62]. Zhang et al. [67] first describe how the quality of demonstrations has a significant impact on the ability of a deep multi-task policy network to learn multiple

manipulation tasks from experts. Ho & Ermon [24] introduce an approach for deep imitation learning influenced by Generative Adversarial Networks (GANs). Similarly to GANs, their method consists of a generator and a discriminator. The generator is trained to replicate outputs from demonstration data, while the discriminator is trained to distinguish between demonstration data and generator output. The authors have shown that such an approach can be more data-efficient than comparable baseline approaches. Watahiki & Tsuruoka [62] learn behavioral policies on a sub-skill level instead of full task demonstrations. They identify whether executable pre-learned skills can be applied in a situation or whether a transition between skills is needed by comparing the current state to a task example.

Robust Approaches

The efficiency of imitation learning approaches can suffer from noise in the data. Several approaches attempt to mitigate these issues with varying success. Since a robot's perception differs from that of a human, some approaches tried to adapt their perceptual model such that better policies could be extracted from expert demonstrations [66, 35, 58, 64]. One example is [66], where a policy is learned on a structural causal model instead of learning it directly from vision (or other perceptual input). Kim et al. [35] attempt to reduce the impact of task-irrelevant objects in imitation learning to increase reliability. They train a gaze predictor model from human gaze information coupled with expert demonstrations to predict where a human would look. This predicted gaze is then used to crop the frames to the relevant objects before they are used as inputs to the policy network. Silver et al. [58] learn sub-skills, predicates, and operators from human demonstration by segmenting the demonstration based on mode transitions. Yuan et al. [64] learn to predict stable grasping and placement poses for objects from point-cloud observations to increase skill success using a transformer architecture. Fu et al. [16] improve execution performance in a mobile manipulation setup by combin-

ing data for their target tasks with a general pre-training on pre-existing imitation learning dataset.

Multi-Task Approaches

In approaches that learn multiple tasks with one policy network, it is important to provide some form of conditioning, task description, or goal description to inform the policy. A variety of approaches for goal conditioning in deep imitation learning exist [29, 8, 62, 31, 57], with transformer-based architectures taking a prominent role [8, 57, 26]. Zero, one, or few-shot approaches try to learn a generalisable policy that uses a description of the desired action at run-time as conditioning. James et al. [29] learn task embeddings to represent the desired task in vector space. A distinct control network is trained on expert demonstrations to predict the actions to execute based on an observation of the environment and the task embedding. Dasari & Gupta [8] and Duan et al. [15] learn to perform actions conditioned on visual observation. Here, the context is a set of RGB frames demonstrating the solution for a problem similar to the desired one (potentially solved by a differently embodied agent), and an observation of the current environment. This context is then used to condition a policy network predicting the following action. Instead of just relying on visual information, Jang et al. [31] propose an imitation learning setup that can be conditioned on a variety of task description modalities like natural language instructions or videos. Huang et al. [26] learn to predict a high-level skill label (e.g., pick object) and a low-level policy conditioned on the label, given trajectories of tasks in simulation that solve a long-horizon problem (e.g., cleaning a household). Shridhar et al. [57] learn to predict key poses based on RGB-D observation and a natural language conditioning of the task from human demonstration. The language conditioning aids with policy generalisation towards unseen mixtures of actions and objects. This approach will be introduced in greater detail in Section 3.8.

2.3 Integrated Task and Motion Planners

The imitation learning approaches we discussed in the previous section were focused on learning singular tasks of varying complexity. However, they all rely on some form of higher-level control to solve more complicated, long-horizon tasks. In this work, we want to tackle long-horizon tasks by interfacing a task planner with an imitation learning policy. Integrated task and motion planning provide a well-established alternative.

Garret et al. [18] identify the following general categories of TAMP solvers based on their level of integration of task- and motion-level planning:

- **Sequencing first:** the solver first finds candidate plan skeletons and then attempts to compute a set of constraint-satisfying values for the continuous parameters.
- **Satisfaction first:** the solver first computes a set of constraint-compliant continuous values, and then attempts to find an action sequence (using these values as groundings) that reaches a goal state.
- **Interleaved search:** in this class of solvers, the search is iterative. Actions are added to the plan step-wise, while constraint-satisfying continuous parameters are sampled at each expansion step.

In the following, we will briefly introduce sequencing first, satisfaction first, and interleaved search TAMP solvers to provide an overview of different approaches to solving problems from TAMP domains.

Interleaved Search TAMP Solvers

Interleaved approaches to TAMP [4, 33, 12, 19, 52] switch between motion planning and task planning during the planning process. ASYMOV, introduced by Cambon et al. [4],

might be considered the first integrated task and motion planner [39]. It iteratively decides between pre-computing the geometric environment and lifting geometric constraints into the task level or planning on the task level itself. Kaelbling & Lozano-Perez [33] propose a strictly hierarchical approach that first computes a complete task plan and then computes a trajectory at each plan step. The motion planning is constrained by serialisability with regard to the rest of the motion plan to avoid unnecessary replanning. Dornhege et al. [12] integrate interfaces to functions of the lower-level planner to determine the truth value of certain predicates or the effects of motion-planning actions dynamically during runtime. Garrett et al. [19] build on this idea and formalise it in the form of streams, which are wrappers for generator functions. Plaku & Hager [52] guide task-level planning by using a motion planner to assign utility values to actions such that preferable actions (i.e., those most feasible on the motion level) are preferred.

Satisfaction First TAMP Solvers

Satisfaction first algorithms attempt to guarantee that motion planning-related task-level actions are only executed if they occur entirely within the feasible configuration space at each state. Choi & Amir [5] extract actions from a pre-computed motion-planning tree such that the high-level planner operates only with actions that are guaranteed to be feasible. Agostini & Piater [1] split objects into functional components and compute constraints from their geometry to encode them in the task-planning space directly.

Sequence First TAMP Solvers

Sequence first TAMP solvers [60, 9, 7] pre-compute on the task level. However, they usually require information from the motion planning level to compute feasible plans. Srivastava et al. [60] integrate off-the-shelf task and motion planners through an intermediary abstrac-

tion layer that lifts geometric information into the task level using logical predicates sampled from a motion planner. de Silva et al. [9] follow a similar approach and use pre-computed grasps and predicates grounded in motion planning in an HTN planner. Dantam et al. [7] introduce a probabilistically complete approach based on incorporating incremental constraint solving to guide the task planning. Iteratively, the planner computes a plan on the task level before computing a corresponding plan on the motion level for each action. If the motion planner fails to find a solution, constraints are added to the task planner, and the plan is recomputed.

Machine Learning and TAMP Solvers

Machine learning approaches have been integrated into TAMP in the form of guides or heuristics for search [14, 32]. Driess et al. [14] use a prediction network as a heuristic for a task and motion planner such that action sequences that are expected to lead to feasible motion plans are prioritised. Jiang et al. [32] use a RL-like approach to improve the conditioning of the motion planning part of a task and motion planner, such that more optimal trajectories are being computed.

Some authors also view TAMP as a whole as something that can be formulated as a machine learning problem [47, 6]. Newaz & Alam [47] formulate TAMP as a reinforcement learning problem where the objective is to learn optimal policies on both the control level (i.e., the motion planning level) and on the policy level (i.e., the task planning level). Dalal et al. [6] use TAMP to produce large datasets of sample problem solutions for a supervised learning approach. These datasets are used to train a transformer-based policy that, given a visual representation of the current state and limited state history, computes a motion action towards the desired state.

3

Background

This section aims to familiarise the reader with the foundational concepts, terminology, and notation used in the subsequent parts of this thesis. These are task planning (or classical AI planning), motion planning, combined task and motion planning (TAMP), machine learning terminology, imitation learning, and transformer architectures for applications in robotics. Finally, we will introduce SORNet [65], an approach for domain grounding, and Perceiver-Actor [57] an imitation learning approach. Both are foundational to this thesis, as we build on their architectures and take inspiration from both for our approach.

3.1 Task Planning

Task planning, also called classical (AI) planning, describes the problem of getting a system from an initial state to a goal state by applying actions whose effects are deterministic and known [20]. The states and transitions of an environment can be represented as a graph. Therefore, we can formulate the planning problem as a search problem over this state graph. We will follow Geffner & Bonet [20] and model classical planning problems as a state model $S = \langle S, s_0, S_G, A, f, c \rangle$ where

- S is a finite set of states,
- $s_0 \in S$ is the initial state.
- $S_G \subseteq S$ is the set of goal states,
- A is the set of actions, with $A(s) \subseteq A$ the set of states applicable in a state $s \in S$,
- $f(a, s)$ is a deterministic transition function that maps the current state s to a state s' that follows after executing action $a \in A(s)$,

Our a goal is to find a series of actions $a_0, \ldots, a_n$, that generates a state sequence $s_0, s_1, \ldots, s_{n+1}$, where $s_n \in S_G$ [20]. We call this series of actions a solution or plan. State-of-the-art planners make use of heuristic functions to find these plans by directing the search in the state graph toward the goal state(s).

To encode a planning problem for a planner, we need some language to express it [20]. In the case of discrete planning problems, we can make use of the language STRIPS, which is based on boolean variables. Coincidentally, STRIPS was originally developed in the context of robotics for the robot Shakey. A planning problem in the STRIPS language is a tuple $P = \langle F, I, O, G \rangle$ where

- F is the set of atoms/propositions of interest

- O represents the set of actions, each $o \in O$ is comprised of the sets $Add(o)$, $Del(o)$, and $Pre(o)$ of atoms over F,
- $I \subseteq F$ is the initial state,
- $G \subseteq F$ is the goal state.

The set $Pre(o)$ describes the preconditions of action o, or the atoms that must be true in the state for an action to be applicable. The sets $Del(o)$ and $Add(o)$ are the delete and add lists of o. After o is executed, the atoms in $Del(o)$ will be removed from the current state (and thus made false), and atoms in $Add(o)$ will be added to it (thus made true). STRIPS encoded problems are easily converted into the state model. $s \in S$ are the collections of atoms over F under the closed-world assumption, F matches s_i, and S_G is the set of states $s \in S$ for which $G \subseteq s$. The actions $a \in A(s)$ in the state model are the ones in O with $Pre(a) \subseteq s$. Finally, we can define the state transition function simply as $f(o, s) = (s \setminus Del(o)) \cup Add(o)$.

The Planning Domain Definition Language (PDDL) was introduced as an extendable superset of STRIPS [45]. In this work, we will use basic PDDL (version 1.2) encodings for our problems. A planning problem in PDDL expresses the preconditions of actions and action arguments in the form of predicates with variables and constants over a set of objects instead of using atoms. However, since this set of objects is fixed and finite for a specific planning problem, we can assume for simplicity that it is converted into an equivalent problem using only atoms.

3.2 Motion Planning

Unlike task planning, motion planning is an inherently non-discrete geometrical problem. The configurations of a robot with d degrees of freedom can be represented by a d-dimensional configuration space over the real numbers [18]. Each non-restricted point in a configuration

space is a valid state for the robot, i.e., the robot's joints can assume any point that is non-restricted. Planning a motion for a robot can, therefore, be framed as finding a non-restricted and continuous trajectory from a starting point to a goal point.

Following the formalism by Garrett et al. [18], we can express the problem in the following way. Let

- $Q \subset \mathbb{R}^d$ be the configuration space of our robot,
- $F : Q \to \{0, 1\}$ be a constraint function, usually a combination of joint and space constraints,
- $Q_F = \{q \in Q | F(q) = 1\}$ be the feasible configuration space satisfying F,
- $q_0 \in Q_F$ be the initial configuration,
- $G \subseteq Q_F$ be the set of goal configurations.

A motion planner's objective is to find a continuous path $\tau : [0, 1] \to Q$ such that $\tau(0) = q_0$, $\tau(1) \in G$, and for all $\lambda \in [0, 1]$, $\tau(\lambda) \in Q_F$ [18]. In other words, it needs to find a continuous path that does not violate any of the constraints modeled by the constraint function F. Constraints in motion planning can come from several sources, like invalid configurations for the specific robot, (potentially dynamic) physical constraints in the environment, or additional safety constraints. Commonly, approaches rely on sampling or trajectory optimization [18]. An example of a sampling-based algorithm is RRT [42], which is probabilistically complete, i.e., its probability of failing to find a path if one exists converges to zero. In this work, we assume that there is a complete and correct motion planner for our subset of problems.

3.3 Task and Motion Planning

Combined task and motion planning (TAMP) refers to integrating a high-level task planner and a low-level motion planner [18]. This combination allows the creation of autonomous

robot systems that are capable of solving more complex and longer-horizon tasks than if these tasks are treated separately.

While the approach presented in this thesis does not follow a classical TAMP scheme, the core problems are closely related. Both our approach and classical TAMP solvers attempt to integrate task planning with low-level motion actions executed on a robot. Therefore, we briefly introduce the core idea of combined TAMP for context.

The integration of task planning and motion planning is not trivial. As discussed before, task planning problems are usually expressed through some formal logic. Motion planning problems, however, are geometric.

Garret et al. [18] show how representations from task planning can be extended to describe TAMP problems. Like in task planning, a solution to a TAMP problem is a sequence of actions $p = \langle a_1, a_2, ..., a_n \rangle$, with associated states s_1 through s_n where for a goal state g, $g \subseteq s_n$ [18]. However, each action $a_i \in p$ now must include ground values for all parameters (including the continuous ones), s.t. the action's constraints are satisfied. If an action sequence contains unground continuous parameters, we call it a plan skeleton.

The search for an action sequence lends itself to be modeled as a tree or graph search problem due to its discrete nature. Meanwhile, the search for continuous parameters is a constraint satisfaction problem, usually through optimization or sampling. The question, then, is how can we combine solvers for both subproblems into an integrated TAMP algorithm? Ideally, we would like to organize our search such that computational complexity is minimized, e.g., by pruning infeasible branches. In Section 2.3 we gave an overview of several approaches and their way of solving the integration problem. Our approach does not perform TAMP, but it can be compared to a sequencing-first TAMP solver in the sense that we want to compute a task plan first, and then execute its actions.

Recent work has demonstrated the viability of imitation learning as an efficient way to program skills for robot systems [49]. In robotics, the goal of imitation learning is to learn a policy that performs specific tasks by reproducing the behavior of experts. Expert behavior is usually provided as demonstrations from human experts or collected algorithmically.

Formally, let us assume that the expert demonstration can be represented as a sequence of features called trajectory $\tau = [\phi_1, ...\phi_T]$ with T elements [49]. The features ϕ_i can be various observations, measurements, or other information representing the robot state at a given time step. Multiple trajectories are collected in a dataset $\mathcal{D} = \{(\tau_i, s_i)\}_{i=1}^{N}$ of size N, where each trajectory τ_i is associated with a context s_i. The context s_i can contain any relevant information for the task demonstrated in trajectory τ_i, such as labels, objects in the scene, initial positions, etc. An imitation learning algorithm aims to learn a policy π^* satisfying

$$\pi^* = \arg\min D(q(\phi), p(\phi)),$$

where D is a distance measure, q is the distribution over the features induced by the expert policy, and p is the distribution of features induced by the learner [49], i.e., an optimal policy π^* is one with minimal distance to the expert policy.

From this formulation, we can already see the critical limitation of classical imitation learning. The performance of policy π^* is limited by the performance of the provided expert demonstrations and can generally not surpass demonstrations during execution [38].

A key advantage of imitation learning approaches is that it is generally considered easier to provide an expert demonstration of a task than to program it [49]. Another key advantage of learning-based acquisition of skills is that explicit geometric models are generally not re-

quired, but low-level observations might suffice [49]. Instead, implicit models are acquired directly during the learning process from low-level observations like RGB images and depth data collected from the scene. Motion planning, on the other hand, generally needs some explicit model to perform the necessary computations [18]. However, this is not limited to imitation learning but applies to other forms of skill learning, such as reinforcement learning.

3.5 Machine Learning Terminology

This thesis will refer to several concepts from machine learning and deep learning. To familiarize the reader with certain terminology (which might have varying definitions in literature), we will briefly introduce and define important terms in this section. However, we assume basic (though not technical) familiarity with deep learning concepts like function approximation and back-propagation. Since some notations and symbols are not used uniformly across literature, we will use the following denotations: $\frown$ is a matrix concatenation, $+$ is a standard matrix addition, $\cdot$ is standard matrix multiplication, $\otimes$ is a tensor product, $*$ is a convolution.

Feedforward Networks

Feedforward neural networks, or multilayer perceptrons (MLPs), are a basic form of a (deep) neural network [21]. They define a mapping $y = f(x, \theta)$, where θ is a set of (partially) learnable parameters. Commonly in feedforward networks, f is a chain of linear models (perceptrons), i.e., $f(\theta, x) = f^{(1)}(\theta^{(1)}, f^{(2)}(...f^{(n)}(\theta^{(n)}, x)))$. In this model, $\theta^{(i)} = (w, b)$ where w is a vector of learnable weights, b is an additive bias and $f(x, \theta) = x^T w + b$.

In the PyTorch framework [51], the linear model is referred to as a linear layer, implemented in the module `nn.Linear`. Using this implementation, we use the term linear layer

to refer to one layer of a deep neural network.

Convolutional Layer

A convolutional neural network layer uses a convolution operation [21] to process multi-dimensional tensors as inputs. Convolutions are common in image processing as they allow to efficiently reduce the dimensionality of image data (which is usually a matrix of the form (H, W, C) with $C = 3$ for RGB images, $C = 4$ for RGB-D images), allow interactions between different parts of the image, and allow us to work with variable size input.

A convolutional layer consists of an input vector (e.g., an RGB image), a kernel vector (usually of smaller dimensionality) with learnable parameters, and a stride value. The idea behind a convolution is to use the learnable kernel to extract relevant local features from the patches. The operation induces a structural prior in the architecture, which is especially beneficial for vision tasks. Mathematically we can express a 2D-convolution operation $I * K$ on input I and kernel K for indices i, j (defined by the stride) as a cross-correlation function [21]:

$$(I * K)(i, j) = \sum_m \sum_n I(i + m, j + n) K(m, n)$$

The kernel is passed over the input sequentially, depending on the stride value to extract so-called patches. For example, a stride value of 3, a kernel size of $(3, 3)$, and an input vector of dimensions $(3, 6)$ leads to two patches being extracted as shown in Figure 3.1.

Attention Layer

Attention is a term used to describe a function that maps a vector called a query Q and a set of key-value (K, V) pair vectors to an output vector [61]. Input is usually in the form of a sequence of vectors (called tokens). The output is a weighted sum (average) of the values,

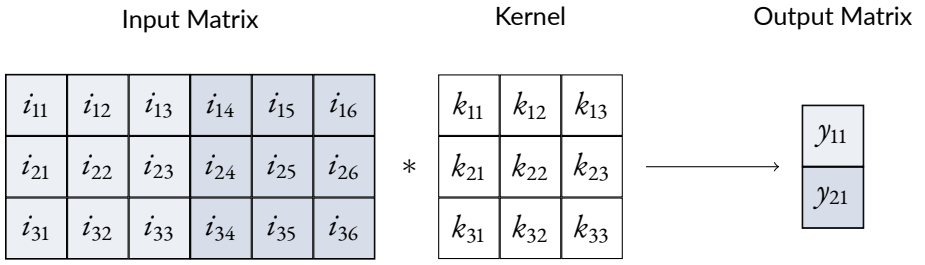


Figure 3.1: A graphical depiction of a 2D convolution with a kernel of size $(3, 3)$, an input of size $(3, 6)$, and a stride value of 3. This results in two patches being extracted via the kernel and thus an output of size $(1, 2)$.

where the weight is a trainable parameter. Intuitively, an attention mechanism enables a network to learn the relevance of each token in the sequence pair (K, V) relative to the sequence Q, which allows the module to extract contextual information from the input vectors. If K, V, and Q are identical, it is called self-attention. If K and V are the same, but Q is different, it is called cross-attention. One implementation of attention is scaled dot-product attention as depicted in Figure 3.3.

The concept of attention can be extended via so-called multi-head attention (depicted in Figure 3.2). Using multiple linear layers as pre-processors combined with multiple attention layers (called attention heads) in parallel, each attention layer can extract different properties from the input, which can then be concatenated into a new latent embedding.

Latent Space and Embeddings

Embedding refers to a vector created by a (hidden) neural network layer, given an input [21]. They represent the input data more abstractly, containing information relevant to the following neural network layers. The space from which an embedding (or other latent variable) is drawn is generally referred to as latent space, typically a sub-space of $\mathbb{R}^n$.

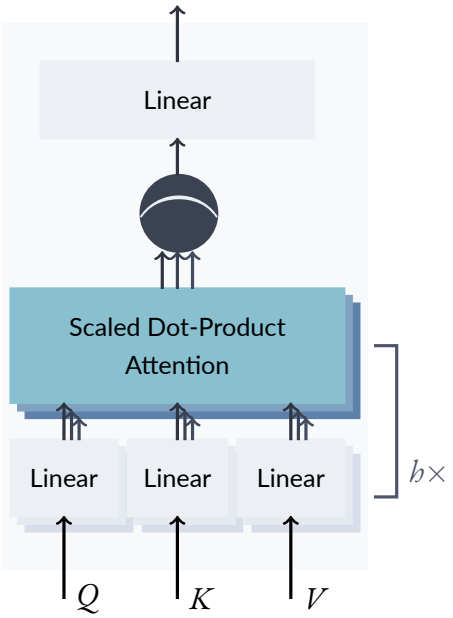


Figure 3.2: Multi-head attention using scaled dot-product attention as used in the transformer architecture [61].

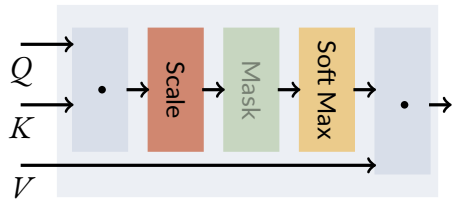


Figure 3.3: Scaled dot-product attention as used by the transformer architecture [61].

Encoders and Decoders

The terms *encoder* and *decoder* as we use them come from work on sequence-to-sequence recurrent neural networks [21]. Originally, an encoder (reader) is a neural network that processes an input sequence X and emits an embedding or context C. The embedding is then fed as input to another neural network called a decoder (writer), which produces an output sequence Y, which might be of a different length.

In this work, we will call any network that takes an input sequence and produces an embedding (either a single vector or a sequence of vectors) an encoder. Every input creating an output vector or sequence of vectors from an embedding will be referred to as a decoder.

Positional Embeddings

Unlike convolutional layers or recurrent layers (which we do not discuss here), attention layers process the input sequence in a way that is invariant to the order of its elements [61]. The order of elements in the input does, however, carry important information for many types of data. In the case of language processing, the order of words of a sentence can change the meaning of the sentence. Similar reasoning applies to the position of patches in images.

To preserve this positional information, several forms of positional embeddings (or encodings) have been suggested [61, 28, 13]. Positional embeddings are usually the same size as one input token (i.e., a vector within the input sequence). Generally, these are just unique vectors for each position we want to encode that are added to the vectors of our input sequence (either through element-wise addition or concatenation).

Transformers

Transformer [61] is a flexible architecture for building deep neural networks that have become the de-facto standard in natural language processing [13, 28], due to their ability to scale to very large numbers of parameters.

A standard transformer encoder (depicted in Figure 3.4) takes a sequence of one-dimensional tokens as input [61]. Since transformers are generally invariant towards the position of an element within the input sequence, learned positional embeddings are added to each input token to provide additional information in the input [28]. The tokens are passed sequentially through a series of encoder blocks of length L. Each block consists of h scaled dot-product attention layers whose results are concatenated. The resulting embedding of the sequence can be used with various output layers. A decoder architecture might be used to query embeddings and create output sequences [61, 27]. Alternatively, the embedding might be used

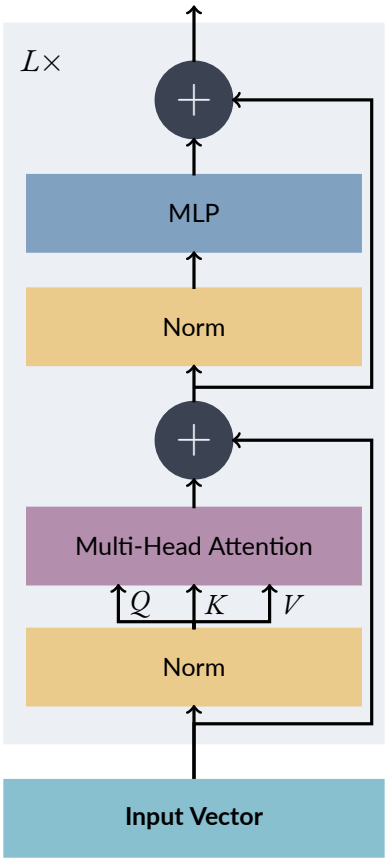


Figure 3.4: A diagram of a standard transformer encoder [61]. The hyperparameters L and h determine the number of attention heads (h) and the depth of the transformer encoder L. Input is expected to be embedded in one-dimensional vectors and re-labeled as V, K, and Q values. Diagram based on Vaswani et al. [61].

with simpler, task-specific layers like classifiers [28, 13].

3.6 Transformer Architectures for Vision

The ability of transformers to scale well to larger numbers of parameters makes them of interest for other areas of deep learning, such as vision and, as a consequence, robotics. However, the standard transformer architecture relies on a self-attention mechanism, which computes the product of every element in the input vector with every other entry, leading to quadratic

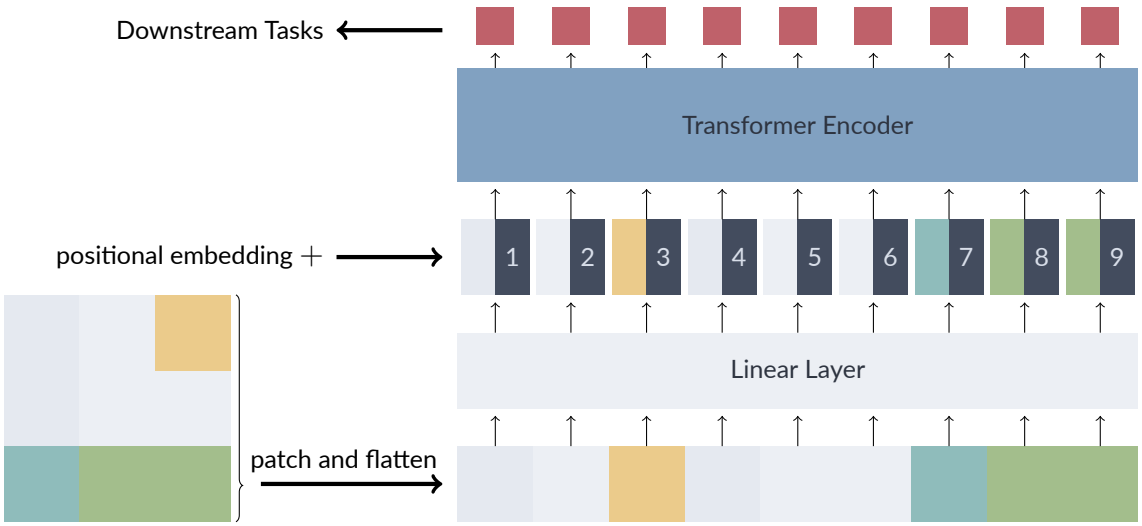


Figure 3.5: The encoder architecture of a vision transformer as introduced by Dosovitskiy et al. [13]. The input picture (bottom left) is first split into patches and flattened. A linear projection is applied onto the flattened patches to get the desired token size. A positional embedding is added to each of the input tokens. The result is fed through a standard transformer encoder, resulting in embeddings for downstream tasks.

cost in the input size [13] This mechanism does not scale well on input with high dimensionality, like (flattened) multi-channel images.

To deal with this shortcoming, several approaches adapting transformer architectures for other input modalities than text have been introduced. Here, we will consider the works of Dosovitskiy et al. [13] as well as the continued works of Jaegle et al. [28, 27]. They represent two distinct approaches to dealing with large inputs with a transformer architecture, achieving similar results on established vision benchmarking tasks like the ImageNet benchmark [11].

Vision Transformer

Dosovitskiy et al. [13] introduce vision transformers, a pre-processing pipeline to make unmodified transformer encoders compatible with image data. creating embeddings of images that task-specific layers can use for downstream tasks such as classification. To solve the pre-

viously discussed scalability problem of transformers for large input sequences, they use a reshaping mechanism that significantly reduces the dimensionality of the input picture.

Figure 3.5 gives an overview of the vision transformer architecture. While the vision transformer makes use of an unmodified transformer encoder, several preprocessing steps are required. Each input image (usually of dimensionality $(H \times W \times C)$) is first split into a sequence of N patches of dimensionality $P \times P$, where P is a hyperparameter [13]. Each of the N patches is reduced to a token vector of size D by reshaping it into a vector of dimension $P^2 \cdot C$ and then linearly projecting it to dimensionality D. A (unique) positional embedding (also of size D) is added to each token to retain information on the token's position within the image. The tokens with added positional information make up the input sequence to the standard transformer encoder.

They evaluate their architecture on classification tasks using a simple MLP with one hidden layer operating on the embeddings from the transformer layers [13]. The authors note that medium-sized datasets lead to performance below that of ResNets [23] of a similar number of parameters as their transformer architecture. The authors argue that this might be due to missing inherent structural biases of the transformer architecture compared to CNNs, even though positional embeddings should somewhat mitigate these effects. The effects of the missing structural bias can be mitigated through pre-training on large datasets (up to 300M images). A pre-training regiment can lead to the same network achieving state-of-the-art like performance on the ImageNet classification task [11].

Perceiver

The perceiver architecture, introduced by Jaegle et al. [28], presents an alternative approach to dealing with the input size problem of transformers. Unlike vision transformers, which reduce the length of the input sequence through modality-specific reshaping, perceivers do not

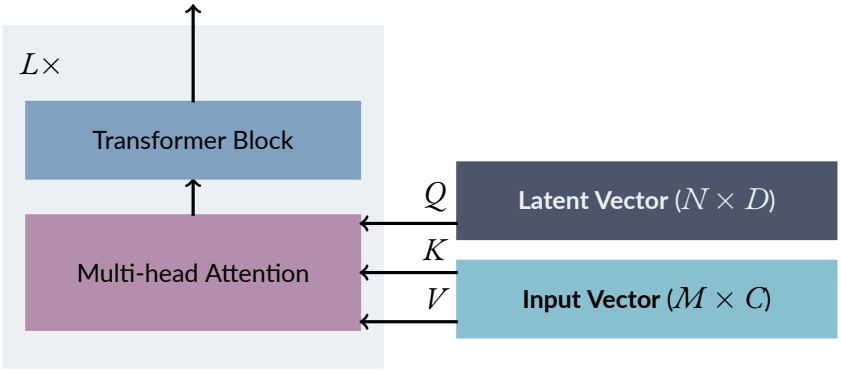


Figure 3.6: The perceiver encoder architecture as introduced by Jaegle et al. [28]. A multi-head attention block (applied as cross attention) is used to extract information from a large input token sequence. This information is then projected into latent space (with lower dimensionality), and then passed to a standard transformer. To iteratively extract information from the input, the sequence of cross attention and transformer blocks can be repeated. The set of latent vectors is randomly initialised and trained with the rest of the network parameters.

perform self-attention directly on the input sequence. Instead, they extract information from the input sequence into smaller-sized latent vectors through a more efficient cross-attention mechanism. This integrates the reshaping procedure directly into the transformer architecture.

In the perceiver architecture (depicted in Figure 3.6), the encoder consists of a stack of alternating cross-attention layers and transformer blocks [27]. A cross-attention layer attends the input with a fixed set of learnable latent vectors of fixed size, which serve as an internal representation of the input data but with lower dimensionality. Multiple cross-attention blocks can be used to extract information from the input sequence iteratively. The unmodified transformer blocks only operate on the latent vectors. Due to their much smaller size than the input sequence, the computational complexity is reduced. Like in standard transformers or vision transformers, a positional encoding (either based on scalable Fourier features or learned) can be used to preserve spatial information of the input sequence. However, unlike with vision transformers, large input sequences like flattened RGB images might be used directly.

The first iteration of the perceiver architecture [28] was applied on classification tasks like ImageNet [11], again leading to state-of-the-art performance. In a later iteration [27], the architecture was extended with a decoder mechanism that uses learnable queries, such that task-specific outputs can be extracted from the embeddings without the need for task-specific decoders. This idea is similar to the decoder of the original transformer architecture [61].

3.7 Grounding Spatial Predicates

In Section 2.1 we discussed the domain grounding problem with a special focus on spatial predicates. The problem entails determining the truth value of a given ground predicate (e.g., an atom in STRIPS), and a visual observation of the environment. A domain grounding model can be used to determine the initial (or any intermediate) state of a planning problem, to create a task planning instance.

SORNet

Spatial Object-Centric Representation Network (SORNet), introduced by Yuan et al. [65], achieves state-of-the art performance on a spatial predicates classification task. Their architecture makes use of a vision transformer-based encoder [13], that is pre-trained on a large, generated dataset from a robotics simulation. The encoder is combined with a task-specific decoder network, to solve various downstream tasks like spatial predicate classification, relative direction regression, and compositional generalisation. In their approach, the same decoder architecture is used for every task, only varying the number of outputs in the final layer.

The SORNet encoder takes as input an RGB image of the scene and a set of reference images (referred to as canonical views) of objects as a query and produces a set of latent embeddings for each of the objects in the scene. This is unlike standard vision transformers,

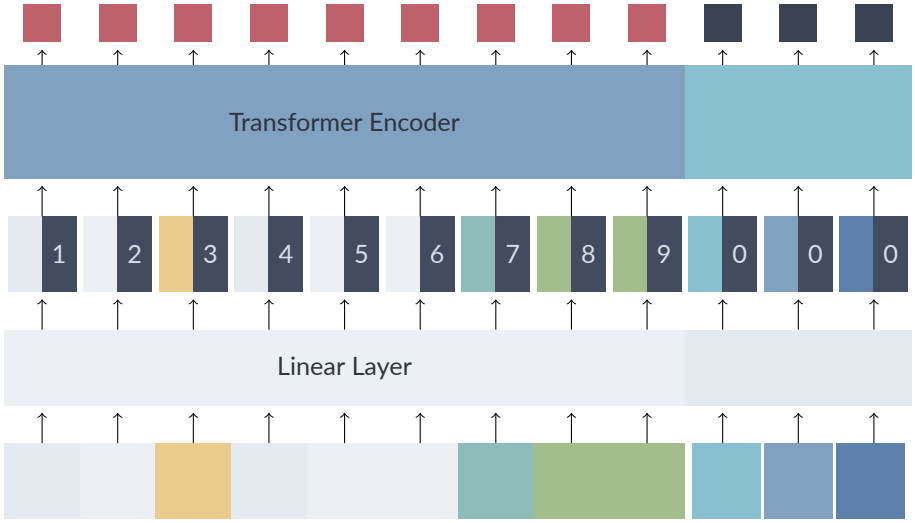


Figure 3.7: A diagram of the encoder architecture used by SORNet as presented by [65]. SORNet builds on the standard vision transformer by modifying the input sequence. Besides an observation of the environment, which is processed identically to vision transformers, a set of images of objects is passed. These object images (called canonical views, represented by the right three tokens here), are passed as part of the input sequence, and processed the same way the scene is processed, but each token corresponding to a canonical view shares the same positional embedding. After being fed through the transformer encoder, the embeddings corresponding to the objects are used for downstream tasks.

where the input consists of only one image. However, since the sequence length of a transformer is variable, the architecture is not changed. The latent embeddings created from an object's canonical view are then used as input for downstream tasks. In the case of spatial predicate classification, SORNet has a distinct classification decoder for each predicate in their training set and train them together with the encoder. Consider, a classifier network for the binary spatial predicate `left` over objects o_1 and o_2. The input of the classifier consists of the embedding vectors of the objects o_1 and o_2. The output is then the predicted truth value of the predicate given these embeddings. Since only the latent vectors for the objects are passed on to the task-specific classifiers, they must contain information of the scene and the spatial information of objects within it to produce an accurate result.

The architecture of SORNet (depicted in Figure 3.7) consists of three main parts:

- A linear projection layer takes as input an image of the scene and the canonical object

views. First, patches are extracted using a convolutional layer. Then the patches are flattened and projected to the target input token size using a linear layer. Finally, each token is concatenated with a learned positional embedding. Each token from the image of the scene is concatenated with a unique positional embedding. The positional embeddings for canonical view tokens are identical.

- In the second step, like with vision transformers, an unmodified transformer encoder is used to process the input tokens. This results in a set of latent embeddings, corresponding to the number of input tokens. The embeddings for the scene are discarded and the embeddings for the canonical views are then used for further downstream tasks.
- Finally, a task-specific decoder (called readout network by the authors) is used to produce the desired output. In the case of SORNet, the decoder consists of a 2-layer MLP which is fed with the canonical view embeddings relevant to the query and, e.g., acts as a binary classifier for a predicate.

The embeddings yielded from SORNet's encoder were proven to contain continuous spatial information in addition to the abstract information relevant to predicate grounding, even without fine-tuning of the encoder. This was demonstrated by training a simple 2-layer MLP as a regressor network, predicting distance vectors between queried objects, using an encoder trained on the predicate classification task.

SORNet was trained on different datasets, depending on the task [65]. The main task relevant to us – spatial predicate classification – used their dataset (leonardo), which consists of 130k sequences of a block stacking task with 4-7 blocks of randomized colors (out of 954). Overall, more than 300k observations are included in the dataset. The authors report their results training on only 10k sampled sequences.

SORNet's use of visual references to create object embeddings for downstream tasks was a major inspiration to the approach of this thesis. Furthermore, one of our implementations will be built around a pre-trained SORNet encoder.

3.8 Skill-Learning via Imitation Learning

Imitation learning of skills (i.e., actions that a robot can perform) is a sample efficient method for creating policies for solving manipulation tasks [49]. Multi-task policies allow us to execute multiple skills, using a single model. Such approaches can benefit from transferable knowledge, regarding objects and their properties [56]. The problem in multi-task imitation learning is to predict a control input (e.g., a pose, a set of joint angles, etc.) for a robot manipulator, given a skill conditioning and an observation of the environment.

Perceiver-Actor

Perceiver-Actor (PerAct) is a multi-task imitation learning model for robot manipulation [57]. PerAct makes use of natural language embeddings from a pre-trained CLIP model [53] to provide task conditioning for multi-task imitation learning. Language encodings and scene observations are processed by a perceiver [27] encoder.

PerAct takes as input an embedding of a natural language description of the task to perform, and an observation of the scene in the form of voxels. From this input, the model produces per-voxel embeddings that are used to predict the next best discretised action for the robot manipulator to perform, to achieve the given task. Each discretised action consists of a translation (given by a target voxel in the input grid), a rotation (discretised into 5 degree steps per rotation axis), and a gripper state (open or closed). Each of the components is predicted separately from the embeddings.

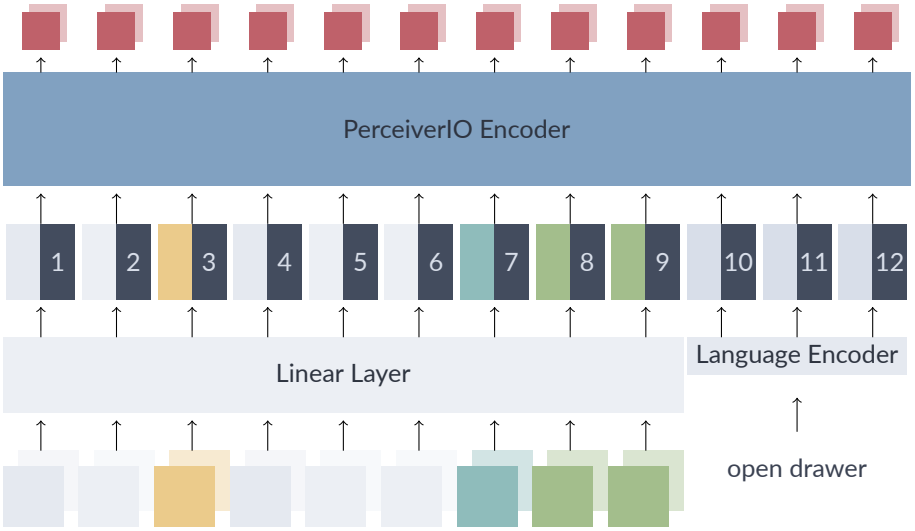


Figure 3.8: A diagram of the encoder architecture used by PerAct as presented by [57]. A natural language description of the task is embedded using a pre-trained encoder. Patches are extracted from the input voxel grid using a convolutional layer, then flattened and scaled to the desired token size. Learned positional embeddings are added to each token. Finally, a perceiver encoder is used to generate per-voxel embeddings, which are then further processed to yield the robot manipulator's target pose.

The architecture of PerAct (depicted in Figure 3.8) consists of 4 parts:

- A pre-trained language encoder from CLIP [53] is used to convert a natural language instruction into a sequence of tokens.
- A convolutional layer is used to extract patches of voxels from the observation of the scene. Each voxel patch is a cube consisting of several voxels. The patches are then flattened and linearly projected such that their dimension fits the dimension of the tokens from the language embeddings.
- Learned positional embeddings are added to each token. The tokens are then fed through a perceiver transformer [27].
- The embeddings from the perceiver corresponding to voxel patches are decoded into the voxel space again by a decoder layer. The final, per-voxel embeddings are then used for the pose prediction task.

The architecture generally does not perform any modifications, as Perceivers are already multimodal by design. However, the authors note that their voxel-patch approach is similar in style to image patches in vision transformers [13].

For training, 18 tasks from the RLBench [30] benchmarking suite with 100 demonstrations each are used. During training, the authors make heavy use of perturbations (translational, rotiatonal shifts of the data) to further increase the density of observed traces. Additional features are added to each voxel like its grid position and coordinate in the robot manipulator's frame, to further induce inductive bias.

PerAct uses an iterative, discretised form of imitation learning, where each action to be learned is split into several sub-goals, referred to as *keypoints*. The authors frame their imitation learning problem as: predict the next keypoint, given the current observation and language-conditioning. The translation between the current pose and the predicted key pose is then realised via a motion planner. To give an intuitive understanding of this approach, we will give a brief example. Consider an action, conditioned by the language goal *pick up the yellow cube*, and an observation of a desk with a yellow cube. The action can be split into several parts: approaching the cube, grasping the cube, and lifting the cube. Each point at which there is a switch from one sub-action to the next can be considered a keypoint. In our example, that means that if the observation shows that the robot arm has already approached the cube but not grasped it yet, the next keypoint to predict is the one where the cube is grasped. PerAct performs this splitting automatically during the creation of the dataset by splitting at points where the delta in the end-effector pose is below a certain threshold. Intermediary positions are collected as well, to increase the observation density in the dataset.

With Perceiver-Actor, Shridhar et al. [57] showed the feasibility of using transformers for multi-task imitation learning. Our thesis builds on their results by using their formulation of the imitation learning problem. Additionally, one of our concrete architectures is inspired

by PerAct's use of the perceiver.

4

Approach

This thesis approaches the problem of solving longer horizon robot manipulation tasks by integrating task planning with imitation learning as an alternative to integrated task and motion planning. Our idea is to learn a *domain grounding* model to lift information from the observation into the task planning level and to learn an *action policy* that can be conditioned on PDDL action names and object parameters to execute a task on the motion level. We will evaluate implementations of these models based on the perceiver [27] and vision transformer [13] architectures in simulation.

Our envisioned system consists of three parts: a domain grounding model, that initialises a task planner, a task planner that produces a high-level plan, a policy network turns this plan into a low-level motion policy. We present an overview of the concrete problems we want to study in Sections 4.1, 4.2, and 4.3 and propose a solution architecture in Section 4.4. The solution consists of several components:

- *Model Architecture*: A concept for a network architecture for our models is given in Section 4.4.
- *Model Implementations*: Based on our concept, two concrete architectures, PerTamp and *OOTAMP*, are introduced in Sections 4.6 and 4.5. PerTamp is strongly inspired by the Perceiver-Actor architecture [57], introduced in Section 3.8. OOTAMP uses a pre-trained encoder from SORNet [65] (introduced in Section 3.7), which in turn is based on a vision transformer architecture.
- *Symbolic Actions and Predicates*: We need to define a set of actions and predicates, compatible with task planners, that we want to learn to execute and ground. The symbols we will use in this thesis are introduced in Section 4.7.

4.1 Problem

One of the critical limitations of existing approaches in imitation learning for robot manipulation is their focus on learning relatively short-term skills. Examples of skills learned include pick-and-place [64, 56, 57], filling cupboards [57], or flipping meat in a pan [16]. While reliably executing such skills is relevant for domains like household robotics, they are – by themselves – insufficient to solve longer horizon tasks like preparing a full meal or cleaning a kitchen.

Meanwhile, approaches from integrated task and motion planning (TAMP) were shown to be able to solve longer horizon tasks in robust ways successfully [33, 19, 9]. However, to achieve this, TAMP solvers rely on extensive models of their environment to perform the motion planning operations and compute properties like motion constraints or object grasping poses [18]. This makes pure TAMP solvers hard to deploy in dynamic, real-world environments.

Based on these observations, one possible answer to this problem is to use task planning to solve longer horizon problems and imitation learning instead of motion planning to execute the actions in the plan.

Problem Formulation

We are formulating the problem of integrating task planning with imitation learning in two parts:

1. Given atoms F, operators O, goal G and an observation of the environment ϕ, determine the initial state I to formulate the planning problem $P = \langle F, I, O, G \rangle$.

2. Given a high-level action $a \in A(s)$ from a task plan and an environment in state s, produce a low-level trajectory that, when executed on the robot, results in state $s' = f(a, s)$.

Viewed independently, these questions are not novel and have been the subject of many prior works. Recall SORNet [65] (discussed in Section 3.7) as an example for domain grounding, or Perceiver-Actor [57] (discussed in Section 3.8) as an example of action-conditioned imitation learning. However, little work has been done to solve these problems in a well-integrated fashion, i.e., performing domain grounding and skill execution using the same representations and learning to execute a PDDL action from a plan directly.

Research Questions

This thesis attempts to provide insights into how such an integrated approach might look like, leveraging recent results in transformer-based architectures for robotics. At the end of this thesis, we want to be able to (at least partially) answer the following research questions:

- **Q1**: Can we bridge the gap between symbolic planning domains on the one hand and vector-based imitation learning and domain grounding on the other?
- **Q2**: Can the same transformer-based encoder be used for imitation learning and domain grounding?
- **Q3**: Is it possible to solve longer horizon tasks by integrating task planning with imitation learning via domain grounding?
- **Q4**: How can we automatically collect training data for imitation learning and domain grounding from a simulation environment?

Before we develop a solution for our problem, we will explore the sub-problems identified in the previous section in more detail and quantify them in terms of a loss function.

4.2 Domain Grounding Problem

Our first identified sub-problem encompasses the problem of determining the truth value of ground predicate $f \in F$ in the initial state I of a planning problem P, given an observation $\omega(I)$. That is, we want to find a function

$$t(\theta, \omega(I), f) \mapsto \begin{cases} 1 & \text{if } f \in I \\ 0 & \text{otherwise} \end{cases}$$

where θ is a set of learnable parameters.

By applying this function to each ground predicate, which is a finite set, we can determine an approximation

$$\hat{I} = \{f \,|\, f \in F \wedge t(\theta, \omega(I), f) = 1\}$$

of the initial state I.

Loss for Domain Grounding

The function t can be modeled as a binary classifier. Assuming that our function is modeled by a neural network with an output dimension of two and a softmax, we can formulate an appropriate loss in the form of a cross-entropy loss or negative log-likelihood of the resulting softmax distribution [21]:

$$\mathrm{CE}(\mathrm{gt}(f, s), t(\theta, \omega(s), f)) = -\mathbb{E}_{\mathrm{gt(f,s)}}[\log t(\theta, \omega(s), f)],$$

where $\omega(s)$ is an observation in the state s, f is our atom of concern, $\mathrm{gt}(f, s)$ is a ground-truth classifier for f in state s.

4.3 Imitation Learning Problem

Imitation learning for manipulation can be framed in a large variety of ways. We will use the discrete imitation learning approach from Perceiver-Actor [57] .

A trajectory τ in a dataset $\mathcal{D}$ is a sequence of T features (observations) $\langle \phi_1, \ldots, \phi_T \rangle$. Each trajectory contains a set of keypoints, i.e., a subset of points that are intermediary goals for the gripper to reach. All non-key point features are additional observations to aid with observation density. Each ϕ_i consists of an observation of the scene and a mapping to the next end-effector keypoint. ϕ_0 and ϕ_T are keypoints, and there might be an arbitrary number of

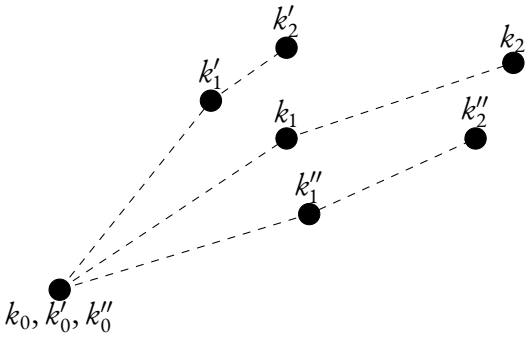


Figure 4.1: Example trajectories in the configuration space of a robot manipulator following our formulation of the imitation learning problem. From any keypoint k_i our policy π needs to predict the next keypoint k_{i+1}, represented by its pose (consisting of coordinates and rotation of the manipulator's end-effector and the open state of the gripper). Observation density is increased by collecting intermediate observation points on the trajectories and multiple trajectories of the same task.

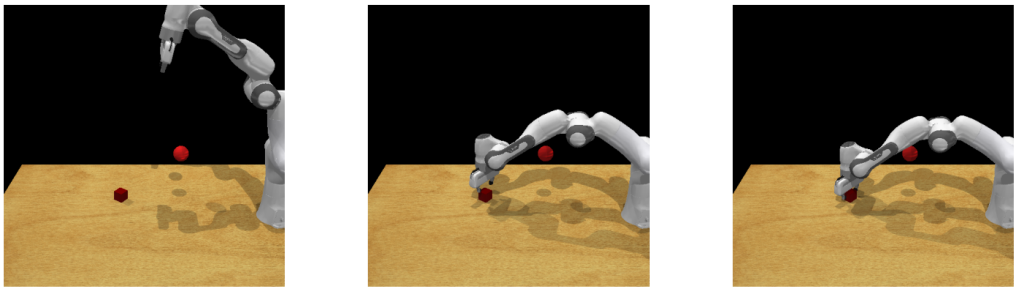


Figure 4.2: An example of the keypoints of a pick action on a red cube. The first keypoint (left picture) is the initial state. The second keypoint (middle picture) is the grip pose for the object. The third keypoint (right picture) changes the gripper state so that the object is held.

keypoints in between. For any ϕ_i, its mapped pose will be the keypoint ϕ_j, where $j > i$ and such there is no $\phi_{j'}$ with $j > j' > i$ that is also a keypoint. The context s_i associated with τ_i contains a symbolic action label and its translation into a one-hot vector. A visualisation of trajectories in the configuration space of the gripper is given in Figure 4.1. Example keypoints from a *pick* task are shown in Figure 4.2.

Each feature ϕ_i contains an observation $\omega(s_i)$ of the robot's environment at point i in the form of an RGB(-D) image and the end-effector pose at the next key point, k_j. An end-effector pose consists of a 3D rotation (x, y, z), a 3D orientation (R_x, R_y, R_z), and a binary gripper state g. The goal of a policy π is to map an observation $\omega(s_i)$ to its next key point in the dataset.

Loss for Imitation Learning

We can formulate a loss for this problem by combining multiple losses, one for each component in the end-effector pose. Given a ground truth $k_j = \langle x, y, z, R_x, R_y, R_z, g \rangle$ and a prediction $k'_j = \pi(\theta, \omega(s_i)) = \langle x', y', z', R'_x, R'_y, R'_z, g' \rangle$ we formulate the loss:

$$\begin{aligned} \text{ActionLoss}(k_j, k'_j) = &\frac{1}{3}((R_x - R'_x)^2 + (R_y - R_x y')^2 + (R_z - R'_z)^2) \\ &+ \frac{1}{3}(|x - x'| + |y - y'| + |z - z'|) \\ &+ \text{CE}(g, g'), \end{aligned}$$

i.e., the mean absolute error for the translational component, the mean squared error for the rotational component, and the binary cross-entropy loss $\text{CE}(g, g')$ for the gripper state. This loss is inspired by the loss used by Shridhar et al. [57] but transformed into a regression learning problem for translation and rotation.

4.4 Integrating Task Planning with Skill Execution through Imitation Learning and Domain Grounding

We will begin conceptualising our solution based on the problem formulation presented in Section 4.1. Like the problem, our solution will consist of two parts: a motion policy learned via imitation learning and a domain grounding model learned via supervised learning. Our idea is to use identical transformer-based encoder architectures (potentially with shared weights) for both downstream tasks. In the following sections, we will illustrate key assumptions that led to our architecture and devise a more detailed description of our model architectures from a high-level perspective. These will be the basis for devising our technical solution in the latter parts of this thesis.

Assumptions

In the following, we make some critical assumptions regarding the structure of our problem and its properties. These assumptions, which we derive from prior results in related approaches, will help guide the design of our model architecture and direction to the problem.

Assumption 1 (Applicability of Transformer-based Architectures) *Transformer architectures are appropriate for both robot imitation learning and domain-grounding learning problems.*

This assumption is backed by various previous approaches that used transformer-based architectures, both in imitation learning and domain grounding. SORNet (discussed in Section 3.7) [65] grounds spatial predicates using visual observations and a vision transformer architecture. Perceiver-Actor (discussed in Section 3.8) [57] learns to solve tasks from demonstrations, given visual observations and a perceiver encoder.

Assumption 2 (One-hot Vectors as Task and Symbol Encodings) *One-hot encodings provide sufficient conditioning to differentiate between tasks in a multi-task imitation learning problem.*

Several learning approaches in robotics rely on language prompts to guide models like motion policies to achieve a particular task [56, 57, 31]. In imitation learning, one might use a language model to encode the instruction (e.g., pick the green ball) into a vector embedding and provide it as additional input to the policy model [56]. However, Liu et al. [43] have shown that language embeddings provide no clear benefit over simple one-hot encodings, which are just direct mappings of a symbol to a vector.

Assumption 3 (Object Representation through Visual References) *Visual references of objects are sufficient representations to differentiate between objects for imitation learning and domain grounding.*

Using visual prompts as guidance has been successfully deployed for a variety of tasks in robot learning [65, 36, 35, 64, 46]. Pictures or other visual representations of objects serve as additional input to a model, e.g., to identify which object to grasp [36, 64], or for which object a predicate's truth value should be determined [65, 46]. Compared to text-based representations, this approach promises to better differentiate instances of the same type of object so long they have physical differences (e.g., a text-based system might struggle with picking the right blue cup among three blue cups with different shapes).

Assumption 4 (Structural Similarity) *Vision-based imitation learning and vision-based domain grounding are structurally similar problems that can be approached with a shared encoder architecture.*

Imitation learning policies [64, 56, 57] and vision-based predicate grounders [65, 46] rely on similar input (some form of processed image data) and use embeddings from an encoder as an intermediary. Approaches like SORNet [65] show that using one encoder for multiple tasks (predicate grounding and distance vector prediction) is feasible.

Architecture

Based on the assumption above, we can now develop a conceptual neural network architecture to solve the problems introduced in Section 4.1. This conceptual architecture will later be built into two concrete implementations for evaluation. Our concept is based on recent transformer-based architectures for robot learning, namely SORNet [65] and Perceiver-Actor [57].

Based on the sub-problems we identified in Section 4.1, we require two different models: a *domain grounding* model and a *motion policy* model. Both models rely on a low-level visual observation of the environment in the form of RGB(-D) images from a simulation environment. Following *Assumption 4*, we will build our architecture around a shared encoder network for both the domain grounding and the motion policy model. The purpose of the encoder network is to create an abstraction of the observation in the form of an embedding. This embedding is then passed to task-specific decoders, one for domain grounding and the other for our motion policy. Based on *Assumption 1*, we opt to use a transformer-based architecture for the encoder. This implies that our input observation will be a vector stream requiring some architecture-specific pre-processing.

Both of our sub-problems are object-centric, i.e., a model requires additional conditioning such that the *correct* objects are identifiable in the observation. We follow *Assumption 3* and represent objects in the form of visual references, i.e., for each object that we want to interact with, we provide one image of the object as conditioning. These reference images

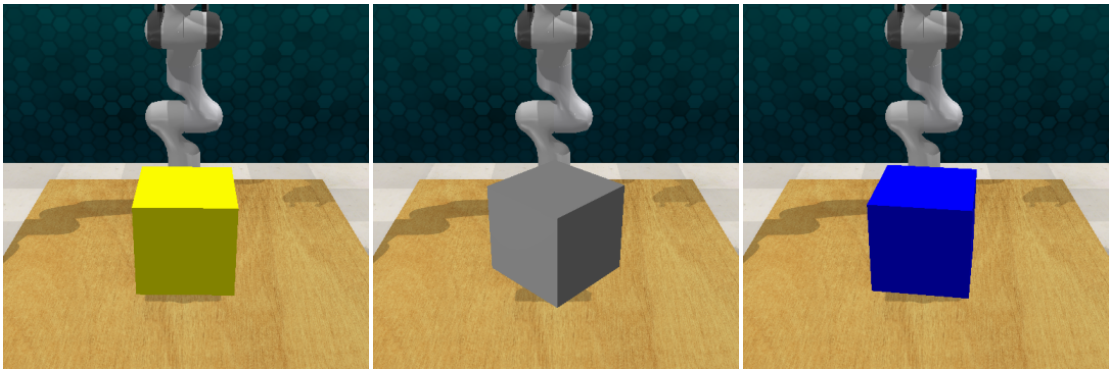


Figure 4.3: Examples of visual references of cuboid objects in a simulation environment used for conditioning of our models. Each object features a different rotation, which should aid the models to generalise to varying object views.

are provided as part of the input stream to the encoder network. This is identical to the concept of canonical views in SORNet [65] but also shares similarities with Perceiver-Actor [57], which uses a language embedding of the object instead of a visual reference. In case of domain grounding, we would provide the objects for which we want to determine a predicate's ground truth (e.g., for a ground predicate `on_table(red_book)`, we would provide a reference image of `red_book`). For the motion policy, we provide references to the objects that we want to interact with (e.g., for an action `pick(blue_glass)`, we would provide a reference image of `blue_glass`). Figure 4.3 shows a set of visual reference examples. This approach extends to predicates and actions of varying arities since transformer-based architectures are generally variable in terms of the input sequence length they permit.

For each task, we will train one task-specific decoder. Whereas SORNet [65] uses an ensemble of decoder networks, one for each predicate they ground, we want to use the same decoder for all predicates and actions. From *Assumption 2*, we can use one-hot vectors to represent different predicate and action symbols.

Based on these general architecture decisions, we will now introduce two concrete implementations, using vision transformers [13] and perceivers [28] respectively.

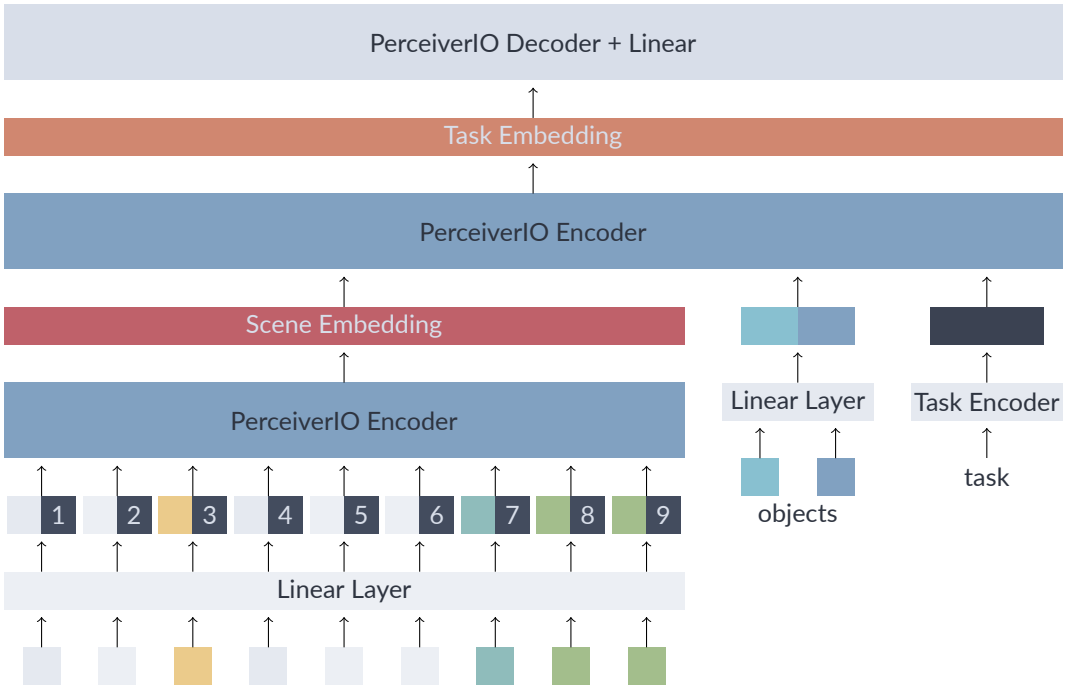


Figure 4.4: Architecture of the perceiver-based [27] PerTamp. In the encoder, an RGB-D observation of the environment is split into patches and fed through a perceiver layer to get a scene embedding. Visual references of objects are split into patches and together with the scene embedding and a task embedding (i.e., the embedding of a predicate or action symbol) fed through a second perceiver. The resulting embedding is then fed through a cross-attention-based decoder and task-specific linear layer to get the desired output (an end effector pose or a truth value).

4.5 PerTamp - A Perceiver Based Approach

PerTamp (Perceiver for TAMP) is our first architecture implementing our proposed concept using a perceiver [27]. Its architecture (depicted in Figure 4.4 makes use of unmodified perceiver encoders and decoders, similar to [57]. However, it does not rely on voxels and uses a different decoder architecture. Like Perceiver-Actor, we do not use pre-training. Instead, we train the perceiver from the ground using only our tasks.

Encoder Architecture

The PerTamp encoder consists of two perceiver encoders (one for scenes and one for the entire input sequence), a set of latents, trainable position embeddings, and a convolutional

patchifier.

An observation ω (an RGB-D image of the environment) is split into $16 \times 16 \times n$ patches through a convolutional layer with kernel size $16 \times 16 \times 4$ and stride 16 and flattened. Each token (corresponding to a flattened patch) is added to a unique position embedding, depending on the token's position in the input sequence. The input sequence is then passed through the first perceiver encoder to get a scene embedding vector.

A set of at most two visual reference pictures is patchified and flattened using the same approach. For actions or predicates with an arity smaller than 2, the vector of flattened patches is filled with 0 to keep the dimensionality. The visual reference tokens are concatenated with the embedding of our observation ω and a linearly-projected one-hot encoding of the action or predicate symbol. Once again, tokens are added with a unique trainable position embedding to build the input sequence for the second perceiver encoder. This results in the final embedding that is used by the task-specific decoder.

Decoder Architecture

The PerTamp decoder consists of a cross-attention-based perceiver decoder, a task-specific linear layer to process the outputs, a set of latents, and a convolutional patchifier.

The embedding from the encoder is processed by the perceiver's standard cross-attention-based decoder [27], which performs cross-attention between the input and a set of trainable latent vectors to extract relevant information from the embedding. Depending on the downstream task, the output is either a linear layer with dimension 2 and a softmax (for domain grounding) or a linear layer with dimension 7 (i.e., a target pose for the end effector for imitation learning).

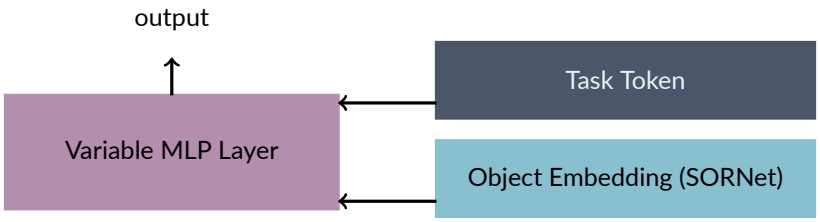


Figure 4.5: Architecture of the SORNet-based [65] OOTamp decoder. The encoder is an unmodified SORNet encoder which takes at most two object references. The embeddings corresponding to the visual references are concatenated with a linearly projected task conditioning (a one-hot vector corresponding to the predicate or action symbol) and fed through the decoder, which is a task-specific linear layer, to get the desired output (an end effector pose or a truth value).

4.6 OOTamp - A Vision Transformer Based Approach

OOTamp (Object-Oriented Transformer for TAMP) is the second implementation of our proposed concept. Its architecture (highlighted in Figure 4.5) is built around a vision transformer [13] with pre-trained weights from the predicate-classification task of SORNet [65]. Since transformers have few structural biases, using weights from a related task may help our model converge faster and potentially give better results than training a model from scratch. The vision transformer encoder is combined with an MLP-based decoder for the downstream tasks.

Encoder Architecture

SORNet provides pre-trained encoder weights for their vision-transformer-based approach to predicate grounding. As we are using pre-trained weights from an existing architecture, our input must be processed the same way as it is for the original model. Just like PerTamp, a set of trainable positional embeddings are added to transformer input tokens. In the pre-trained weights, 50 positional embeddings were available (49 for the scene, 1 for the visual references). The RGB observation of the environment ω is therefore first scaled to $224 \times 224 \times 3$ and then split into patches of size $32 \times 32 \times n$ using a convolutional layer, which

leads to 49 input tokens. Each of the tokens is then added to its corresponding positional embedding.

Visual references are also RGB and scaled to $32 \times 32 \times n$ before again being patchified, leading to one token per visual reference. Although SORNet supports an arbitrary number of visual references, we use the same system as PerTamp, supporting at most two visual references and filling the vector with zeroes for smaller arities. The final positional embedding is added to each of the visual reference tokens, and the result is concatenated with the scene tokens and finally fed through the vision transformer.

The two embedded tokens corresponding to our visual references are now available for further processing in downstream tasks. Since the vision transformer performs cross-attention, the embeddings of the visual references should also contain information about the scene.

Decoder Architecture

Our decoder architecture is similar to SORNet's, as we also rely on a simple multi-layered MLP. Similarly to PerTamp, we want to use the same decoder for both downstream tasks, only adapting the output dimensions and processing.

The decoder takes as input the embedded tokens from the encoder, corresponding to the visual references, and a linearly projected one-hot encoding of the action or predicate of interest. Like PerTamp, the output is either a linear layer with dimension 2 and a softmax (for domain grounding) or a linear layer with dimension 7 (i.e., a target pose for the end effector for imitation learning).

4.7 Spatial Relations and Compositional Planning Actions

A significant point that we have not discussed yet is which predicates and actions we are learning to ground and replicate. The choice of predicates and actions is a delicate one, as it restricts the system's capabilities in perception and interaction.

Inspired by the results by Occhipinti et al. [48] and Yuan et al. [65], we decided to rely on simple spatial predicates to express observations of the world state abstractly. The choice of the right actions, however, is difficult. Actions can be defined at different levels of complexity. For example, cleaning up an entire kitchen can be seen as one singular action. However, this would make it very hard to learn a reliable policy from demonstrations due to the length of the trajectories. Actions can also be as small as pick, or even parts of a pick skill. Since approaches like those by Yuan et al. [64], Shridhar et al. [56] use these somewhat granular definitions of actions successfully, we follow suit but maintain some more complex actions. Since most of the tasks we want to learn from can be split into many small pick-and-place operations, we can learn these actions in various contexts and with multiple objects.

Spatial Relation Predicates

For this thesis, we defined a set of 13 predicates, with most of them referring to spatial properties of objects. The list of predicates is given in Table 4.1.

Compositional Actions

We defined a set of 11 actions that we want to learn via imitation learning. This list was developed by analysing the tasks learned by Shridhar et al. [57], splitting them into smaller actions likely to re-occur in different tasks. Table 4.2 gives the list of actions.

Predicate	Arity	Description
open	0	Indicates whether the gripper is open
free	0	Indicates whether the gripper is holding an object
ground	1	Indicates whether an object is on the table
clear	1	Indicates whether an object's top surface is not obstructed
empty	1	Indicates whether an object is empty
holding	1	Indicates whether an object is being held
above	2	Indicates that one object is above another
on	2	Indicates that one object is on top of another
left	2	Indicates that one object is to the left of another
right	2	Indicates that one object is to the right of another
front	2	Indicates that one object is in front of another
behind	2	Indicates that one object is behind another
in	2	Indicates that one object is inside another

Table 4.1: Description of predicates we learn to ground in our approach.

4.8 Summary

Our proposed architecture allows us to create object-centric representations of a scene. Given these representations, we can use task-specific decoders to perform imitation learning and domain grounding. We now focus on evaluating our proposed architecture and its concrete implementations, OOTAMP and PerTamp, on both the domain grounding and imitiation learning tasks to determine their suitability for our problem.

Action	Arity	Description
pick	1	Grasping an object with the hand
pick-from	2	Picking an object from a specific location
place	1	Putting an object at a specified location
put-on	2	Putting one object on top of another
put-in	2	Putting one object inside another
grasp	1	Holding an object with a firm grip
screw-in	2	Rotating an object to attach it by screwing it in
slide-on	2	Placing one object onto another by sliding it
open-drawer	1	Opening a drawer
sweep	1	Sweeping an area to remove debris
pour	2	Pouring liquid from one container into another
push	2	Applying force to an object like a switch

Table 4.2: Description of actions that we learn from the demonstration.

5

Evaluation

In the following, we describe the evaluation of our models PerTamp and OOTamp for domain grounding and imitation learning. Our goal is to determine the performance of our models on both tasks in a robotics simulation, both quantitively and qualitatively, to understand their suitability for an integrated agent with a task planner. A focus of our evaluation will be a qualitative analysis of the sub-tasks that perform the worst to deduce shortcomings in our approach.

We will first introduce the simulation environment we used and the adaptions we had

to make to generate training data with symbolic annotations. Next, we will introduce our dataset, the tasks we used for training, and our computing environment. Then, we perform an ablation study on multiple hyperparameters for both models to identify the best-performing settings for further analysis. Finally, we will evaluate the models qualitatively and quantitatively on both tasks.

5.1 Data Generation

To learn actions from demonstration and to ground predicates based on observation, we need a dataset that provides a variety of tasks and a large variation of these tasks. Since creating such large datasets by hand is not feasible, we need to automate the data generation process. Like Perceiver-Actor [57], we opted to use the RLBench [30] environment, built around CoppeliaSim [55] due to its extensive set of prebuilt tasks. Another benefit of RLBench is its scriptability, which allows us granular control over the data generation.

RLBench

RLBench is a benchmarking and dataset generation environment for learning robot manipulation tasks, introduced by James et al. [30], providing 100 example tasks of varying complexity. It builds on the CoppeliaSim (formerly known as V-REP) [55] robotics simulator by extending it with tools for building scriptable and randomisable manipulation tasks and for automated data generation.

RLBench supports the creation of tabletop robot manipulation tasks by providing a predefined simulation environment. The base environment consists of a workspace, a robot manipulator (a simulated version of a Franka Emika Panda with 7 DOF and a simple claw gripper), and several cameras (front, left- and right overhead, and gripper-attached). Several

control modes for the manipulator are supported for inference (e.g., joint velocities or using a motion planner). Demonstrations support randomisation and are generated using a motion planner.

Tasks in RLBench

Tasks in RLBench consist of a set of objects to interact with, a set of programmer-defined waypoints, a success condition, a natural language description of the task, and a Python script for controlling randomisation [30]. Instead of pre-defining a complete trajectory or letting an expert manually solve the task to gather demonstrations for imitation learning, the waypoints are used to automate the process. The expert places the waypoints in the simulation environment (e.g., attached to an object) to guide the movement of the manipulator, which is controlled by a motion planner moving the arm through the sequence of waypoints. This allows for task randomisation, as the manipulator is controlled dynamically. We treat waypoints in RLBench tasks as equivalents to our prior definition of keypoints in our imitation learning formulation.

To illustrate this, consider a simple grasping task with an object placed randomly on a table. We place our first waypoint statically and define our initial position. The second waypoint is attached 5 cm above the object, and the final waypoint represents the grasping pose for the object (which is static relative to it). By varying the object's position, color, and shape, this simple task can be randomised in many ways to create a large set of demonstrations.

To the best of our knowledge, there is currently no scriptable environment for generating manipulation task demonstrations for imitation learning that supports symbolic annotations of actions and predicates. We developed a custom set of tools around RLBench to allow us to collect symbolic annotations for observations and demonstration trajectories suitable for task-conditioned imitation learning.

```
waypoint_map = {
    -1 : TampAction("grasp", ["pick_and_lift_target"]),
    0 : None,
    1 : TampAction("lift", ["pick_and_lift_target"]),
    2 : None,
    3 : None
}
```

Listing 5.1: An example of a symbolic annotation of waypoints for a task in RLBench. The first three waypoints are mapped to an action `grasp` on the object `pick_and_lift_target`. In waypoint 1, `grasp` is completed, and the new action `lift` begins, which continues for another 3 waypoints.

Symbolic Action Annotations in RLBench

For each task in RLBench's task library, we define a mapping between its waypoints and a fitting action label from the list we established in Section 4.7. An example of an action annotation with two actions (`grasp` and `lift`) on an object `pick_and_lift_target` is given in Listing 5.1. The object's name corresponds to its name in the simulation environment, automatically allowing us to extract a visual reference for the dataset. Observations in the dataset, including intermediary observations between waypoints, are grouped automatically based on their annotated action labels.

Symbolic States in RLBench

At each waypoint, we automatically compute the symbolic state of the observation. Each predicate introduced in Section 4.7 is grounded with all possible permutations of objects in the environment. Then, for each ground predicate, the truth value is determined by either a simulation environment property or a simple mathematical heuristic. For example, an object a is left of another object b if a overlaps with the projection of b on the y-axis and its rightmost point is left of the center point of b on the x-axis of the world frame. The ground truths of predicates like `holding` or `ground` can be directly extracted from object properties within

the simulation environment.

5.2 Training

Custom, symbolically annotated train and test datasets were generated from RLBench tasks to train our models. We describe the dataset and sampling procedures used to train our models in the following. Furthermore, we describe the computing hardware used to train our models and run our experiments.

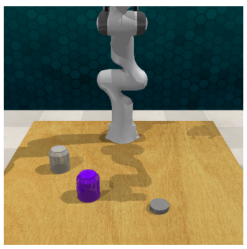
Cluster

For training, we used a cluster of nodes featuring four Nvidia A10 GPUs with 24GB of memory and 512GB of RAM. Each model instance was only trained using a single GPU; therefore, we could train up to 4 models in parallel on the same node. Experiments with trained models were performed on cluster nodes or a workstation with a commodity GPU (Nvidia RTX 3090 with 24GB of memory).

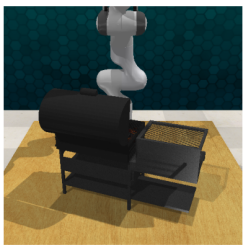
Dataset

Our training dataset consists of 10 tasks and 100 demonstrations per task. The test dataset consists of 10 demonstrations and the same set of tasks. On average, each demonstration consists of more than 80 observations and 3 labeled actions. Observations in RGB-D from three of the four preset camera views are saved with a 128×128 pixels resolution for each observation. The front view RGB-D observation is saved at 512×512 pixels. All views are free of artificial noise. Our training dataset consists of more than 80k observations across all tasks and demonstrations.

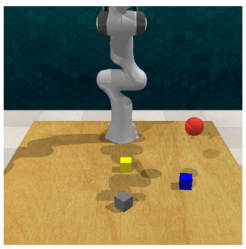
Both datasets are generated using our extensions to RLBench and feature different forms



(a) close_jar

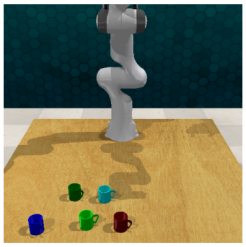


(b) meat_off_grill

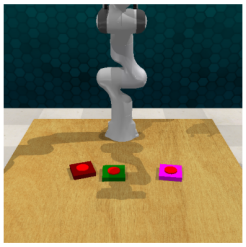


(c) pick_and_lift

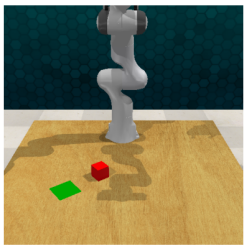
(d) turn_tap



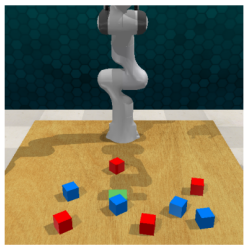
(e) pour_from_cup_to_cup



(f) push_buttons



(g) slide_block_to_target



(h) stack_blocks

(i) stack_wine

(j) sweep_to_dustpan

Figure 5.1: Front view camera pictures of the initial states of one demonstration for each task in our dataset.

of randomisations, depending on the task setup. All tasks feature randomised object positions, and those with simple objects also feature randomised colors. The tasks in the dataset are listed with descriptions in Table 5.1, and screenshots of each task are given in Figure 5.1. Tasks were chosen based on their compatibility with visual object references and based on the list of tasks used by Perceiver-Actor [57].

Task	Description
stack_blocks	Stacking block of the same color on top of each other
pick_and_lift	Grasping and raising an object out of a selection of three from a surface
slide_block_to_target	Moving a block to a specific location by sliding it
turn_tap	Grasping and rotating a tap
sweep_to_dustpan	Using a broom to gather debris and transfer it to a dustpan
meat_off_grill	Removing a random choice of meat (steak or chicken) from a grill
stack_wine	Stacking a wine bottle into a bottle holder
push_buttons	Pressing a button
close_jar	Picking a lid and putting it on a jar
pour_from_cup_to_cup	Transferring liquid from one cup to another

Table 5.1: Description of tasks in the final generated dataset.

Sampling

We use randomisation of the training dataset during the training of both models. For the imitation learning policy, we shuffle the entire set of observations before each epoch and then draw a sequence of the shuffled dataset.

Since most predicates describe geometric relations between objects, our dataset is biased toward negative truth values for ground predicates. We customise the sampling procedure for the domain grounding task to minimize the impact of this over-representation in our domain grounding task. First, we sample whether we want to draw a ground predicate that is true or

false from the dataset. Then, among all the predicates where we have an observation of that truth value (in our dataset, there is always one), we choose one randomly. Finally, we sample an observation from the dataset where this predicate occurs grounded and with the sampled truth value.

In both cases, we define the size of one epoch as equal to the total number of observations, even though that means we will likely oversample some observations in the domain grounding case.

5.3 Ablation Study

In the following sections we will analyse the effects of different hyperparameter settings on the our architectures PerTamp and OOTamp in both tasks. Based on the findings of this ablation study, we will select the models that perform the best, to evaluate our approach.

Hyperparameters

Both OOTamp and PerTamp have various hyperparameters resulting from their architecture and the training procedure. We train both architectures with the AdamW [44] and LAMB [63] optimisers, and training rates of 10^{-5} and 10^{-6} for each optimiser.

Since OOTamp makes use of a pre-trained encoder from SORNet [65], the epoch time is much lower than we needed to train the entire model. Because of that, we train the model between 50 and 150 epochs depending on the optimiser and learning rate. SORNet [65] provides several weights pre-trained on their datasets. We test our approach with two different sets of weights *leonardo_3view* (L3V) and *leonardo_gripper_3view* (LG3V). The decoder consists either of 2 or 3 linear layers.

Each PerTamp model consists of two perceiver [27] encoders. Each perceiver encoder

consists of latents of dimensionality 512, two attention blocks, with 6 self-attention layers per block, 8 self-attention heads, and ReLU activation. The perceiver block in the encoder uses one cross-attention head per attention block, with an attention dropout of 0.1, while the decoder perceiver blocks have no attention dropout and 4 cross attention heads. In addition to learning rates and optimisers, we experiment with different numbers of latent vectors. We train each variation for up to 50 epochs.

Domain Grounding

Tables 5.2 and 5.3 compare the performance of various instances of our OOTamp and PerTamp architectures, trained on the same dataset. For each model, we present the average loss over the entire test dataset in its best epoch. OOTamp generally performs better when compared to PerTamp, which is likely due to the pre-training of the SORNet encoder. Since we only train an MLP decoder for OOTamp and freeze the encoder weights, training is also significantly faster.

Model	Optimizer	Learning Rate	Sornet Model	Decoder Layers	Epochs	Time per Epoch	Loss
OO1	AdamW	10^{-5}	SORNet-L3V	2	50	875.6 s	0.014
OO2	AdamW	10^{-5}	SORNet-L3V	3	50	886.1 s	0.014
OO3	AdamW	10^{-5}	SORNet-LG3V	3	50	915.0 s	0.014
OO4	AdamW	10^{-6}	SORNet-L3V	2	150	832.3 s	0.035
OO5	AdamW	10^{-6}	SORNet-L3V	3	150	839.7 s	0.013
OO6	Lamb	10^{-5}	SORNet-L3V	2	100	789.1 s	0.072
OO7	Lamb	10^{-5}	SORNet-L3V	3	100	806.4 s	0.039
OO8	Lamb	10^{-5}	SORNet-LG3V	3	100	770.5 s	0.041
OO9	Lamb	10^{-6}	SORNet-L3V	2	150	776.1 s	0.340
OO10	Lamb	10^{-6}	SORNet-L3V	3	150	747.3 s	0.279

Table 5.2: Evaluation results of the OOTamp architecture on the domain grounding task with multiple hyperparameter variations.

OO5 is the best performing instance of OOTamp and will be used for our more detailed evaluation. However, *OO1-3* show comparable performance with less epochs which indicates that the learning rate (given enough epochs to compensate) has no significant impact

Model	Optimizer	Learning Rate	Encoder Latents	Decoder Latents	Epochs	Time per Epoch	Loss
PT1	AdamW	10^{-5}	256	512	50	3102.7 s	**0.059**
PT2	AdamW	10^{-5}	512	1024	50	5858.7 s	0.120
PT3	AdamW	10^{-6}	256	512	50	3089.3 s	0.492
PT4	AdamW	10^{-6}	512	1024	50	5876.2 s	0.516
PT5	Lamb	10^{-5}	256	512	50	3421.3 s	0.128
PT6	Lamb	10^{-5}	512	1024	50	6146.9 s	0.531
PT7	Lamb	10^{-6}	256	512	50	3445.3 s	0.584
PT8	Lamb	10^{-6}	512	1024	50	6189.2 s	0.615

Table 5.3: Evaluation results of the PerTamp architecture on the domain grounding task with multiple hyperparameter variations.

on performance. All models using the LAMB optimiser performed significantly worse than those trained with AdamW. We notice no relevant difference between the SORNet-L3V and SORNte-LG3V weights in terms of performance

Due to its significantly longer training time, we only trained PerTamp for 50 epochs (which is comparable to Perceiver-Actor [57]). Each of the PerTamp models performed significantly worse than the best OOTamp models. This indicates that pre-training on large datasets is a valuable for better model performance. Using a larger number of latents increased the training time almost by a factor of 2. The best PerTamp model was *PT1*, which we will use for our detailed evaluation.

Imitation Learning

We perform a similar analysis of our architectures on the imitation learning tasks, using the same variations in the hyperparameters. The results are presented in Tables 5.4 and 5.5.

Generally, we can observe a similar pattern, with OOTamp outperforming PerTamp. However, the difference in the loss measure is not as significant as in the domain grounding task. We can also observe that the same patterns emerge regarding the use of LAMB and AdamW optimisers, and the number of latents.

The best OOTamp instance is *OO2*, which performs almost identical to model 3. This

Model	Optimiser	LR	SORNet Model	Decoder Layers	Epochs	Time per Epoch	Loss	Position	Orientation	Gripper State
OO1	AdamW	10^{-5}	SORNet-L3V	2	50	927.6 s	0.031	0.010	0.004	0.016
OO2	AdamW	10^{-5}	SORNet-L3V	3	50	930.8 s	0.028	0.009	0.003	0.015
OO3	AdamW	10^{-5}	SORNet-LG3V	3	50	933.1 s	0.029	0.009	0.003	0.016
OO4	AdamW	10^{-6}	SORNet-L3V	2	150	854.3 s	0.055	0.013	0.012	0.030
OO5	AdamW	10^{-6}	SORNet-L3V	3	150	869.9 s	0.046	0.014	0.012	0.020
OO6	Lamb	10^{-5}	SORNet-L3V	2	100	893.7 s	0.090	0.020	0.025	0.045
OO7	Lamb	10^{-5}	SORNet-L3V	3	100	899.5 s	0.077	0.021	0.023	0.033
OO8	Lamb	10^{-5}	SORNet-LG3V	3	100	787.4 s	0.078	0.022	0.022	0.034
OO9	Lamb	10^{-6}	SORNet-L3V	2	150	771.4 s	0.263	0.037	0.090	0.136
OO10	Lamb	10^{-6}	SORNet-L3V	3	150	852.4 s	0.250	0.040	0.094	0.116

Table 5.4: Evaluation results of the OOTamp architecture on the imitation learning task with variation on optimiser, learning rate (LR), version of SORNet, and number of decoder layers.

Model	Optimiser	LR	Encoder Latents	Decoder Latents	Epochs	Time per Epoch	Loss	Position	Orientation	Gripper State
PT1	AdamW	10^{-5}	256	512	50	3052.9 s	**0.068**	0.015	0.005	0.048
PT2	AdamW	10^{-5}	512	1024	50	5834.2 s	0.090	0.022	0.010	0.057
PT3	AdamW	10^{-6}	256	512	50	3124.0 s	0.114	0.030	0.022	0.063
PT4	AdamW	10^{-6}	512	1024	150	6060.8 s	0.189	0.037	0.477	0.104
PT5	Lamb	10^{-5}	256	512	100	3432.5 s	0.122	0.031	0.027	0.063
PT6	Lamb	10^{-5}	512	1024	100	6264.9 s	0.223	0.040	0.059	0.124
PT7	Lamb	10^{-6}	256	512	100	3362.2 s	0.560	0.077	0.183	0.340
PT8	Lamb	10^{-6}	512	1024	150	6133.6 s	0.845	0.077	0.167	0.601

Table 5.5: Evaluation results of the PerTamp architecture on the imitation learning task with variation on optimiser, learning rate (LR), and number of encoder and decoder latents.

further confirms, that there is no significant difference in the pre-trained SORNet weights. PerTamp's best instance is *PT1*, which performs significantly better than all other variations of PerTamp.

We will use OOTamp OO2 and PerTamp PT1 for our detailed evaluation.

5.4 Results in Predicate Grounding

We need an accurate initial state to plan for a planning problem instance successfully. This requires our domain grounding model to have strong predictive capabilities.

Our results indicate that both architectures have similar predictive capabilities despite their different loss values during training. We will now evaluate the performance of both architectures in detail.

Quantitative Analysis

Over the entire dataset, both models have very similar performance. Figure 5.2 shows both models' accuracy values for each predicate we defined. Predicates like *clear*, *open*, *empty*, *holding*, *in*, *on*, and *free*, have accuracy values of > 0.8 over the entire dataset. However, the basic spatial predicates *left*, *front*, *right*, *behind* perform equally badly on both models. One possible explanation for these results is that those predicates require information from more than one image patch in the input sequence. In contrast, the perceptual information necessary for predicates with high accuracy can be contained in one patch. This might be a clue that cross-patch information is insufficiently acquired in the embeddings.

Table 5.6 gives accuracy, true positive, false positive, true negative, and false negative rates over the entire dataset for both models. Again, both models perform very comparably on all metrics. What is noticeable is that both have a very large number of false positives, leading

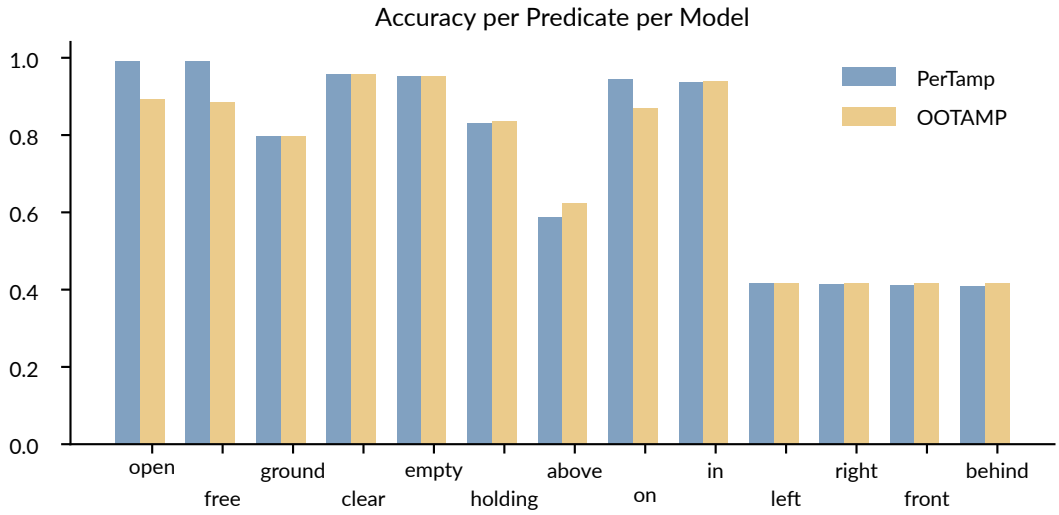


Figure 5.2: Accuracy per predicate for OOTamp and PerTamp. We can see that the accuracy is similar between both models, with PerTamp performing slightly better on *open* and *free*, whereas OOTamp performs better on *above*. The predicates *left*, *right*, *front*, *behind* have significantly lower accuracy values than the other predicates.

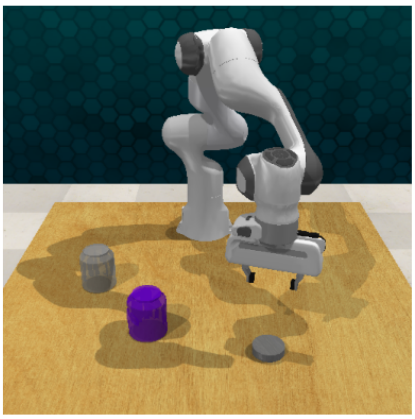
	PerTamp	OOTamp
True Positive	48429	47394
True Negative	43852	43620
False Positive	55307	56342
False Negative	16	248
Accuracy	0.625	0.617
Precision	0.442	0.436
F1-Score	0.613	0.607

Table 5.6: Evaluation of the performance of PerTamp and OOTamp on the domain grounding task.

to comparably low accuracy values. This might be a result from our oversampling procedure during training, which significantly biases the data towards positive predicate evaluations compared to the base distribution in the dataset. While not directly comparable, due to the difference in the kind of predicates and the dataset used, the original SORNet architecture [65] achieves a slightly better F1-score of 68.4 on their kitchen dataset task.

Qualitative Analysis

We will now look at examples of predicates to understand possible challenges in our observations that might explain the quantitative results. The predicate with the lowest accuracy was *behind* for PerTAMP and its dual *front* for OOTAMP. In Figure 5.3a, we see a false positive for the predicate *behind*. This specific scenario is an edge case based on our definition of behind, as the grey and purple jars do not align in the frame's x or the y coordinate. Thus, this spatial arrangement yields false for any of the spatial predicates.



(a) Example of a false positive for the ground predicate `behind(jar0, jar1)`. The purple *jar0* is in front of the grey *jar1*.

(b) Example of a false positive for the ground predicate `on(target, block)`. The green *target* is below the red *block*.

Figure 5.3: Examples of false positives for ground predicates.

The predicate with the lowest precision was *on* for both models since our models have high accuracy values for *on*, which is the result of a large number of false positives. *on* is challenging to determine visually, especially in obstructed views like in Figure 5.3b, since the only visual indication is generally some form of overlap between object shapes from the camera's perspective. We notice that in most cases where *on* is a false positive, it is partially overlapped by another object like the manipulator. Generally, in our data, the predicate *on* is usually only positive after a gripper interaction. This means that the models might have learned to predict true for *on* when the queried objects are near the manipulator, regardless of their position.

5.5 Results in Imitation Learning

We will now study the effectiveness of our best models *PT1* and *OO2* for solving imitation learning tasks in more detail. Our goal is to determine the accuracy and quality of the policies we learned and their applicability in the simulation environment. First, we will perform a quantitative analysis of both models' performance to compare them objectively. We will also do a set of experiments in the RLBench environment to determine issues with the learned policies.

Quantitative Analysis

We evaluated OOTamp and PerTamp on the test dataset to determine their respective performance regarding the loss value per action and task. We also evaluated both models manually in simulation to test their inference capabilities. Overall, OOTamp performs significantly better than PerTamp in almost every task and for nearly every action.

In Figure 5.4, we compare the average loss of both models for each action represented in our dataset. OOTamp produces significantly better predictions than PerTamp for all but

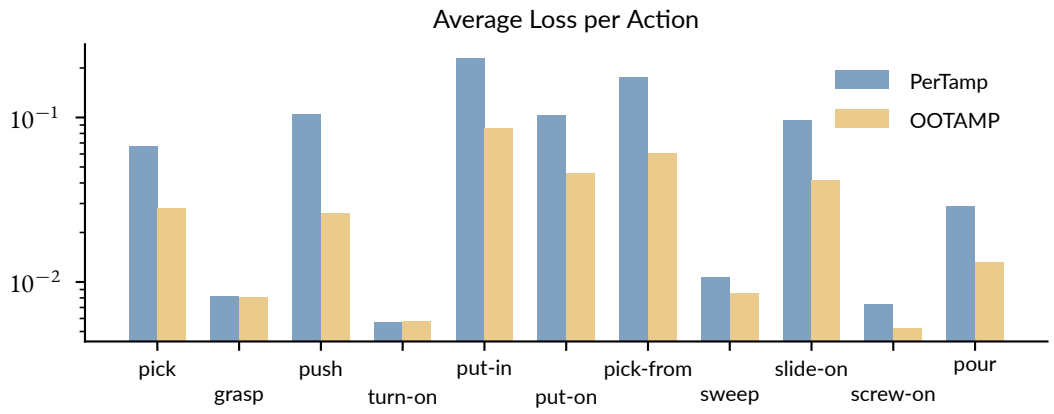


Figure 5.4: Average loss (on a logarithmic scale) of OOTamp and PerTamp per action in the test dataset. OOTamp outperforms PerTamp significantly on all but two actions.

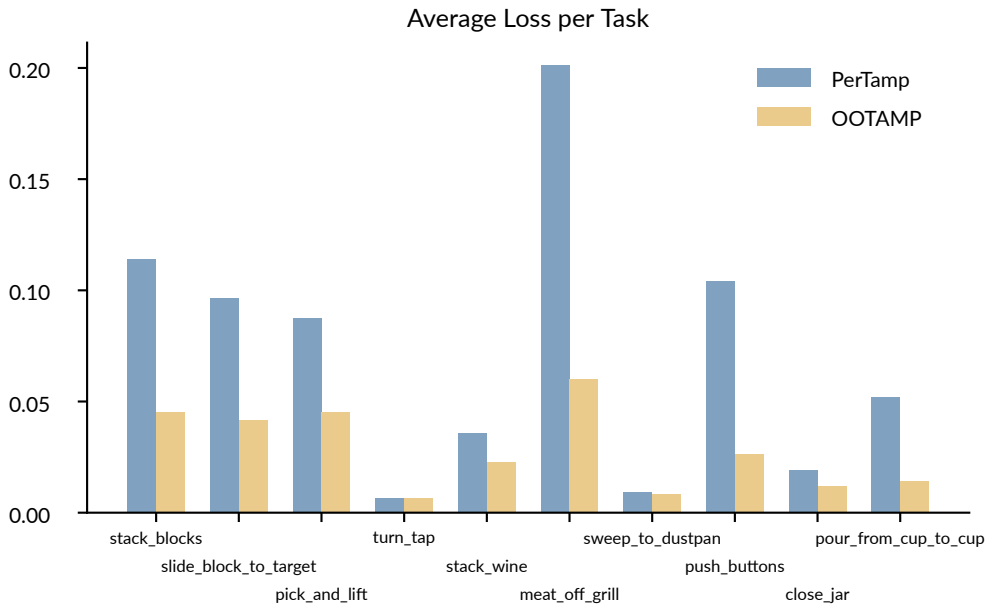


Figure 5.5: Average loss of OOTamp and PerTamp per task in the test dataset. OOTamp outperforms PerTamp on 6 of the 10 listed tasks.

Method	OOTamp pick	grasp	push	turn-on	put-in	put-on	pick-from	sweep	slide-on	screw-on	pour
Average Loss	0.028	0.008	0.026	0.006	0.086	0.049	0.061	0.008	0.042	0.005	0.031
Average Loss (KP)	0.077	0.010	0.053	0.006	0.073	0.148	0.176	0.007	0.160	0.006	0.021
Successful Inference	1	0	5	3	0	1	0	2	6	1	0
	PerTamp pick	grasp	push	turn-on	put-in	put-on	pick-from	sweep	slide-on	screw-on	pour
Average Loss	0.067	0.008	0.104	0.006	0.230	0.103	0.175	0.011	0.096	0.007	0.029
Average Loss (KP)	0.247	0.010	0.183	0.007	0.183	0.430	0.437	0.009	0.042	0.005	0.029
Successful Inference	0	0	0	1	0	0	0	0	1	0	0

Table 5.7: Results of OOTamp and PerTamp on different evaluations, evaluated per action. Average loss and average loss on keypoints (KP) are determined on the test dataset. Successful inference refers to the number of successful executions of an action in the simulation environment out of 10. For this measure, actions were tested only in tasks where the action was also used in the training set.

two actions and is on par with PerTamp for the remaining two. Since tasks are composed of multiple actions, we can see a similar effect when comparing the average loss over all actions in a task as shown in Figure 5.5. Again, OOTamp significantly outperforms PerTamp on 6 of the 10 tasks evaluated.

Our dataset mainly contains intermediate points between keypoints to increase the density of the distribution of samples. However, since our approach is designed as a keypoint-to-keypoint prediction approach, we are especially interested in the quality of the policy at keypoints. Table 5.7 compares the average loss at keypoints for each action. While the loss is not consistently higher at keypoints, it seems generally worse for both models. The loss is significantly higher for some actions like *pick* or *put-on*. This indicates that both learned policies are unstable at keypoints.

A possible explanation of this behaviour is that the space around keypoints consists of intermediary points which map to the keypoint they surround. Slight inaccuracy in the prediction, maps the current position not to the target keypoint, but to one close to the known intermediaries. This means that the next prediction is likely going to be the same keypoint again.

Qualitative Analysis

Table 5.7 also compares the performance of both models on an inference task. In this task, we run the model in inference mode in RLBench and count the successful attempts at executing an action given our policy out of 10 attempts. Each action was evaluated in one of the available task environments, where it also occurs in our sample solutions in the dataset:

- *pick* was evaluated in *pick_and_lift*
- *grasp* was evaluated in *turn_tap*
- *push* was evaluated in *push_buttons*
- *turn-on* was evaluated in *turn_tap*
- *put-in* was evaluated in *stack_wine*
- *put-on* was evaluated in *meat_off_grill*
- *pick-from* was evaluated in *meat_off_grill*
- *sweep* was evaluated in *sweep_to_dustpan*
- *slide-on* was evaluated in *slide_block_to_target*
- *screw-on* was evaluated in *close_jar*
- *pour* was evaluated in *pour_from_cup_to_cup*

The evaluation was performed under manual supervision from pre-defined states, i.e., we manually called the policy model in the simulation until no progress was made. The success of an action was evaluated manually based on visual feedback. OOTamp successfully executed 7 out of 11 actions at least once. However, it only manages to execute two actions *push* and *slide-on* at least 50% of the time. PerTamp only manages to successfully execute two actions (*turn-on* and *slide-on*). For both, we only recorded one successful attempt. No attempt was made to chain multiple actions due to the low success rate of the tasks.

Figure 5.6: An example of a failed attempt to execute the action *grasp* with OOTamp. Given the initial state, the model successfully predicts a point close to the target object. All further attempts result in another point close to the object and no progress towards completing the task.

The low performance of both models on the successful iterations task indicates that even though the policies have converged to relatively low levels of loss, their accuracy is still insufficient for the task. This holds especially for keypoints where we can observe instability in the policies outputs, leading to the increased loss on the dataset. In simulation, we can observe two common phenomena among unsuccessful attempts: the model gets stuck predicting positions close to the object to interact with, or the model creates a phantom trajectory without interacting with the object.

Figure 5.6 gives an example of the policy being stuck in the action *grasp*. In the first step, the manipulator successfully moves from its initial position toward the object of interest. However, from the second point onwards, no matter how often the policy network is called to give the next keypoint, it only predicts a point within proximity of the current point, thus failing to complete the action. Similar behaviour can be observed across almost all actions in our dataset.

Another common behaviour leading to unsuccessful attempts to execute an action is what we refer to as a phantom execution. In Figure 5.7, we give an example of this behavior for the action *pick*. First, the manipulator successfully moves towards the object. In the training

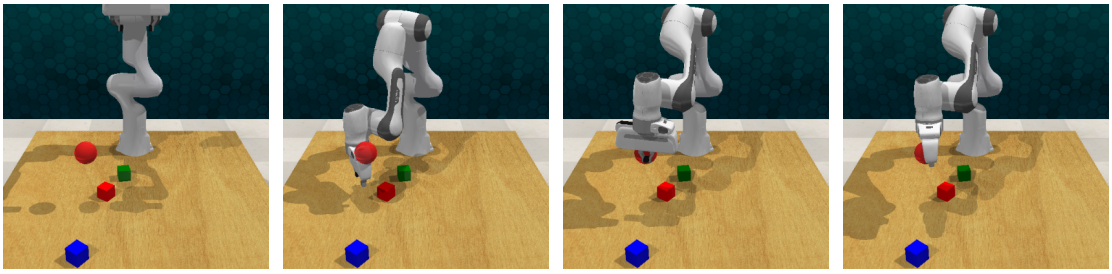


Figure 5.7: An example of a failed attempt to execute the action *pick* with the OOTamp model. In this example, we can see a phantom execution of the action. The model fails to predict a grasping pose after moving toward the object from the resting position. Instead, the arm is lifted towards the red ball, which signals a lifting target without picking the object. This movement towards the red ball is the next step after grasping the cube, as it occurs in the training data for this task.

data, the next keypoint would be a grasping pose, followed by a pose where the object is held and lifted off the ground. However, the grasping sub-action is completely skipped, and the arm moves directly up instead.

We assume that both phenomena are related to the higher loss around keypoints. However, another explanation might be insufficient resolution or insufficient variation in the data during training. While our training dataset is similar in size to the one used by Perceiver-Actor [57], we do not make use of any perturbations during training, which decreased the density of our observation distribution in comparison. Another observation is that Perceiver-Actor, a state-of-the-art approach, has task success rates of less than 50% for many of its tasks, which indicates the difficulty of the purely observation-based imitation learning problem.

6

Conclusion

Integrating task planning with imitation learning holds promise as an alternative to well-established techniques like integrated TAMP for solving long-horizon tasks in unstructured environments. However, the integration presents two key challenges: domain grounding and action-conditioned imitation learning. Both challenges are closely related, as they revolve around object-centric scene understanding.

In this thesis, we present a transformer-based encoder-decoder architecture for learning domain grounding and action-conditioned policies in an integrated way. This multi-modal

architecture comprises a scene encoder and task-specific decoders, utilizing visual references to objects and one-hot embeddings for conditioning the task.

We adapted an existing simulation environment to automatically generate symbolically annotated manipulation demonstrations in ten different tasks featuring variations of 12 actions. Additionally, we automatically collect symbolic descriptions of the world state using 13 different (primarily spatial) predicates for learning domain grounding.

We evaluated two concrete implementations of our architecture — OOTamp and PerTamp — on our tasks with 100 demonstrations per task. OOTamp builds on a vision-transformer encoder architecture and uses weights from SORNet [65], pre-trained on a large dataset. PerTamp is inspired by Perceiver-Actor's [57] usage of the perceiver architecture and is trained from scratch.

Our experimental results indicate that both architectures perform almost identically on the domain grounding tasks. While their performance is comparable to existing architectures like SORNet, it suffers from many false positives, likely due to oversampling of true positives during the training procedure. On the imitation learning task, OOTamp significantly outperforms PerTamp regarding loss on the training set both on single actions and tasks. However, transferred to an interactive simulation environment, both exhibit low reliability, with OOTamp only achieving success rates of $\geq 50\%$ on two actions with ten trials per action. PerTamp only manages to execute two actions successfully once, which indicates that pre-training has a large influence on the model's performance in imitation learning. While these results imply that the models are not yet ready for integration with a task planner, they show the general feasibility of using one shared architecture among both tasks.

6.1 Future Work

So far, we have yet to achieve an integration of task planning with imitation learning, given the performance of our models, especially on the imitation learning task. Various improvements can be made regarding our architecture, data, and procedure. Our architecture can be improved by providing additional structural bias in the input (e.g., by carefully choosing position embeddings) and adding support for input from multiple camera angles.

The dataset can be improved by increasing the variety of tasks, such that generalisation is more enforced. Actions could be split into smaller sub-actions, meaning they would have fewer key points. The over-representation of true negatives should also be mitigated by changing the distribution in the dataset instead of oversampling. For this purpose, the dataset might be split into two parts (one for imitation learning and one for domain grounding) for more granular control over these properties.

Our imitation learning task could be improved in various ways. First, performing pre-training on large datasets that are more closely aligned with our task seems beneficial. This could be combined with a form of curriculum learning, where we change the training task from something simpler (e.g., recognising an object) to our actual training task over multiple steps. The same can be applied to our imitation learning task, focusing first on simple actions and then switching to more complicated ones. These changes promise to supply more structural bias to the transformers, which is helpful for learning objectives. Finally, we can incorporate ideas from interactive imitation learning to manually correct the adverse effects of our keypoint-based learning approach.

References

[1] Agostini, A. & Piater, J. (2023). Unified Task and Motion Planning using Object-centric Abstractions of Motion Constraints.

[2] Asai, M., Kajino, H., Fukunaga, A., & Muise, C. (2022). Classical Planning in Deep Latent Space. *jair*, 74, 1599–1686.

[3] Bonet, B. & Geffner, H. (2020). Learning First-Order Symbolic Representations for Planning from the Structure of the State Space. In *Proceedings of the 24th European Conference on Artificial Intelligence* Santiago de Compostela.

[4] Cambon, S., Gravot, F., & Alami, R. (2004). A robot task planner that merges symbolic and geometric reasoning. In *Proceedings of the 16th European Conference on Artificial Intelligence*, ECAI'04 (pp. 895–899). NLD: IOS Press.

[5] Choi, J. & Amir, E. (2009). Combining planning and motion planning. In *2009 IEEE International Conference on Robotics and Automation* (pp. 238–244). Kobe: IEEE.

[6] Dalal, M., Mandlekar, A., Garrett, C., Handa, A., Salakhutdinov, R., & Fox, D. (2023). Imitating Task and Motion Planning with Visuomotor Transformers.

[7] Dantam, N. T., Kingston, Z. K., Chaudhuri, S., & Kavraki, L. E. (2018). An incremental constraint-based framework for task and motion planning. *The International Journal of Robotics Research*, 37(10), 1134–1151.

[8] Dasari, S. & Gupta, A. (2021). Transformers for one-shot visual imitation. In J. Kober, F. Ramos, & C. Tomlin (Eds.), *Proceedings of the 2020 Conference on Robot Learning*, volume 155 of *Proceedings of Machine Learning Research* (pp. 2071–2084).: PMLR.

[9] de Silva, L., Pandey, A. K., & Alami, R. (2013). An interface for interleaved symbolic-geometric planning and backtracking. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 232–239). Tokyo: IEEE.

[10] Dearden, R. & Burbridge, C. (2014). Manipulation planning using learned symbolic state abstractions. *Robotics and Autonomous Systems*, 62(3), 355–365.

[11] Deng, J., Dong, W., Socher, R., Li, L.-J., Kai Li, & Li Fei-Fei (2009). ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition* (pp. 248–255). Miami, FL: IEEE.

[12] Dornhege, C., Eyerich, P., Keller, T., Trüg, S., Brenner, M., & Nebel, B. (2009). Semantic attachments for domain-independent planning systems. *Proceedings of the International Conference on Automated Planning and Scheduling*, 19(1), 114–121.

[13] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N. (2021). An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *International Conference on Learning Representations*.

[14] Driess, D., Ha, J.-S., & Toussaint, M. (2020). Deep Visual Reasoning: Learning to Predict Action Sequences for Task and Motion Planning from an Initial Scene Image. In *Robotics: Science and Systems XVI*: Robotics: Science and Systems Foundation.

[15] Duan, Y., Andrychowicz, M., Stadie, B., Jonathan Ho, O., Schneider, J., Sutskever, I., Abbeel, P., & Zaremba, W. (2017). One-shot imitation learning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems*, volume 30: Curran Associates, Inc.

[16] Fu, Z., Zhao, T. Z., & Finn, C. (2024). Mobile ALOHA: Learning Bimanual Mobile Manipulation with Low-Cost Whole-Body Teleoperation.

[17] Garnelo, M., Arulkumaran, K., & Shanahan, M. (2016). Towards Deep Symbolic Reinforcement Learning.

[18] Garrett, C. R., Chitnis, R., Holladay, R., Kim, B., Silver, T., Kaelbling, L. P., & Lozano-Pérez, T. (2021). Integrated task and motion planning. *Annual Review of Control, Robotics, and Autonomous Systems*, 4(1), 265–293.

[19] Garrett, C. R., Lozano-Pérez, T., & Kaelbling, L. P. (2020). PDDLStream: Integrating Symbolic Planners and Blackbox Samplers via Optimistic Adaptive Planning. *ICAPS*, 30, 440–448.

[20] Geffner, H. & Bonet, B. (2013). *A Concise Introduction to Models and Methods for Automated Planning*. Number 22 in Synthesis Lectures on Artificial Intelligence and Machine Learning. San Rafael: Morgan & Claypool Publishers.

[21] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. Adaptive Computation and Machine Learning. Cambridge, Massachusetts: The MIT Press.

[22] Harnad, S. (1990). The symbol grounding problem. *Physica D: Nonlinear Phenomena*, 42(1), 335–346.

[23] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 770–778). Las Vegas, NV, USA: IEEE.

[24] Ho, J. & Ermon, S. (2016). Generative adversarial imitation learning. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems*, volume 29: Curran Associates, Inc.

[25] Huang, D.-A., Xu, D., Zhu, Y., Garg, A., Savarese, S., Fei-Fei, L., & Niebles, J. C. (2019). Continuous Relaxation of Symbolic Planner for One-Shot Imitation Learning. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (pp. 2635–2642). Macau, China: IEEE.

[26] Huang, X., Batra, D., Rai, A., & Szot, A. (2023). Skill Transformer: A Monolithic Policy for Mobile Manipulation. In *2023 IEEE/CVF International Conference on Computer Vision (ICCV)* (pp. 10818–10828). Paris, France: IEEE.

[27] Jaegle, A., Borgeaud, S., Alayrac, J.-B., Doersch, C., Ionescu, C., Ding, D., Koppula, S., Zoran, D., Brock, A., Shelhamer, E., Hénaff, O., Botvinick, M. M., Zisserman, A., Vinyals, O., & Carreira, J. (2022). Perceiver IO: A General Architecture for Structured Inputs & Outputs. In *International Conference on Learning Representations*.

[28] Jaegle, A., Gimeno, F., Brock, A., Vinyals, O., Zisserman, A., & Carreira, J. (2021). Perceiver: General perception with iterative attention. In M. Meila & T. Zhang (Eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research* (pp. 4651–4664).: PMLR.

[29] James, S., Bloesch, M., & Davison, A. J. (2018). Task-embedded control networks for few-shot imitation learning. In A. Billard, A. Dragan, J. Peters, & J. Morimoto (Eds.), *Proceedings of the 2nd Conference on Robot Learning*, volume 87 of *Proceedings of Machine Learning Research* (pp. 783–795).: PMLR.

[30] James, S., Ma, Z., Arrojo, D. R., & Davison, A. J. (2019). RLBench: The Robot Learning Benchmark & Learning Environment.

[31] Jang, E., Irpan, A., Khansari, M., Kappler, D., Ebert, F., Lynch, C., Levine, S., & Finn, C. (2022). BC-Z: Zero-shot task generalization with robotic imitation learning. In A. Faust, D. Hsu, & G. Neumann (Eds.), *Proceedings of the 5th Conference on Robot Learning*, volume 164 of *Proceedings of Machine Learning Research* (pp. 991–1002).: PMLR.

[32] Jiang, Y., Yang, F., Zhang, S., & Stone, P. (2019). Task-Motion Planning with Reinforcement Learning for Adaptable Mobile Service Robots. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (pp. 7529–7534). Macau, China: IEEE.

[33] Kaelbling, L. P. & Lozano-Perez, T. (2011). Hierarchical task and motion planning in the now. In *2011 IEEE International Conference on Robotics and Automation* (pp. 1470–1477). Shanghai, China: IEEE.

[34] Kase, K., Paxton, C., Mazhar, H., Ogata, T., & Fox, D. (2020). Transferable Task Execution from Pixels through Deep Planning Domain Learning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 10459–10465). Paris, France: IEEE.

[35] Kim, H., Ohmura, Y., & Kuniyoshi, Y. (2020). Using Human Gaze to Improve Robustness Against Irrelevant Objects in Robot Manipulation Tasks. *IEEE Robot. Autom. Lett.*, 5(3), 4415–4422.

[36] Kim, H., Ohmura, Y., & Kuniyoshi, Y. (2021). Transformer-based deep imitation learning for dual-arm robot manipulation. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (pp. 8965–8972). Prague, Czech Republic: IEEE.

[37] Konidaris, G., Kaelbling, L. P., & Lozano-Perez, T. (2018). From Skills to Symbols: Learning Symbolic Representations for Abstract High-Level Planning. *jair*, 61, 215–289.

[38] Kroemer, O., Niekum, S., & Konidaris, G. (2021). A review of robot learning for manipulation: Challenges, representations, and algorithms. *Journal of Machine Learning Research*, 22(30), 1–82.

[39] Lagriffoul, F., Dantam, N. T., Garrett, C., Akbari, A., Srivastava, S., & Kavraki, L. E. (2018). Platform-Independent Benchmarks for Task and Motion Planning. *IEEE Robot. Autom. Lett.*, 3(4), 3765–3772.

[40] Lamanna, L., Gerevini, A. E., Saetti, A., Serafini, L., & Traverso, P. (2021). On-line Learning of Planning Domains from Sensor Data in PAL: Scaling up to Large State Spaces. *AAAI*, 35(13), 11862–11869.

[41] Lamanna, L., Serafini, L., Saetti, A., Gerevini, A., & Traverso, P. (2022). Online Grounding of Symbolic Planning Domains in Unknown Environments. In *Proceedings of the Nineteenth International Conference on Principles of Knowledge Representation and Reasoning* (pp. 511–521). Haifa, Israel: International Joint Conferences on Artificial Intelligence Organization.

[42] LaValle, S. M. (1998). *Rapdily-Exploring Random Trees: A New Tool for Path Planning*. Technical report, Computer Science Department, Iowa State University.

[43] Liu, B., Zhu, Y., Gao, C., Feng, Y., Liu, Q., Zhu, Y., & Stone, P. (2023). LIBERO: Benchmarking Knowledge Transfer for Lifelong Robot Learning.

[44] Loshchilov, I. & Hutter, F. (2019). Decoupled Weight Decay Regularization.

[45] McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., & Wilkins, D. (1998). PDDL - The Planning Domain Definition Language.

[46] Migimatsu, T. & Bohg, J. (2022). Grounding Predicates through Actions. In *2022 International Conference on Robotics and Automation (ICRA)* (pp. 3498–3504). Philadelphia, PA, USA: IEEE.

[47] Newaz, A. A. R. & Alam, T. (2021). Hierarchical Task and Motion Planning through Deep Reinforcement Learning. In *2021 Fifth IEEE International Conference on Robotic Computing (IRC)* (pp. 100–105). Taichung, Taiwan: IEEE.

[48] Occhipinti, A., Bonet, B., & Geffner, H. (2022). Learning First-Order Symbolic Planning Representations That Are Grounded. In *PRL Workshop at International Conference on Planning and Scheduling*.

[49] Osa, T., Pajarinen, J., Neumann, G., Bagnell, J. A., Abbeel, P., & Peters, J. (2018). An Algorithmic Perspective on Imitation Learning. *FNT in Robotics*, 7(1-2), 1–179.

[50] Pastor, P., Hoffmann, H., Asfour, T., & Schaal, S. (2009). Learning and generalization of motor skills by learning from demonstration. In *2009 IEEE International Conference on Robotics and Automation* (pp. 763–768).

[51] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., & Chintala, S. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library.

[52] Plaku, E. & Hager, G. D. (2010). Sampling-Based Motion and Symbolic Action Planning with geometric and differential constraints. In *2010 IEEE International Conference on Robotics and Automation* (pp. 5002–5008). Anchorage, AK: IEEE.

[53] Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., & Sutskever, I. (2021). Learning Transferable Visual Models From Natural Language Supervision.

[54] Ratliff, N., Bagnell, J. A., & Srinivasa, S. S. (2007). Imitation learning for locomotion and manipulation. In *2007 7th IEEE-RAS International Conference on Humanoid Robots* (pp. 392–397).

[55] Rohmer, E., Singh, S. P. N., & Freese, M. (2013). V-REP: A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 1321–1326). Tokyo: IEEE.

[56] Shridhar, M., Manuelli, L., & Fox, D. (2021). CLIPort: What and Where Pathways for Robotic Manipulation.

[57] Shridhar, M., Manuelli, L., & Fox, D. (2022). Perceiver-actor: A multi-task transformer for robotic manipulation. In *6th Annual Conference on Robot Learning*.

[58] Silver, T., Athalye, A., Tenenbaum, J. B., Lozano-Pérez, T., & Kaelbling, L. P. (2022). Learning neuro-symbolic skills for bilevel planning. In *6th Annual Conference on Robot Learning*.

[59] Silver, T., Chitnis, R., Tenenbaum, J., Kaelbling, L. P., & Lozano-Perez, T. (2021). Learning Symbolic Operators for Task and Motion Planning. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (pp. 3182–3189). Prague, Czech Republic: IEEE.

[60] Srivastava, S., Fang, E., Riano, L., Chitnis, R., Russell, S., & Abbeel, P. (2014). Combined task and motion planning through an extensible planner-independent interface layer. In *2014 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 639–646). Hong Kong, China: IEEE.

[61] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems*, volume 30: Curran Associates, Inc.

[62] Watahiki, H. & Tsuruoka, Y. (2022). One-shot imitation with skill chaining using a goal-conditioned policy in long-horizon control. In *ICLR 2022 Workshop on Generalizable Policy Learning in Physical World*.

[63] You, Y., Li, J., Reddi, S., Hseu, J., Kumar, S., Bhojanapalli, S., Song, X., Demmel, J., Keutzer, K., & Hsieh, C.-J. (2020). Large Batch Optimization for Deep Learning: Training BERT in 76 minutes.

[64] Yuan, W., Murali, A., Mousavian, A., & Fox, D. (2023). M2T2: Multi-Task Masked Transformer for Object-centric Pick and Place.

[65] Yuan, W., Paxton, C., Desingh, K., & Fox, D. (2021). SORNet: Spatial object-centric representations for sequential manipulation. In *5th Annual Conference on Robot Learning*.

[66] Zhang, J., Kumor, D., & Bareinboim, E. (2020). Causal imitation learning with unobserved confounders. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20 Red Hook, NY, USA: Curran Associates Inc.

[67] Zhang, T., McCarthy, Z., Jow, O., Lee, D., Chen, X., Goldberg, K., & Abbeel, P. (2018). Deep Imitation Learning for Complex Manipulation Tasks from Virtual Reality Teleoperation. In *2018 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 5628–5635). Brisbane, QLD: IEEE.