Abstract of "Learning to Ground Natural Language Instructions to Plans" by Nakul Gopalan, Ph.D., Brown University, October 2019.

In order to intuitively and efficiently collaborate with humans, robots must learn to complete tasks specified using natural language. Natural language instructions can have many intentions: for example, they can specify a goal condition, or provide guidance to achieve a goal, or provide constraints that must be satisfied in achieving goals. Given a natural language command, a robot needs to ground the instruction to an action sequence executed in the environment, which satisfies the request along with all its constraints and guidances. This work addresses the problem of grounding natural language instructions to plans with various approaches, that all depend upon predicates from first order logic. We will first describe a hierarchical planner for planning in large state spaces. We will then present different procedures such as classification, sequence to sequence mapping and compositional parser learning, to ground natural language for hierarchical and constraint based tasks. These approaches allow for language commands specifying simple goal based specifications to richer temporal constraints with guidances. We finish with looking at an approach that learns symbols and their natural language groundings using demonstrations and instructions. This allows the agent to follow novel instructions in unseen environments without hand-specified symbols.

Learning to Ground Natural Language Instructions to Plans

by

Nakul Gopalan

MSc. Brown University 2015

MSc. T.U. Darmstadt 2012

B.E. Visvesvaraya Technological University 2008

A dissertation submitted in partial fulfillment of the

requirements for the Degree of Doctor of Philosophy

in the Department of Computer Science at Brown University

Providence, Rhode Island

October 2019

This dissertation by Nakul Gopalan is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date: ____________________ ____________________________

Stefanie Tellex, Director
Brown University

Recommended to the Graduate Council

Date: ____________________ ____________________________

Michael L. Littman, Reader
Brown University

Date: ____________________ ____________________________

Marie desJardins, Reader
Simmons University

Date: ____________________ ____________________________

George D. Konidaris, Reader
Brown University

Approved by the Graduate Council

Date: ____________________ ____________________________

Andrew G. Campbell

Dean of the Graduate School

Acknowledgements

This thesis was supported by the work of a lot of people. Firstly, I would like to thank Stefanie for inviting me over and introducing me to a rich set of robotics problems. I admire the amount of freedom that you gave me, while helping me become an independent researcher. I hope to use all the advice you have given me in the next phase of my career, and keep collaborating with you.

I am indebted to Michael Littman for being wise and providing stimulating inputs for a lot of my work. Michael has been a good listener and an amazing teacher. I have received a great deal of helpful advice from Marie DesJardins for my projects and the overall presentation. My writing is a lot better thanks to Marie. I am thankful to George Konidaris for the helpful early morning conversations about robotics and academia,, and also his help with my projects and presentation. Hopefully we can work together on more fun projects.

James MacGlashan helped me find my footing in projects and has provided helpful conversations since. Lawson L.S. Wong has helped me be precise and provide structure to my ideas and presentations. I am also thankful to Eric, Eddie, Dilip, Sidd, Mel and Vanya, for all the projects we worked on and the conversations we shared. I would also like to thank my other labmates and collaborators Thao, David W., John O., Jun, Carl, Deniz, Karthik, Ben B., Ben A., Matt, Matthew, Becky, Ariel, Izaak, Ifrah, Kyra, Emily, Steve, Stephen, Sam, Akhil, Jonathan, Nishanth, John W., Michael, Aaron, Josh and Kaiyu.

I am grateful for the support system provided by the CS department students Justin, Vikram, Thomas, John M., Johannes, Sasha, Amy, Betsy Eric, Preston and Michael Michaelidis. The T-Staff, and A-Staff at CIT helped me in innumerable occasions, I am especially thankful to Lauren Clarke, Suzanne Alden, Frank Pari, Max, Genie, Doug, Benjamin Nacar, and Paul. I am also thankful to my first friends at Brown: Sam and Guille.

I owe a lot to my friends in Jabalpur (Mini, Ankit, Golu, Munnu, Rani, Chinna, and Rahul), Bangalore (Ramesh, Siddharth, Sandeep, Kiran, Pushkar, Mandy, Kishore, Laxman, R.D., Anuraag, Varun, Girish, Vikhyath, Nitin K, Nitin C.S.,) and Darmstadt (Sriram, Varun, Sesha, Mohil, Ayshwarya, Naga, Vineet, Rahul, Abhisekh, Vibha, Vijay).

I would like to thank my aunts and my uncles and my grandparents for humouring me. I am grateful to my brother Kaushik for putting up with me. I owe more than just thanks to my parents for letting me explore the world, and backing me in all circumstances. I also want to thank Patricia for everything from walks to hugs. Hopefully I can make it up to all of you.

Thanks!
nakul

Contents

List of Figures

Chapter 1

Introduction

Language is an important part of human life. It allows us to express ourselves in different forms, for example discussing weather, or singing, or debating, etc. Language is so ubiquitous that people generally do not pay attention to how effortlessly it allows us to communicate, sometimes even allowing us to attempt to communicate with animals and inanimate objects with varying degrees of success. Machines are one class of objects with which we try to communicate regularly. Usually the form of this communication is via switches, or dials, or a programming language which might require special expertise or training. However, communicating with machines using natural language would allow us to collaborate with machines effortlessly. This dissertation examines the problem of communicating and collaborating and learning with machines, specifically robots, using natural language.

Natural language provides an intuitive interface with which humans can collaborate with machines. Given that people can already communicate "easily" with each other using natural language, communicating with robots or other artificial agents will be an intuitive interaction. Moreover, such an interaction would be welcome by the public. Roughly 66 million or 1 in 6 Americans own a smart-speaker, which at this point have only been around for 5 years [145]. People are willing to buy intelligent devices that interact with them via language. If there was a general purpose home robot platform, a natural language interface would not only be welcome, but might even be a necessity.

Despite the importance of language, humans are not born with the ability to use language effortlessly: and must learn to use language at infancy along with other faculties. Human infants learn different types of faculties related to perception, motor control and language learning. For example, a four day old infant can already discriminate between sounds of different languages; at one month they can discriminate between consonants; at 6-8 months of age they start to babble; at 10-12 months they produce their first words [50]. This language learning in the first year happens along with regular child development milestones as listed by the Center for Disease Control and Prevention manual [42]: for example, learning object permanence, paying attention to faces, smiling, reaching for toys, rolling, sitting, and walking. By eighteen months children can recognize object classes like telephone, spoons, name body parts, and can use objects like brushing their hair and helping dress themselves. This requires for the children to learn skills such as grasping and manipulation of objects. At two years they can follow simple two step instructions, make their own sentences, play make believe games, sort colors and shape. Three year old children can learn and generalize concepts they are observing for the first time and associate them with words [101, 148]. This learning is not only remarkably fast, but also seems to lack complete supervision or a set curriculum. The parents of children cannot always explicitly help in the learning of these concepts, or skills, or words, or grammar, etc.

We have yet been unable to create an artificial agent or robot that learns language, skills or concepts as fast as a human child. A plausible reason for this is our lack of understanding of the mechanisms for language acquisition in human children. Given the large number of concepts, skills and vocabulary, we cannot endow agents with the ability to express the meaning of every word or skill in a program or code that a robot can run. Moreover, with new technology, engineering or art, there will always be new concepts that an agent must represent. Hence our only option to have effective robotic collaborators with a general populace is to endow them with the capacity to learn and understand concepts, skills, and language. To bridge this gap this dissertation will first delve into the problem of learning to map language to symbolic representations that can be used for planning in robotics. We will then consider the problem of teaching tasks to agents by learning these representations using demonstrations and narration. This is an important problem as it allows even the general populace to specify and teach tasks to robots using an intuitive interface.

There are multiple hurdles in human-robot collaboration using natural language. Firstly, there is the symbol-grounding [53] problem, where the agent does not know which object, task, situation, etc. is being referred to in the interaction. For example, when a human says “pour the tea in a cup,” an agent must understand which physical objects the words “tea” and “cup” refer to, and which task the verb “pour” refers to. Secondly, the robot need not know all the concepts or symbols that are being referred to. For example, an agent might need to learn the concept of a cup before it can manipulate it, or map the word “cup” or “teacup” to the object. Thirdly, even if the agent knew all the objects and tasks, and knew the mapping from language to the words, there is still the problem of planning fast enough so the interaction seems natural. The focus of this dissertation is to present solutions to these hurdles like planning fast, language grounding, learning symbols for language grounding.

This work aims to answer the broad question of goal specification for a robot using natural language. We introduce and examine methods to identify the goal conditions that a natural language command implies. We also learn these goal specifications and their mappings to natural language together from demonstrations and natural language. This dissertation has four specific contributions to solve problems in human-robot collaboration using natural language:

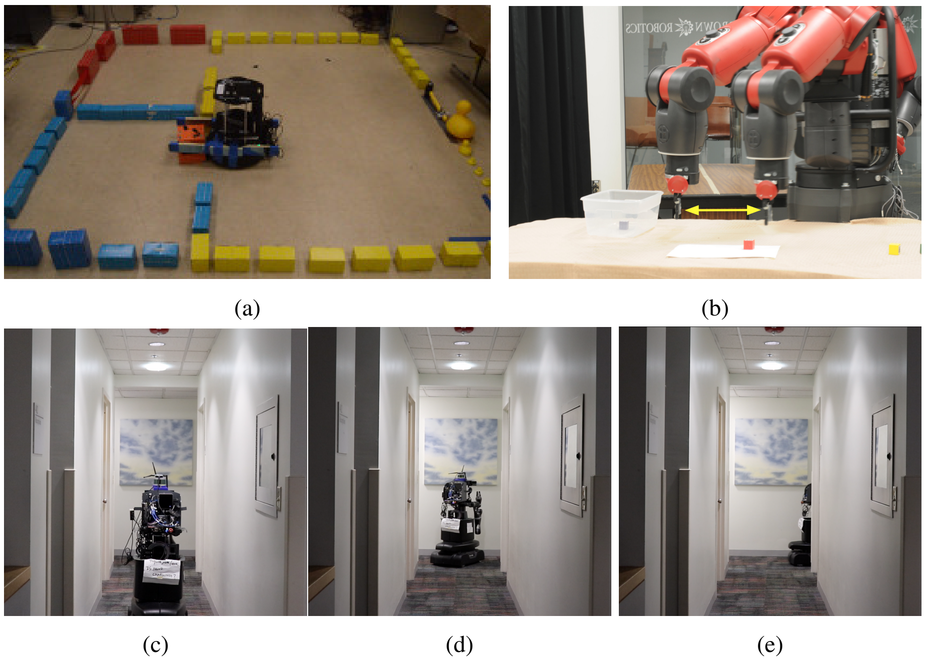


Figure 1.1: Images of different robots and behaviors demonstrated as part of this dissertation: a) a Turtlebot completing a simple goal based command to "Push the block into the blue room"; b) a Baxter robot performing a temporally constrained task to "Put all but the red blocks into the bin"; c), d) & e) a Movo robot learning the symbols to plan and navigate the corridors to complete the task "Take a right at the end of the corridor."

- A fast and reactive hierarchical planner specifically designed for the large state-action spaces encountered in task and motion planning. Fast and reactive planning is critical for a robot that interacts with humans in real time.

- We model language grounding for different types of behaviors or plans, from simple goal conditions to richer temporal constraints.

- We solve the problem of language grounding itself with different mechanisms: simple classification, deep sequence-to-sequence methods, traditional grammar based parser learning. We describe the pros and cons of each method in this dissertation.

- Learning goal and sub-goal conditions of a plan from scratch and mapping language to these symbols. This allows demonstration based learning on robots while keeping in mind the small number of learning interactions available to train the robot.

More importantly these solutions are designed to be applicable on robots, keeping in mind that robots need to be fast, reactive and learn with as few samples as possible. We implemented these solutions on a broad variety of platforms exhibiting different behaviors, as shown in Figure 1.1. We now present a broad chapter-wise overview of the dissertation.

The second chapter of this dissertation discusses related work in the relevant areas of natural language processing, learning in robotics and planning.

Chapter three discusses the problem of planning in domains with large state spaces. Specifically these state spaces are combinatorically large, that is, these problems have a lot of objects, and the agent must plan fast by ignoring objects that are unimportant in reaching the goal. Humans do this abstraction all the time. For example, while pouring tea, we do not pay attention to all the objects in the room or the world. We attempt to manipulate only the tea kettle and the cup in which the tea is being poured. For this we introduce the formalism Abstract Markov Decision Processes (AMDPs) [47].

Chapter four presents an initial approach to grounding language to sub-goals within the AMDP planning framework. We describe how mapping language to a hierarchical framework allows us to refer to goals at different levels of specificity. For example, this framework allows us to specify high level commands like “make me a pot of tea,” and granular commands like “pick up the spoon,” all within the same hierarchy.

Chapter five considers the problem of grounding language to non-Markovian task specifications. The task specifications have a preferred ordering or constraint, which make the goal specification depend on the entire trajectory and not just the terminal state of the agent. We can see these specifications in commands like “bring me the tea without spilling." The agent must now ensure that it never spills tea while delivering the cup. It is a condition that must be satisfied by the entire trajectory of the robot, and not just one particular state, which makes the goal condition non-Markovian.

Chapter six examines the role of compositionality in grounding language. Compositionality allows us to transfer concepts like color and shape to unseen tasks. This means if the robot knows the language grounding for a "blue mug" and a "green spoon, " it can ground a novel language task specification to "bring a green mug."

In all the previous chapters the agent knew the symbols that the language was grounded to in advance. They were hand-specified by an human designer. In chapter seven we ground language to learned symbols from demonstrations and instructions. We learn symbols that can transfer to novel environments when specifying goals on the robot.

We then conclude with a discussion about unresolved issues and open problems within the solutions provided in this dissertation.

Chapter 2

Related Work

While the various environments in which robots can be found are expanding to include the home, the workplace, and on the road, the most common interfaces for controlling these robots remain fixed in either tele-operation or directly programmed behaviors. Humans use natural language to communicate ideas, motivations, and goals with other humans. If we want robots to collaborate and solve tasks with humans the must understand these goals and then solve them. Two broad fields that are relevant in solving these problems: Computational Linguistics and Artificial Intelligence. Computational linguistics aims for computers or machines to understand or process natural language, written or spoken, and produce an output. The output can be a reply in a dialog, or the answer to a query by searching a database, or a translation to a foreign language, etc. Artificial Intelligence (AI), on the other hand, aims to provide machines or agents human capabilities of learning novel behaviours or concepts, and problem solving. It has been a long-standing goal in both of the aforementioned fields to have an agent understand and perform tasks specified through natural language. In this chapter we will cover related works in both these fields specific to our problem of learning to ground natural language to plans. We will first look at the historical perspectives in which these two fields have worked together, and then we will look at more modern approaches to grounding natural language to plans.

2.1 Good Old-Fashioned Artificial Intelligence (GOFAI)

One of the earliest works that combined natural language understanding and planning was SHRDLU by Winograd [152]. SHRDLU consisted of a simulated domain of blocks on a table. The artificial agent in the domain could answer questions about the state of the blocks, such as “Is the red triangle over the square?” and could also manipulate blocks to achieve specific configurations, such as “Place the green circle over the cube.” More importantly, the agent could be commanded by natural language to perform these behaviours. The agent parsed the natural language command to a query placed to the computer program. The rules for such parsing were hand-specified. Depending on the type of natural language command the agent replied with an answer about the state of the world or manipulated the blocks to achieve a goal configuration. There were similar works in the same period such as LUNAR [153], answered natural language queries about types of lunar rocks using hand-specified rules. However, what made SHRDLU special was that it could manipulate its environment based on a natural language query. The SHRDLU domain exposed a broad class of problems related to instruction following in embodied agents, which led to development of more general purpose techniques.

Slightly earlier to the development of SHRDLU (1966- 1971) was the development of the Shakey robot by Stanford Research Institute (SRI) International [118]. Shakey, shown in Figure 2.1, was an important project for robotics as it was for the first time researchers developed a general purpose robotic platform. The Shakey project led to the development of some very important planning algorithms like Stanford Research Institute Problem Solver (STRIPS) [41] and A* [54], important results in computer vision like the use of Hough Transform features [37], important robot control software

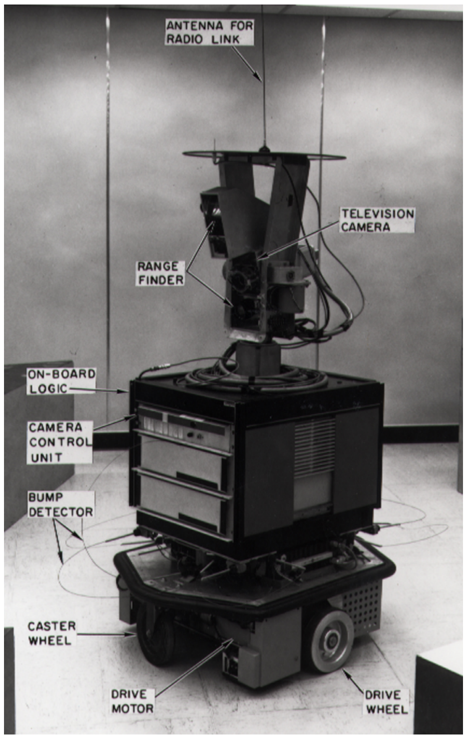


Figure 2.1: Image of Shakey the robot from the original Tech Report [118]
.

and some demonstrations involving natural language.

We will discuss the planning algorithms used in Shakey here in more detail here as they are a focus of this dissertation. STRIPS allowed the robot to plan in large continuous state spaces by breaking down goal conditions into sub-goals that can be achieved by operators or actions. The actions themselves had pre-conditions and post-conditions. The planner would learn to sequence an action's post-condition as the next action's pre-condition, until the goal condition was reached. STRIPS required abstracting goal and sub-goal conditions present in the domain, which was achieved with propositional variables or predicates. The propositional variables were used to specify the pre- and post-conditions of actions, and the goal conditions. A* was used for performing the physical path planning on the robot. A* is a heuristic search algorithm, based on Dijkstra's shortest path algorithm [34], allowing the robot to travel from one location to another, by sequencing low level motor control actions. The robot itself was present in a room that was precisely measured and had objects like blocks placed in specific locations. Moreover, all the predicates used by planners were pre-specified and hand designed.

Shakey took in goal-based commands by users and converted them into a goal specification. This goal specification was then achieved by the combination of STRIPS and A* planners. The goal commands were specified using an English-like natural language, and were converted into a formal specification using the ENGROB system [31]. Quoting the tech report by Nilsson: "Each of these tasks is stated in English and entered into the system via teletype. The first task is stated as "USE BOX 2 TO BLOCK DOOR DPDPCLK FROM ROOM RCLK." This statement is converted by the English language system ENGROB to a goal expressed by a well-formed formula (wff) of the first-order predicate calculus: BLOCKED(DPDPCLK,RCLK,BOX2). " The first goal of ENGROB [31] as specified by Coles was to "translate English statements, questions, and commands into a formal language based on the first-order predicate calculus." These English language goal commands were translated into a formal specification using a series of rules, that mapped snippets of parsed natural language command to snippets of the well formed predicate language. The system also checked for the correctness of the translated query to check if the command is possible in the current environment. We can consider this translation as going from English language to its meaning representation in a more structured language, which is studied by the branch of Semantics in Computational Lingustics.

As we discussed earlier, methods like LUNAR [153] and SHRDLU [152] convert a natural language question into a query that a computer could answer. They were developed for different applications such as LUNAR for answering questions about lunar rocks from a database, Chat-80 [146] for answering questions from a geographical database, and Tina [132] which was more general purpose and provided general rules that different domains could follow for a natural language interface. The query that was being placed to the computer was structured with first order predicates, and used syntactic rules from English to infer the functions in predicate logic and their arguments. This idea of using the English (or natural) language's syntax along with its constituent words to infer a sentence's semantic meaning comes from an approach developed by Montague [114]. Montague defines the syntactic and semantic rules for a large fragment of the English language [114, 113], allowing us to write formal semantic meanings, using higher order predicate language, for a large fragment of the English language. However, we still had to write the rules for translating the English language into its semantic meaning, which can then be used as a query for a computer or a goal condition by a robot. Early methods like LUNAR, SHRDLU, Chat-80 and Tina used hand-specified rules to translate between English and the formal semantic meaning. These hand-specified rules made such early systems brittle, and did not allow them to generalize to larger and more realistic domains.

Besides the problem of mapping natural language to its meaning, Harnad [53] defined the problem of symbol grounding. Symbol grounding defines the problem of mapping tokens or words in a language to a referent, that is, objects or sensory attributes in the real world, or an abstract concept, that the language is referring to. A more realistic example can be when Shakey the robot is asked to "USE BOX 2 TO BLOCK DOOR DPDPCLK FROM ROOM RCLK," it needs to know which objects the tokens "DOOR" or "ROOM" refer to. In this dissertation we look at grounding language to objects in the world, and behaviours that a robot is expected to perform.

While GOFAI helped define a lot of the fundamental problems of AI as a field the solutions it proposed were brittle and did not scale very well beyond the domains that the individual systems were developed upon. Part of the problem was the lack of data available from different sources and domains; however, we also lacked models rich enough to use these forms of data. As a response, recent approaches looked to use a lot more data and more complex models to learn more generalizable behaviors. This was partially enabled by the internet as we could collect images, language, and video data more easily. In the next section we discuss some of the recent methods for solving AI problems related to language and planning.

2.2 Recent Approaches for Language Grounding and Planning

There are multiple ways to approach the problem of learning to ground language to plans. In this section we first discuss methods that use or learn abstractions for planning. These methods were not developed with language in mind, but they form the bedrock of an agent that can follow instructions. Moreover, these abstractions might provide us with symbols abstracted to which we can ground language. We will then look at methods that ground language to trajectories, goals, or objects specified by a human user.

2.2.1 State and Action Abstractions

Human beings make abstractions about concepts and actions when solving tasks. For example, when we are trying to pick an object, we do not think about every single muscle contraction that our body makes, but only the overall "action" of reaching the cup and picking it up. Similarly we can disregard all objects in the world except the cup, allowing us to create a reduced or "abstracted" state of the world. Such abstractions not only allow us to plan over longer horizon tasks, but also allow us to learn novel tasks more easily. In this section we will look at the use of abstractions in an agent's state and actions to learn and or plan faster.

Action Abstraction

An action abstraction can be thought of as a sequence of actions that have a single label. The simplest way to consider action abstraction is to consider macro-actions [41], which are just a fixed sequence of actions. Such actions are useful if the agent's world is deterministic and the agent can plan over multiple macro-actions to get to the goal.

A more holistic approach for action abstraction is the options framework [138]. An option has a policy that provides an action that the agent can take for states it encounters while performing the option. Such a policy allows the agent to handle unexpected states that might arise due to stochasticity. Along with a policy, options have an initiation set of states from where an option can be executed, and a termination set of states which when reached indicate cessation of the option's execution. This allows options to be accessed in only from the initiation set, which in turn reduces the number of options an agent might have to consider in any given state. Learning an option is an open problem and involves learning its initiation set, termination set, and policy. Different approaches have looked at learning options using human demonstrations [77] or autonomously [80].

In robotics a common approach to modelling action abstractions is Dynamical Movement Primitives (DMPs) [61]. DMPs model trajectories either by demonstration or by learning from scratch. DMPs consists of a second order dynamical system, similar to a spring-mass physical system, that stably proceeds towards the goal of the trajectory being learned. This stable dynamical system is then modulated using a parameterized linearly weighted function to create the required contour of the example trajectory. Hence, DMPs provide an action abstraction that is both stable for use with robotics [61], and learnable [117, 75].

State Abstraction

State abstraction considers the problem of merging states together that have some similar property or feature. Common approaches have looked at merging states together that have the same actions or costs or distance from goal as other states [90]. Other approaches consider feature spaces and repetitions of features while solving a problem, which is especially useful in continuous state-action spaces. These features can be Tile Coding [137], or Fourier Basis Functions [78], or learned features using a neural network [112]. Other approaches have tried to use the termination set of an action abstraction as a form of state abstraction [79].

Methods that use a State *Action Abstraction Hierarchy*

Another common approach in learning or planning in large action spaces is to construct a hierarchy with both state and action abstraction. MAXQ [33] is one of the first approaches that provided a handcrafted state-action hierarchy and learned a policy over this hierarchy. DetH* [12] is another top-down hierarchical approach that learns a two-level hierarchy by breaking the state space into macro-states and planning deterministically over the macro-states. Hierarchical Task and Motion Planning in the Now [68] provides a controller hierarchy with state abstraction to perform task and motion planning. Similarly Hierarchical Task Networks (HTN) [39] allowed planning in deterministic domains with abstracted pre- and post- conditions allowing deterministic planning in large domains. There have been other works attempting to learn HTN hierarchies for efficient planning [115].

Hence there has been plenty of modern work attempting to specify and learn abstractions for planning and learning. These approaches do not attempt to map language to these learned abstraction hierarchies, but instead desire to learn abstractions that make planning efficient. In the next section we will look at approaches that perform language grounding to abstract concepts such as goals, objects or behaviors. In most cases such abstractions are pre-specified, and are not subject to their suitability for planning.

2.2.2 Language Grounding

The Language Grounding problem in the GOFAI era attempted to map language to fixed symbol spaces that were observed, with queries whose grammar was fixed and known. Modern methods attempt to accept more free-form language as input while also learning the underlying symbolic representation to which language can be mapped.

Semantic parsing was initially used to solve instruction following problems [99, 6]. The goal of the field of semantic parsing was to convert natural language to interpretable programs that can be run on computers. This requires goal conditions and sub-goal conditions to be pre-specified. For example, if an agent was asked to "go to the chair," the agent would need a pre-specified goal condition that returns true when the agent is next to the chair. These hand specified goal conditions are labor intensive to design.

Some progress was made in learning symbols from data for instruction following [140, 57]. These methods specifically learn symbols that can be used as constraints and goal conditions within a probabilistic planner. They also use syntactic information of the input sentence to ground verbs and referent objects correctly. While Tellex et al. [140] attempt to map language directly to a constraint satisfying output trajectory, Howard et al. [57] attempt to map language to constraints that can then be used by a planner to generate a trajectory, reducing the size of the inference problem. Such an approach has also been used to ask questions about missing parts of a plan [141].

Some recent work has tried to use neural approaches for sequence to sequence mapping from language to trajectories [3, 107]. These approaches have not been demonstrated on robots, and might require an infeasible amount of data.

In other related work, Matuszek et al. [103] learned mappings from sentences to attributes learned from sensor data directly using a joint language and object attribute model. Such a learning method required very little language interaction data with the objects to teach object attributes. Recently end-to-end approaches have allowed mapping language to object attributes in image data [70, 144]. Such end-to-end approaches are generally performed on large databases of images collected on the internet.

2.3 Background

In this section we define some formalisms that we use throughout the dissertation.

2.3.1 Markov Decision Processes (MDP)

Throughout this work we consider the problem of an agent interacting with an environment modeled as a Markov Decision Process (MDP) [13]. An MDP is defined by a six-tuple ($\mathcal{S}$,$\mathcal{A}$, $\mathcal{T}$, $\mathcal{R}$, $\mathcal{E}$, γ), where $\mathcal{S}$ is the environment state space; $\mathcal{A}$ is the agent's action space; $\mathcal{T}(s, a, s')$ is a function defining the transition dynamics (i.e., the probability that a transition to state s' will occur after taking action a in state s); $\mathcal{R}(s, a, s')$ is the reward function, which returns the reward that the agent receives for transitioning to state s' after taking action a in state s; $\mathcal{E} \subset \mathcal{S}$ is a set of terminal states that, once reached, prevent any future action; and γ is the discount factor that differentiates immediate and future rewards.

2.3.2 Planning Within an MDP

The goal of planning in an MDP is to find a *policy*—a mapping from states to actions—that maximizes expected future discounted reward. Classical planning approaches like Value Iteration applied the Bellman [13] equations to compute the optimal value of a state. The value of a state can be defined as its maximum expected future discounted reward given the optimal sequence of actions. The optimal value $V^*(s)$ needs to satisfy the Bellman Equation, that is:

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} T(s, a, s')(R(s, a, s') + \gamma V^*(s')). \tag{2.1}$$

A practical method of computing the optimal value function is to use the Bellman Backup [13], which uses estimates of the value function from the previous iteration:

$$V_{n+1}(s) = \max_{a \in A} \sum_{s' \in S} T(s, a, s')(R(s, a, s') + \gamma V_n(s')), \tag{2.2}$$

where n is the number of iterations. This algorithm is called Value Iteration and it is run until the values in V converge.

Bounded Real Time Dynamic Programming (BRTDP) [**?**] is a more efficient planner that we use instead of value iteration. More importantly BRTDP is an anytime algorithm, so we do not have to wait until convergence as in the case of value iteration. BRTDP is a search based algorithm that maintains both an upper and a lower bound on the true optimal value function from a state. The algorithm then explores actions with the widest gaps between the upper and lower bounds. The algorithm uses the values of upper bounds to return a greedy policy.

2.3.3 Object Oriented MDPs

Instead of using vanilla MDPs, we use a factored representation in the form of the Object Oriented-MDPs (OO-MDPs) []. An OO-MDP adds sets of object classes and propositional functions; each object class is defined by a set of attributes and each propositional function is parameterized by instances of object classes. Further, we use predicates in this work to represent constraints, sub-goals, and objectives specified in instructions. A predicate based representation is rich enough to allow for planning efficiently in domains with stochasticity and/or non-Markovian goal specifications.

2.3.4 Neural Sequence to Sequence Mapping for Translation

Apart from MDPs, some of the other related work uses deep neural network language models to perform language grounding. Deep neural networks have had great success in many natural language processing (NLP) tasks, such as traditional language modeling [15, 108, 109], machine translation [27, 29], and text categorization [62]. One reason for their success is the ability to learn meaningful input representations [15, 110]. These "embeddings" are dense vectors that not only uniquely represent individual words (as opposed to otherwise sparse approaches for word representation), but also capture semantically significant features of the language. Another reason is the use of recurrent neural networks (RNNs), a type of neural network cell that maps variable length inputs (i.e. commands) to a fixed-size vector representation, which have been widely used in NLP [27, 29, 154]. We will discuss RNNs and their functionality in more detail in Chapter 5.

With this we finish our review of related work and background materials. Next we will discuss an approach that attempts to plan fast in a stochastic setting using hierarchies.

Chapter 3

Hierarchical Planning with Abstract Markov Decision Processes

This chapter has been previously presented at ICAPS 2017 as "Planning with Abstract Markov Decision Processes," with Marie desJardins, Michael L. Littman, James MacGlashan, Shawn Squire, Stefanie Tellex, John Winder and Lawson L. S. Wong [47]. The idea for such hierarchies was developed before I joined the project. I was the primary driver of this project and my contributions were programming the hierarchies and evaluating these hierarchies in simulation and on the robot.

Planning in unstructured, multifaceted environments such as factory floors or kitchens, is extremely challenging due to the large state and action spaces [18, 73]. The state–action space in these domains grows combinatorially with the number of objects. Typical planning methods require the agent to explore the state–action space at its lowest

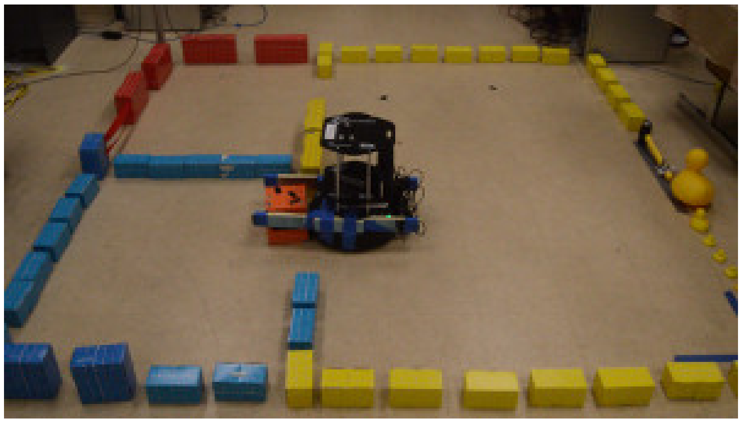


Figure 3.1: The continuous object manipulation domain, where the Turtlebot agent needs to fetch a block into a goal room. The task is solved online with an AMDP hierarchy that provides low-level actions to solve the abstractly defined task to "push the block into the blue room". The AMDP planner shows reactive control when moving the blocks to deal with stochasticity of the environment and the robot's controllers.

level, resulting in a search for long sequences of actions in a combinatorially large state space. For example, cleaning a room requires arranging objects in their respective places. A naive approach for arranging objects might have to search over all possible states by placing all objects in all possible locations, resulting in an intractable inference problem with increasing objects.

One promising approach is to decompose planning problems in such domains into a network of independent subgoals. This approach is appealing because the decision-making problem for each subgoal is typically much simpler than the original problem. There are two ways in which the decision problem can be simplified. First, instead of selecting between actions, the agent can select between subgoals that are recursively solved, decreasing the search depth. Second, the state representation of the world can be compressed to include only information that is relevant to the current decision problem. Importantly, planning algorithms for each subproblem can be custom-tailored, allowing each goal to be solved as efficiently as possible.

We propose *Abstract Markov Decision Process* (AMDP) hierarchies as a method for reasoning about a network of subgoals. AMDPs offer a model-based hierarchical representation that encapsulates knowledge about abstract tasks at each level of the hierarchy, enabling much faster, more flexible top-down planning than previous methods (see Section 3.1). An AMDP is an MDP whose states are abstract representations of the states of an underlying *environment* (the ground MDP). The actions of the AMDP are either primitive actions from the environment MDP or subgoals to be solved. An AMDP hierarchy is an acyclic graph in which each node is a primitive action or an AMDP that solves a subgoal defined by its parent. The main advantage of such a hierarchy is that *only* subgoals that help achieve the main task need to be planned for; crucially, plans for irrelevant subgoals are never computed. Another desirable property of AMDPs is that agents can plan in stochastic environments, since each subgoal's decision problem is represented by an MDP. Moreover, each subgoal can be independently solved by any off-the-shelf MDP planner best suited for solving that subgoal. Finally, if each AMDP's transition dynamics accurately models the subgoal outcomes, then an optimal solution for each AMDP produces a recursively optimal solution for the whole problem; if the transition dynamics are not accurate, then the error associated with the overall solution can still be bounded (Section 3.2.1).

For example, consider the Taxi problem [33] shown in Figure 3.3a and its AMDP hierarchy in Figure 3.2. The objective of the task is to deliver the passenger to their goal location out of four locations on the map. The subgoal of Get Passenger, which picks up the passenger from a source location, is represented by an MDP, with lower-level navigation subgoals, Nav(R), and a passenger-pickup subgoal, Pickup. The state space to solve the Get Passenger subgoal need not include certain aspects of the environment such as the Cartesian coordinates of the taxi and passenger. To solve this small MDP when picking up a passenger at the Red location, it is unnecessary to solve for the subpolicy to navigate to the Blue location. Our hierarchy enables a decision about which subgoal to solve without needing to solve the entire environment

MDP. Moreover, since the tasks are abstractly defined (for example, “take passenger to blue location”), changing the task description from the “blue” to the “red” location is straightforward, and users do not have to directly manipulate the reward functions at each level of the hierarchy. This abstraction is useful in robotics, as human users can simply change the top-level task description and the required behavior will be achieved by the hierarchy.

In the next section, we discuss the relationship of our work to previous hierarchical planning methods. We then formally describe an AMDP hierarchy and planning over AMDPs (Section 3.2). Our results (Section 5.4) show that AMDPs use significantly less planning time than base-level planners and MAXQ to complete tasks in domains with large state spaces. In smaller domains, such as the Taxi problem, AMDPs do as well as a fast base planner like Bounded Real Time Dynamic Programming (BRTDP) [106]; however, in large domains such as Cleanup World [98]—which has multiple manipulable objects that require long plans—AMDPs can produce plans orders of magnitude faster than all other planners we have examined. We also implemented a continuous mobile manipulation domain on a Turtlebot, as shown in Figure 4.1, which is solved using an AMDP hierarchy that spans all the way from low-level control actions to very high-level abstract goals, with exceptionally efficient planning. Our approach enables the Turtlebot to choose actions at the lowest level of the hierarchy at 20Hz. Further, if the environment changes (for example, if a target object moves), the robot instantly replans and completes the task. AMDPs are the first hierarchical method to allow real-time planning on abstractly specified object manipulation tasks in stochastic environments with continuous variables.

3.1 Related Work

Subgoals and abstraction provide mechanisms for allowing an agent to efficiently search large state/action spaces. One of the earliest reinforcement learning (RL) methods using the ideas of subgoals and abstraction is the MAXQ model [33]. MAXQ decomposes a “flat” MDP into smaller subtasks, each accompanying a subgoal and potentially a state abstraction that is specific to the subtask’s goal. A subtask is a sequence of actions that is used to complete a subgoal. MAXQ constrains the choice of subtasks that can be chosen in its hierarchy, depending on the context or parent task. MAXQ can also use relevant state abstractions for each context, which speeds up learning. MAXQ hierarchies were originally designed for use in model-free RL. However, they have been used in planning settings [35]. An AMDP has the form of a non-primitive subtask of a MAXQ hierarchy, but also includes an “abstract” transition and reward function defined over the subtask’s children. AMDPs are therefore model-based, in contrast to MAXQ. In AMDPs, these abstract transition and reward functions enable planning purely at the abstract level, without needing to further descend to lower-level subtasks or primitive actions until execution. In this work, we focus on the planning aspects of AMDP hierarchies, and assume these abstract transition and reward functions are given; later on in this dissertation we look at learning these subgoal conditions from data.

A major limitation of MAXQ is that value functions over the hierarchy are found by processing the state–action space at the lowest level and backing up values to the abstract subtask nodes. This *bottom-up* process requires full expansion of the state–action space, resulting in large amounts of computation. By contrast, Figure 3.2 shows the Taxi task hierarchy, with shaded cells indicating the nodes that get expanded or solved by an AMDP planner from the initial state in the environment. AMDPs model each subtask’s transition and reward functions locally, resulting in faster planning, since backup across multiple levels of the hierarchy is unnecessary. This *top-down* planning approach decides what a good subgoal is before planning to achieve it. In this work, because we want to compare planning speed across methods, we have provided the transition and reward models for subtasks to the AMDP planner.

The concept of using a subtask model for computing a policy has also appeared in previous work, such as R-MAXQ [65], a model-based learning algorithm for MAXQ. However, in R-MAXQ, the transition and reward functions violate our model-abstraction requirement: their local models are not self-contained. Instead, the effects of a subtask's child are computed recursively, requiring planning for each child and descendant for each state visited in the subtask. Learning transition functions in R-MAXQ requires recursion all the way to the base level. AMDPs, by contrast, open the door to new learning algorithms that operate separately at each level, potentially requiring much less data and computation to learn a model. Moreover, within most of these methods, there are no means to use specialized planners or heuristics specific to a subtask.

Temporally extended actions [105] and options [138] are other commonly used bottom-up planning approaches, that encapsulate reusable segments of plans into a more tractable form, and are another way to represent subgoals. Generally, options are used as pre-computed policies for planning; once available, options can speed up planning in large domains. However, these pre-computed policies may themselves be hard to find. When policies for options are planned for with the entire task, they are as slow as other bottom-up approaches. Moreover, although temporally extended actions can decrease plan length and complexity, their inclusion also increases the branching factor of search, and can lead to dramatic increases in planning time [66]. Recently, Konidaris [76] discussed creating MDPs at multiple levels of the hierarchy with options, which were used as precomputed policies. This hierarchy plans for a task completely at one of the levels of the hierarchy, instead of planning only for the subgoals that are needed across the hierarchy. Construction of these sets of options at multiple levels requires searching through the entire hierarchy, bottom-up, which is slower than an AMDP-based approach.

Top-down approaches have been used previously for planning and have shown improvements in planning speeds when compared to bottom-up methods. Hierarchical

Dynamic Programming (HDP) [11] is a top-down approach that was used for planning in navigation problems. However, HDP is specific to navigation domains, and relies on value iteration [14] for each subtask, whereas AMDPs allow any suitable planner to be used for each subgoal. For example, for a navigation subtask, A* [54] might be an appropriate method, but for a subtask of picking a particular object in a room full of different objects, BRTDP [106] might be a better choice. Furthermore, the state aggregations formed by HDP are based on the adjacency of physical locations, which do not transfer to mobile manipulation completely, since the abstraction is not based on completion of subgoals such as picking up passengers or objects. Nevertheless, the previous application of HDP to robotic navigation provides strong evidence that the AMDP approach generalizes to very large robotics-relevant problems; to support this claim, we demonstrate the use of AMDPs in a full-stack mobile-robot controller.

Angelic hierarchical planning [102] also uses a top-down approach for planning. However, its use is limited to deterministic domains, and it does not offer the modularity of using different planning algorithms at each node. DetH* [12] is another top-down hierarchical approach that learns a two-level hierarchy by breaking the state space into macro-states and planning deterministically over the macro-states. Domains like Taxi have stochasticity in the higher levels of abstraction, so DetH* cannot be directly used. Further, DetH* does not allow planning in MDPs with a continuous state space. AMDPs are a generalized top-down planner, which can solve in real time a variety of stochastic domains with combinatorial and/or continuous state variables, and can completely plan trajectories for a robot to perform mobile manipulation.

3.2 Abstract Markov Decision Processes

We consider the problem of an agent interacting with an environment that is modeled as an MDP [13], which has previously been defined in Section 2.3. The goal of planning in an MDP is to find a *policy*—a mapping from states to actions—that maximizes the

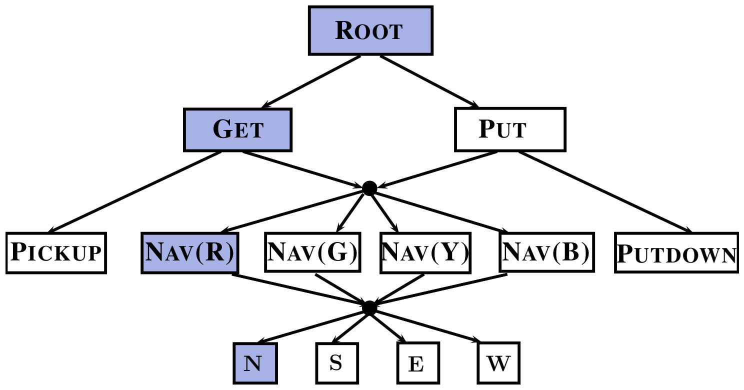


Figure 3.2: The Taxi AMDP hierarchy. Nodes indicate subgoals which are solved using an AMDP or a primitive action. The edges are actions belonging to the parent AMDP. Shaded nodes indicate which subgoals are expanded by AMDPs in a given state. By contrast, bottom-up approaches like MAXQ [33] expand all nodes in the figure. These savings result in significant total planning computation gains: AMDP planning requires only 3% of the number of backups that MAXQ requires for the Taxi problem.

expected future discounted reward.

Solving an MDP can be quite challenging, so we introduce the concept of an *Abstract MDP* (AMDP) to simplify this process. An AMDP is an MDP in its own right, but captures higher-level transition dynamics that serve as an abstraction of the lower-level environment MDP with which the agent is interacting. Formally, we define an AMDP as a six-tuple ($\tilde{\mathcal{S}}$, $\tilde{\mathcal{A}}$, $\tilde{\mathcal{T}}$, $\tilde{\mathcal{R}}$, $\tilde{\mathcal{E}}$, F). These are the usual MDP components, with the addition of $F : \mathcal{S} \to \tilde{\mathcal{S}}$, a *state projection* function that maps states from the environment MDP into the AMDP state space $\tilde{\mathcal{S}}$. Additionally, the actions ($\tilde{\mathcal{A}}$) of the AMDP are either primitive actions of the environment MDP, or are associated with subgoals to solve in the environment MDP. The transition function of the AMDP ($\tilde{\mathcal{T}}$) must capture the expected changes in the AMDP state space upon completion of these subgoals. With these action and state semantics, an AMDP, in effect, defines a decision problem over subgoals for the environment MDP.

Naturally, each subgoal for a task must be solved. However, even a single subgoal might be challenging to solve in the environment MDP. Therefore, we introduce the concept of an AMDP hierarchy $H = (V, E)$, which is a directed acyclic graph (DAG) with labeled edges. The vertices of the hierarchy V consist of a set of AMDPs $\mathcal{M}$ and the set of the primitive actions $\mathcal{A}$ of the environment MDP. The edges in the hierarchy link multiple AMDPs together, with the edge label associating the action of an AMDP with either a primitive environment action or a subgoal that is formulated as an AMDP itself. Consequently, an AMDP hierarchy recursively breaks down a problem into a series of small subgoals.

For example, consider the Taxi problem, where the objective is to ferry a passenger to their goal location with AMDP hierarchy shown in Figure 3.2. The AMDP for the Get Passenger subgoal has access to the location-parameterized abstract action Nav(i) and the primitive action Pickup. The abstract states and projection function in this AMDP remove the Cartesian coordinates of the taxi and passenger, replacing both with a discrete value corresponding to one of the colored destinations. The termination states consist of any states in which the passenger is in the taxi. The reward function is simply defined to return $+1.0$ when the passenger is picked up. The navigate to Red subgoal, Nav(R), which is referenced by Get Passenger, is itself another AMDP. In this case, its actions consist only of the primitive environment actions, North, South, East, and West. Its states and state projection function exclude variables related to the passenger and its terminal states are defined to be states in which the taxi is at the Red location. The reward function returns 1 when the Red location is reached, 0 otherwise. The AMDP definition for the other Nav subgoals is similar.

Although there are many possible ways to define state abstractions, a useful heuristic is subgoal-based state abstraction. Subgoals specifically allow the condition of Result Distribution Irrelevance [33], in which the state space collapses into a few states at the end of a temporally extended action, behaving as a "funneling action," allowing a concise representation of the abstract state space.

When a parent AMDP invokes a lower-level node, it expects that particular subgoal

Algorithm 1 Online Hierarchical AMDP Planning

```
function SOLVE($H$)
    GROUND($H$, ROOT($H$))
function GROUND($H, i$)
    if $i$ is primitive then                          ▷ recursive base case
        EXECUTE($i$)
    else
        $s_i \leftarrow F_i(s)$                       ▷ project the environment state $s$
        $\pi \leftarrow$ PLAN($s_i, i$)
        while $s_i \notin \mathcal{E}_i$ do           ▷ execute until local termination
            $a \leftarrow \pi(s_i)$
            $j \leftarrow$ LINK($H, i, a$)            ▷ $a$ links to node $j$
            GROUND($H, j$)
            $s_i \leftarrow F_i(s)$
```

to be achievable; there is currently no mechanism for backtracking. This is potentially problematic if the lower-level AMDP's failure conditions are not expressible in the higher-level state abstraction, since the parent would then have no means to reason about the failure. We currently avoid this problem by ensuring that termination sets include all failure conditions, and designing higher-level state spaces that are capable of representing such failures. However, if the abstractions are themselves imperfect (e.g., learned), then we would likely need to address the problem of failure recovery.

3.2.1 Planning in AMDPs

In this section, we describe how to plan with a hierarchy H of AMDPs. The critical property of our planning approach is to make decisions online in a top-down fashion by exploiting the transition and reward function defined for each AMDP. In this top-down methodology, planning is performed by first computing a policy for the root AMDP for the current projected environment state, and then recursively computing the policy for the subgoals the root policy selects. Consequently, the agent never has to determine how to solve subgoals that are not useful subgoals to satisfy, resulting in significant performance gains compared to bottom-up solution methods. This top-down approach does require that the transition model and reward function for each AMDP are available. In this work, we assume they are given by a designer to demonstrate the power afforded by this top-down approach. However, in the future, we plan to investigate how to learn these abstract transition models and reward functions by using model-based reinforcement learning approaches. Our work is analogous to hierarchical task network (HTN) planners, which use designer-provided background knowledge to guide the planning process, but can also be learned from observation [115]

Pseudocode for online hierarchical AMDP planning is shown in Algorithm 1. Planning begins by calling the recursive *ground* function from the root of H. If node i passed to the ground function is a primitive action in the environment MDP, then it is executed in the environment. Otherwise, the node is an AMDP that requires solving. Before solving it, the current environment state s is first projected into AMDP i's state space with AMDP i's projection function F_i. Next, any off-the-shelf MDP planning algorithm associated with AMDP i is used to compute a policy. The policy is then followed until a terminal state of the AMDP is reached. Following actions selected by the policy for AMDP i involves finding the node the actions links to in hierarchy H, and then calling the ground function on that node. Note that after the ground function returns, at least one primitive action in the environment should have been executed. Therefore, after ground is called, the current state for the AMDP is updated by projecting the current state of the environment with F_i.

AMDP planning can be substantially faster than general MDP planning for two reasons. First, the hierarchical structure allows high-level AMDP actions to specify subgoals for the lower-level MDPs, dramatically decreasing search depth. Second, each subgoal can take advantage of strong heuristics or planning knowledge that is not easily incorporated into a planning algorithm solving the global problem. For example, in the Taxi problem, the navigation subgoals enable A* to be used with a strong heuristic, such as Euclidean distance from the goal, to help complete the subtasks. By contrast, bottom-up hierarchical algorithms, including MAXQ, do not afford the ability to use custom planning algorithms for each subgoal and require expanding subgoals that will ultimately prove unhelpful to solving the task.

Since hierarchical planners constrain which decisions can be made, optimal solutions to them do not necessarily maximize the expected future reward for the underlying environment MDP. Dieterrich [33] distinguishes between optimality for the environment and several notions of optimality that are constrained by the hierarchy. A *hierarchically optimal* policy is one that achieves the maximum reward at the base level, subject to the constraint that actions are consistent with the given hierarchical structure. By contrast, *recursive optimality* is a local concept in which the policy is optimal at each level of the hierarchy; that is, the policy at each node is optimal given the policies of its children. A MAXQ policy is recursively optimal [33].

Actions in an AMDP are chosen to be optimal with respect to its level of abstraction, given any goal conditions from the level above. The notion of optimality within each level of abstraction is weaker than recursive optimality, since the abstract MDPs do not query the transition and reward functions of the flat MDP to learn a correct semi-MDP (SMDP) [138] model of the task while planning. However, with appropriate design a planner optimal at each abstract level can be as effective as a recursively optimal planner. An AMDP hierarchy without state abstraction is equivalent to solving the base-level MDP with temporally extended actions in the form of subtasks. Without state abstraction, if the transition and reward functions for each AMDP respect the true multi-timestep transitions for completing those subgoals in the environment, and each AMDP is optimally solved, then the solutions of the hierarchical MDP planner will be recursively optimal by definition [33]. Further, Brunskill and Li [23], in Lemma 5, prove that bounded error in an SMDP transition and reward function leads to bounded error in its value function and Q function. That is:

$$|Q_1^*(s,a) - Q_2^*(s,a)| \leq \max_{s,a} \epsilon_{s,a} \ , \tag{3.1}$$

where $Q_1^*(s,a)$ and $Q_2^*(s,a)$ are the optimal Q values of the state–action pair under two different SMDP models, and ϵ is a polynomial function of the difference in the transition and reward functions of the two SMDPs. Hence, without state abstraction, AMDPs with bounded error in their transition and reward functions would have bounded errors in their value function. Consequently, using imperfect AMDP models can still produce reasonable results.

3.2.2 Example AMDP Hierarchies

As examples of source (environment) MDPs and AMDP hierarchies, we present hierarchies for Cleanup World [98] and the "fickle taxi" domain used by Dieterrich [33] for MAXQ. We use the object-oriented MDP (OO-MDP) formalism to represent these domains [36]. These hierarchies are extensively defined in the supplementary material.[1]

Fickle Taxi

The Fickle Taxi problem was used by Dieterrich [33] to introduce the MAXQ planning hierarchy. We created AMDP analogs for the hierarchical structure used in the original MAXQ work. In the flat (level 0) MDP, there are six actions: north, south, east, west, pickup passenger, and drop passenger. States in the flat MDP consist of the physical grid-cell coordinates of passengers, taxi, and locations. Locations and passengers also have a color attribute, and the taxi has an attribute pointing to its current passenger, if any. Movement actions of the taxi are noisy with a probability of 0.2, and the passenger changes their destination after getting in the taxi with a probability of 0.3. The stochasticity of the fickle passenger penalizes hierarchical methods that use

[1] http://h2r.cs.brown.edu/wp-content/uploads/2017/03/AMDP-ICAPS-SupplementaryMaterial.pdf

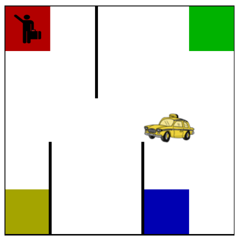
temporally extended navigation actions, which terminate only when the taxi reaches the goal location.

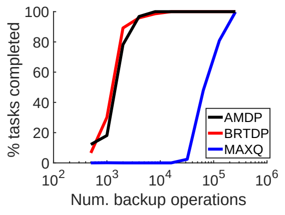
At level 1 of the hierarchy, the abstract actions are Pickup, which puts a passenger into the taxi if they are at the same location; Putdown, which drops a passenger off at the current location; and Nav(i), which drives the taxi to location i. The transitions have been defined deterministic at this level; however, they can be defined stochastically according to the passenger's fickle behavior. The state for this AMDP abstracts away the physical coordinates of passengers, taxi, and goal locations, replacing them with level 1 passenger and taxi variables, each containing one of the four possible locations or an "on-road" designation that represents all states in which the taxi and the passengers are not at one of the four locations. For Nav(R), the reward function returns 1 when the taxi is at location Red, and 0 otherwise. Nav(R) terminates when the taxi is at the location Red.

At level 2, the AMDP actions are Get, which puts the passenger into the taxi; and Put, which drops the passenger at their destination. The states abstract away the taxi and passenger locations. Hence, the subtask hierarchy of MAXQ is similar to AMDPs; however, the task decomposition is local in its reward functions, transition function, and value function or planning computations, resulting in substantial savings in planning time.

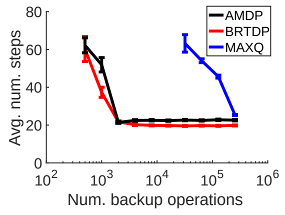
Cleanup World

Our second evaluation domain is Cleanup World [98], a mobile-manipulation task, as shown in Figure 3.4a. Cleanup World is a good testbed for planning algorithms, because its state space grows combinatorially with the number of objects and rooms, similar to real-world robotics planning problems. The robot can have a variety of goals, such as moving the chair to the red room or moving all objects to the blue room. Abstract actions include moving to a door connected to the room in which the robot


(a) Taxi problem



(b) Percentage of completed plans (of 1000 problems), given a backup budget.

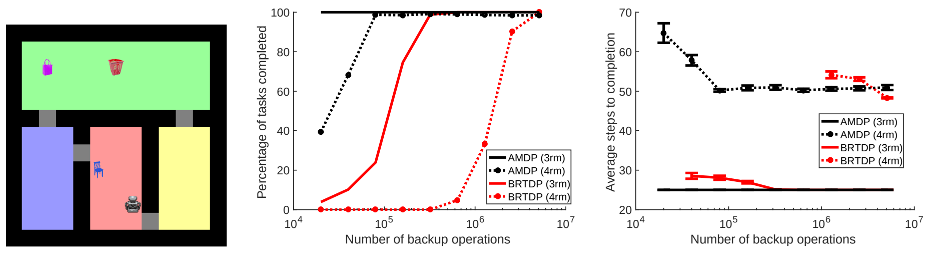


(c) Average number of steps needed to deliver the passenger in completed plans.

Figure 3.3: Taxi domain using AMDPs, base-level planning (BRTDP), and MAXQ. AMDPs and the base-level planner have almost 100% completion rate after 4000 backup operations, whereas MAXQ needs orders of magnitude more backups to plan.

currently resides, moving from a door to a connected room, moving to an object currently in the same room (or doorway) as the robot, taking an object next to the robot to a door, and taking an object from a door to a connected room. The source MDP goal conditions for each of these AMDP actions can be defined based on the action arguments. Using the MoveToDoor(d) AMDP as an example, its reward function returns 1 when the agent is at the specified doorway d, and 0 otherwise. MoveToDoor(d) terminates at any state in which the robot is in the doorway.

The abstract actions define a corresponding state representation that abstracts away the geometric spatial information from the source OO-MDP. Such a representation retains the same objects as the source OO-MDP, but represents objects' positions and the room–door topology relationally instead of spatially. Specifically, the robot and household objects have an attribute that points to the doorway/room in which they reside; the robot has an attribute that points to adjacent household objects; and the room points to connected doorways (and vice versa). A second-level AMDP involves even higher-level actions, such as taking any given object to any given room. The corresponding high-level state space for this AMDP abstracts away the room-door topology.



(a) Cleanup World (4rm)

(b) Tasks completed within a planning budget.

(c) Steps taken to move an object to a room.

Figure 3.4: (a) Cleanup World (4-room configuration shown) using AMDPs and base-level (BRTDP) planning. (b-c) AMDPs (black) have near-perfect completion rates, finding plans of similar length orders of magnitude faster than BRTDP (red).

3.3 Results

We compared AMDP planning performance against a flat planner and MAXQ in Fickle Taxi and Cleanup World. For each method, we generated plans using bounded RTDP [106], a state-of-the-art method with performance guarantees. We used MAXQ with state abstraction, that is, MAXQ-Q by Dieterrich [33]. We compared approaches via two metrics: (1) Bellman updates/backups needed for effective planning and (2) steps taken to solve the task given a budget of Bellman updates. We plotted the average steps to completion only if the task was solved in more than 5% of 1000 trials. We recognize that MAXQ is learning a Q-function model-free; however, it is the closest comparison to AMDPs in terms of state and temporal abstraction. We also implemented a comparison to options, but found, consistent with Jong et al. [66], that they performed less well than the base-level planner. For more details please refer to Gopalan et al. [46]. We cannot use other top-down approaches, because HDP does not generalize to non-navigation based domains and DetH* does not allow multi-level hierarchies or stochastic transitions at higher levels of abstraction, which are needed for Taxi. We further used the AMDP hierarchy for Cleanup World to control a Turtlebot in a continuous space with planning over low-level control actions in the lowest level of the hierarchy.

3.3.1 Taxi Domain

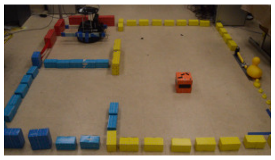
In Taxi, we defined a task to be completed if a planner found a solution that executed ≤ 100 primitive actions. For MAXQ, we used the same state abstractions and tuned parameters given by Dieterrich [33]. MAXQ required about $32{,}000$ updates/backups to first becoming capable of completing the task, and about $256{,}000$ backups to learn an optimal policy. By contrast, as shown in Figure 3.3b, AMDPs exhibited a 100% completion rate at 8000 backups, slightly faster than the base-level planner. The number of steps to complete the task with AMDPs (22.5) was higher than the base-level planner (19.8) (see Figure 3.3c), because the hierarchical action of navigate is penalized by the passenger's fickle behavior. The base-level planner outperforms AMDPs in this domain because the taxi task is relatively simple and plans are very short. However, this will no longer be the case when the domain's state space is combinatorially large.

3.3.2 Cleanup World

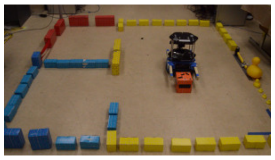
We constructed two different configurations of the Cleanup World domain. The first configuration (3rm) had 3 rooms with 3 objects and about 56 million states. The optimal plan was 25 steps. The second configuration (4rm, shown in Figure 3.4a) had 4 rooms with 3 objects and more than 900 million states. The optimal plan was 48 steps long. We defined the task to be completed if the agent is able to solve it within 5 times the number of steps to solve the task optimally.

In both configurations, AMDPs solved the task with an order of magnitude fewer backups than the base-level BRTDP, as shown in Figure 3.4. AMDPs solved 3rm and 4rm to almost 100% completion rate with about 5000 and $80{,}000$ backups, respectively, whereas BRTDP needed $320{,}000$ and $512{,}000$ backups, respectively, to achieve the same performance.

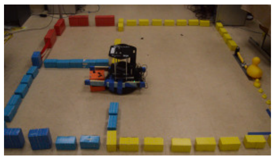
We also solved a basic Cleanup World problem with 3 rooms and one object ($25{,}000$ states), in which both AMDPs and the base-level planners could find the optimal solu-



(a) Starting position for the turtlebot.



(b) Moving to capture the block.



(c) Pushing the block into the goal room.

Figure 3.5: In this figure, we see that the robot starts from a random position and pushes the block into the goal room, with blue walls, using its arms. The robot makes minor corrections in trajectories when it overshoots.

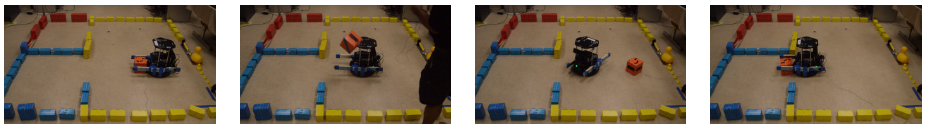
tion within 500 backups, showing that the task was simple enough not to need complex hierarchies. However, we found that, even in this simplified configuration, MAXQ was unable to find a solution with $8{,}000{,}000$ backups.

AMDPs have a particular advantage in Cleanup World because long plans are needed, and solving tasks requires manipulating one out of many present objects. Such combinatorial state spaces are present in everyday manipulation tasks, and AMDPs are better equipped than other hierarchical methods or base-level planning to tackle them.

3.3.3 Continuous Cleanup World on Turtlebot

To illustrate the ability of AMDP hierarchies to transfer between domains and use different planners at different levels, we constructed a continuous version of a 3-room, 1-object Cleanup World domain, which we tested on a Turtlebot. The agent in Continuous Cleanup World is a modified TurtleBot with a pair of appendages that can guide blocks. The appendages make pushing the block easy; however, they make movement within the domain hard, as a point turn is not possible in most places, increasing the overall plan lengths.

At level 0, the agent has a continuous position in Cartesian coordinates, in a $9\text{ft}\times 9\text{ft}$ world. The agent also has a continuous angle attribute for its orientation. The Turtlebot can move forward and turn clockwise/counter-clockwise. The continuous forward action moves the robot forward by about 0.1ft. The turn actions change the orientation of



(a) Trying to push the block to the goal. (b) Losing the block accidentally. (c) Adjusting trajectory. (d) Getting to the goal.

Figure 3.6: This figure shows the importance of having MDPs to control the lowest level of the hierarchy. The Turtlebot starts to push the block into the goal location, but has the block removed from its arms, and replaced at another location. The Turtlebot corrects its trajectory immediately, getting to the block and pushing it into the goal.

the agent about 0.15 radians. The object also has a continuous position attribute. The objective again is to move the object to a goal room, from any location in the world.

Our abstraction of the continuous source MDP connects to the (discrete) Cleanup World AMDP hierarchy from the previous subsection. At level 1, we discretize the world into 9×9 cells and the agent's bearing into the 4 cardinal directions. The subtasks at this level are moving forward by one grid cell, and turning clockwise/counter-clockwise by $\frac{\pi}{2}$ radians. This level-1 AMDP corresponds to the source MDP of (discrete) Cleanup World, so we immediately obtain higher-level abstractions for Continuous Cleanup World as well. Additionally, the top-down nature of AMDP planning enables the retention of all higher-level models and previously computed policies. By contrast, bottom-up approaches such as MAXQ would have needed to re-solve all subtasks.

In addition to the above level-1 abstraction, we also need to specify how to solve level-1 subtasks. Since the source MDP is continuous, low-level planners such as Rapidly Exploring Randomized Trees [89] could have been used to plan short distances of single grid movement. However, we use closed-loop controllers, because they allow faster planning and are less complicated to specify for these small movements. This domain application shows how different planners can be used at different levels of the AMDP hierarchy, depending on their suitability.

The Turtlebot is controlled using movement messages sent by our planner over

ROS [126] at 20Hz. Our planners can publish commands at over 100Hz, but 20Hz is the standard rate at which the Turtlebot publishes commands to its mobile base. The robot can be moved faster by publishing higher velocities per motion command, but we do not want the Turtlebot to move too quickly within the lab. The location and pose of the Turtlebot and the object are obtained using a motion capture system.

Put together, our AMDP hierarchy solves this continuous problem in real time. We can see in the video[2] that the robot plans in real time to complete the manipulation task as shown in Figure 3.5. The robot makes minor corrections to its trajectory and recovers from mistakes in real time. It also recovers when the object is removed from its arms, and replans to recover the object instantaneously to push the object to the goal location as shown in Figure 3.6. This shows that fast reactive control is possible in top-down hierarchies such as AMDPs, even in domains with significant stochasticity. Furthermore, the hierarchy allows us to control the robot at different levels of abstraction, since we can execute any subgoal by only planning with its subtree in the task hierarchy.

3.4 Conclusion

In this chapter we introduced a novel planning approach using Abstract Markov Decision Process (AMDP) hierarchies, which decompose large planning problems into AMDPs, representing subtasks that have local transition and reward functions. AMDPs compute plans for large problems orders of magnitude faster than other planners. Options and MAXQ may fail to plan in large domains because of the time spent on recursively decomposing the value functions from the base level. Hence, even though MAXQ and options provide strong theoretical guarantees of being recursively and hierarchically optimal, respectively, their planning time might be too long for actual agents to act in the world. AMDPs, on the other hand, offer a weaker notion of optimality

[2]https://youtu.be/Bp3VEO66WSg

at every abstract level, but allow faster planning in large domains. Previous top-down planning approaches such as HDP and DetH* are not generic enough to allow mobile manipulation in stochastic continuous domains. Moreover, the AMDP hierarchical structure allows AMDPs to be invariant to small changes in stochasticity at the flat level. This property has useful applications in robotics, when accurate models for the base-level transitions are not available, allowing the robot to recover reactively from missteps or environmental noise.

We demonstrated with the Taxi and Cleanup World domains that AMDPs trade orders-of-magnitude faster planning time for potentially suboptimal solutions. Additionally, in the Turtlebot Continuous Cleanup World, our AMDP hierarchy provides a model for a robot's entire capability stack. This unified model allows tasks, specified as high-level abstract goals, to be efficiently planned for and grounded into low-level control actions. This planning speedup is crucial for planning in large domains such as robotics, navigation, and search and rescue.

There has been significant follow up work performed to learn AMDPs. Winder et al. [151] attempted to learn the transition dynamics of individual AMDPs within a given hierarchy. Roderick et al. [130] attempted to learn model-free policies within each AMDP, while learning abstract actions given a state hierarchy. The state hierarchies were pre-specified using the ROM data of Atari Games. In Chapters 4, 5 and 6 AMDPs are used for the underlying planning to satisfy the high level behavior specified using natural language. In Chapter 7 we learn subgoal conditions that a planner can satisfy. We hope that such a method would be used with language to learn AMDP hierarchies in the future.

Chapter 4

Interpreting Human-Robot Commands at Multiple Levels of Abstraction via Classification

This chapter in the dissertation has been previously presented at RSS 2017 as "Accurately and Efficiently Interpreting Human-Robot Instructions of Varying Granularities," with Dilip Arumugam, Siddharth Karamcheti, Lawson L. S. Wong and Stefanie Tellex [8]. The core idea to use language with abstractions came from Stefanie to show capabilities of the AMDP hierarchies from the previous chapter. The importance of this chapter in my dissertation, and my contribution, is the timing analysis for AMDP hierarchies when connected to a language grounding pipeline. The timing analysis establishes the importance of hierarchies for real time behavior on a robot, and led my interest in combining language and hierarchies in other parts of my thesis. The language grounding work here *was done by Dilip and Siddharth.*

In everyday speech, humans use language at multiple levels of abstraction. For example, a brief transcript from an expert human forklift operator instructing a human trainee has very abstract commands such as "Grab a pallet," mid-level commands such as "Make sure your forks are centered," and very fine-grained commands such as "Tilt back a little bit" all within thirty seconds of dialog. Humans use these varied granularities to specify and reason about a large variety of tasks with a wide range of difficulties. Furthermore, these abstractions in language map to subgoals that are useful when interpreting and executing a task. In the case of the forklift trainee above, the sub-goals of moving to the pallet, placing the forks under the object, then lifting it up are all implicitly encoded in the command "Grab a pallet." In this chapter we decompose generic, abstract natural language commands into an AMDP hierarchy's subgoals, which allows human users to exert more specificity with planning efficiency and control in the robot's planning and execution of tasks.

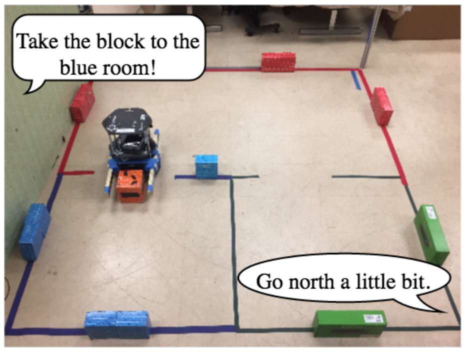


Figure 4.1: Examples of high-level and fine-grained commands issued to the Turtlebot robot in a mobile-manipulation task.

Existing approaches map between natural language commands and a formal representation at some fixed level of abstraction [26, 104, 140]. While effective at directing robots to complete predefined tasks, mapping to fixed sequences of robot actions is unreliable in changing or stochastic environments. Accordingly, MacGlashan et al. [98] decouple the problem and use a statistical language model to map between language and robot goals, expressed as reward functions in an MDP. Then, an arbitrary planner solves the MDP, resolving any environment-specific challenges with execution. As a result, the learned language model can transfer to other robots with different action sets so long as there is consistency in the task representation (*i.e.*, reward functions). However, MDPs for complex, real-world environments face an inherent tradeoff between including low-level task representations and increasing the time needed to plan in the presence of both low- and high-level reward functions [47].

To address these problems, we present an approach for mapping natural language commands of varying complexities to reward functions at different levels of abstraction within a hierarchical planning framework. This approach enables the system to quickly and accurately interpret both abstract and fine-grained commands. Our system uses a deep neural network language model that classifies natural language commands to the appropriate level and reward function of an AMDP planning hierarchy. By coupling abstraction-level inference with the overall grounding problem, we exploit the subsequent hierarchical planner to efficiently execute the grounded tasks. To our knowledge, we are the first to contribute a system for grounding language at multiple levels of abstraction.

Our evaluation shows that classification based deep neural network language models can infer reward functions more accurately than statistical language model baselines. We present results comparing a traditional statistical language model to three different neural architectures that are commonly used in natural language processing. Furthermore, we show that a hierarchical approach allows the planner to map to a larger, richer space of reward functions more quickly and more accurately than non-hierarchical baselines. This speedup allows the robot to respond faster and more accurately to a user's request, with a much larger set of potential commands than previous approaches. We also demonstrate the rapid and accurate response of our system to natural language commands at varying levels of abstraction on a Turtlebot.

4.1 Related work

Humans use natural language to communicate ideas, motivations, task descriptions, etc. with other humans. Some of the earliest work in this area mapped tasks to another planning language, which then grounded to the actions performed by the robots [38, 26]. More recent methods ground natural language commands to tasks using features that describe correspondences between natural language phrases present in the task description to the physical objects [57, 104, 140, 20, 127], or abstract spatial concepts [123] present in the world and the actions available in the world. This featurized representation can then describe the sequence of actions needed to complete the task. All these approaches ground commands to action sequences, leading to brittle behavior if the environment is stochastic.

MacGlashan et al. [98] proposed grounding natural language commands to reward functions associated with certain tasks, allowing robot agents to plan in stochastic environments. They treat the goal reward function as a sequence of propositional functions, much like a machine language, to which a natural language task can be translated, using an IBM Model 2 [21, 22] (IBM2) language model. While their propositional functions only lie at one level of abstraction, we want the robot to understand commands at different levels of specificity while still maintaining efficient planning and execution in the face of multiple levels of abstraction.

Crucially, MacGlashan et al. [98] actually perform inference over reward function templates, or lifted reward functions, along with environmental constraints. A lifted reward function merely specifies a task while leaving the environment-specific variables of the task undefined. The environmental binding constraints then specify the properties that an object in the environment must satisfy in order to be bound to a lifted reward function variable. By doing this, the output space of the language model is never tied to any particular instantiation of the environment, but can instead align to objects and attributes that lie within some distribution over environments. Given a lifted reward function and environment constraints (henceforth jointly referred to as only a lifted reward function), a subsequent model can later infer the environment-specific variables without needing to relearn the language understanding components for each environment. In order to leverage this flexibility, all of our proposed language models produce lifted reward functions which are then completed by a grounding module before being passed to the planner (see Sec. 4.3). Specifically we use an AMDP planner, introduced in the previous chapter, to allow fast hierarchical planning in large combinatorial domains. Moreover the reward functions at different levels of an AMDP hierarchy allows specifying behavior at different levels of granularity.

We use a deep neural network language model to perform language grounding. Our approach uses both word embeddings and a state-of-the-art RNN model to map between natural language and MDP reward functions.

4.2 Technical Approach

To interpret a variety of natural language commands, there must be a representation for all possible tasks and subtasks. We specify an OO-MDP [36] to model the robot's environment and actions.

An OO-MDP builds upon an MDP by adding sets of object classes and propositional functions; each object class is defined by a set of attributes and each propositional function is parameterized by instances of object classes. For example, an OO-MDP for the mobile robot manipulation domain seen in Figure 4.1 might denote the robot's successful placement of the orange block into the blue room via the propositional function blockInRoom block0 room1, where block0 and room1 are instances of the block and room object classes respectively and the blockInRoom propositional function checks if the location attribute of block0 is contained in room1. Using these propositional functions as reward functions that encode termination conditions for each task, we arrive at a sufficient, semantic representation for grounding language. For our evaluation, we use the Cleanup World [67, 98] OO-MDP, which models a mobile manipulator robot; this domain is defined in Sec. 5.3.1.

However, this approach does not generalize well to different environment configurations. At training time, any natural language command that moves objects or agents to a specific room is conditioned to map room attributes to specific room instances (i.e. in the case of Figure 4.1, the blue room is always room1). With this in mind, consider what happens if we switched the blue and green rooms at test time, so that the green room is now room1. In this case, any language command that moves an object or agent to the blue room would fail, as the room instances have been switched around.

To this end, we "lift" the propositional functions from before, to better generalize to unseen environments. Given a command like "Take the block to blue room," the corresponding lifted propositional function takes the form blockInRoom block0 roomIs-Blue, denoting that the block should end up in the room that is blue. We then assume an environment-specific grounding module (see Sec. 4.3.3) that consumes these lifted reward functions and performs the actual low-level binding to specific room instances, which can then be passed to a planner.

In order to effectively ground commands across multiple levels of complexity, we assume a predefined AMDP hierarchy over the state-action space of the given grounding environment. Furthermore, each level of this hierarchy requires its own set of reward functions for all relevant tasks and sub-tasks. Finally, we assume that the entire command at runtime is generated from a single, fixed level of abstraction.

Given a natural language command c, we find the corresponding level of the abstraction hierarchy l, and the lifted reward function m that maximizes the joint probability of l, m given c. Concretely, we seek the level of the state-action hierarchy $\hat{l}$ and the lifted reward function $\hat{m}$ such that:

$$\hat{l}, \hat{m} = \arg\max_{l,m} Pr(l, m \mid c), \tag{4.1}$$

For example, as illustrated in Figure 4.1, a high-level natural language command like "Take the block to the blue room" would map to the highest abstraction level, while a low-level command like "Go north a little bit" would map to the finest-grained level. We estimate this joint probability by learning a language model (described in Sec. 4.3) and training on a parallel corpus that pairs natural language commands with a corresponding reward function at a particular level of the abstraction hierarchy.

Given this parallel corpus, we train each model by directly maximizing the joint probability from Eqn. 4.1. Specifically, we learn parameters $\hat{\theta}$ that maximize the corpus likelihood:

$$\hat{\theta} = \arg\max_{\theta} \prod_{(c,l,m)\in\mathbb{C}} Pr(l, m \mid c, \theta). \tag{4.2}$$

At inference time, given a language command c, we find the best l, m that maximize the probability $Pr(l, m \mid c, \hat{\theta})$. The lifted reward function m is then completed by the grounding module (see Sec. 4.3.3) and passed to a hierarchical planner, which plans the corresponding task at abstraction level l.

4.3 Language Models

We compare four language models: an IBM Model 2 translation model (similar to MacGlashan et al. [98]), a deep neural network bag-of-words language model, and two recurrent neural network (RNN) language models, with varying architectures. For detailed descriptions and implementations of all the presented models, as well as the datasets used throughout this chapter, please refer to the supplemental repository[1].

[1]https://github.com/h2r/GLAMDP

4.3.1 IBM Model 2

As a baseline, task grounding is formulated as a machine translation problem, with natural language as the source language and semantic task representations (lifted reward functions) as the target language. We use the well-known IBM Model 2 (IBM2) machine translation model [21, 22] as a statistical language model for scoring reward functions given input commands. IBM2 is a generative model that solves the following objective (equivalent to Eqn. 4.1 by Bayes' rule):

$$\hat{l}, \hat{m} = \arg\max_{l,m} Pr(l, m) \cdot Pr(c \mid l, m). \tag{4.3}$$

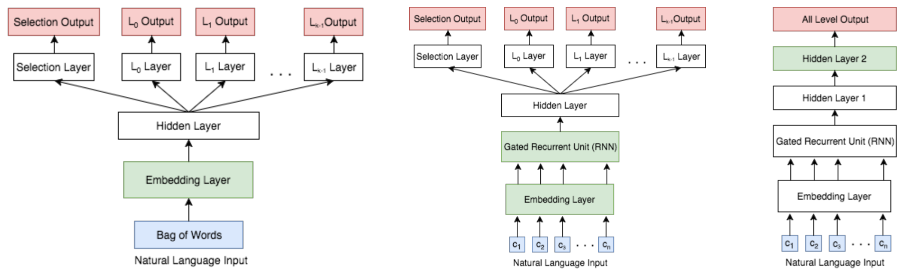
This task grounding formulation follows directly from MacGlashan et al. [98] and we continue in an identical fashion training the IBM2 using the standard EM algorithm.

4.3.2 Neural Network Language Models

We develop three classes of neural network architectures (see Figure 4.2): a feed-forward network that takes a natural language command encoded as a bag-of-words and has separate parameters for each level of abstraction (**Multi-NN**), a recurrent network that takes into account the order of words in the sequence, also with separate parameters (**Multi-RNN**), and a recurrent network that takes into account the order of words in the sequence and has a shared parameter space across levels of abstraction (**Single-RNN**).

Multi-NN: Multiple Output Feed-Forward Network

We propose a feed-forward neural network [15, 62, 110] that takes in a natural language command c as a bag-of-words vector $\vec{c}$, and outputs both the probability of each of the different levels of abstraction, as well as the probability of each reward function. We decompose the conditional probability from Eqn. 4.1 as $Pr(l, m \mid c) = Pr(l \mid c) \cdot Pr(m \mid l, c)$. Applying this to the corpus likelihood (Eqn. 4.2) and taking logarithms,



(a) Multi-NN Model (b) Multi-RNN Model (c) Single-RNN Model

Figure 4.2: Model architectures for all three sets of deep neural network models. In blue are the network inputs, and in red are the network outputs. Going left to right, the green denotes significant structural differences between models.

the Multi-NN objective is to find parameters $\hat{\theta}$:

$$\hat{\theta} = \arg\max_{\theta} \sum_{(\vec{c},l,m)} \log Pr(l \mid \vec{c}, \theta) + \log Pr(m \mid l, \vec{c}, \theta), \tag{4.4}$$

To learn this set of parameters, we use the architecture shown in Figure 4.2a. Namely, we employ a multi-output deep neural network with an initial embedding layer, a hidden layer that is shared between each of the different outputs, and then output-specific hidden and read-out layers, respectively.

The level selection output is a k-element discrete distribution, where k is the number of levels of abstraction in the given planning hierarchy. Similarly, the reward function output at each level L_i is an r_i-element distribution, where r_i is the number of reward functions at the given level of the hierarchy.

To train the model, we minimize the sum of the cross-entropy loss on each term in Eqn. 4.4. We train the network via backpropagation, using the Adam Optimizer [72], with a mini-batch size of 16, and a learning rate of 0.001. Furthermore, to better regularize the model and encourage robustness, we use Dropout [134] after the initial embedding layer, as well as after the output-specific hidden layers with probability $p = 0.5$.

Multi-RNN: Multiple Output Recurrent Network

Inspired by the success of recurrent neural networks (RNNs) in NLP tasks [27, 108, 109, 136], we propose an RNN language model that takes in a command as a sequence of words and, like the Multi-NN bag-of-words model, outputs both the probability of each of the different levels of abstraction, as well as the probability of each reward function, at each level of abstraction. RNNs extend feed-forward networks to handle variable length inputs by employing a set of one or more hidden states, which are updated after reading in each input token. Instead of converting natural language command c to a vector $\vec{c}$, we use an RNN to interpret it as a sequence of words $s = \langle c_1, c_2 \dots c_n \rangle$. The Multi-RNN objective is then:

$$\hat{\theta} = \arg\max_{\theta} \sum_{(c,l,m)} \log Pr(l \mid s, \theta) + \log Pr(m \mid l, s, \theta) \tag{4.5}$$

This modification is reflected in Figure 4.2b, which is similar to the Multi-NN architecture, except in the lower layers where we use an RNN encoder that takes the sequence of raw input tokens and maps them into a fixed-size state vector. We use the gated recurrent unit (GRU) of Cho et al. [27], a particular type of RNN cell that have been shown to work well on natural language sequence modeling tasks [29].

Similar to the Multi-NN, we train the model by minimizing the sum of the cross-entropy loss of each of the two terms in Eqn. 4.5, with the same optimizer setup as the Multi-NN model. Dropout is used to regularize the network after the initial embedding layer and the output-specific hidden layers.

Single-RNN: Single Output Recurrent Network

Both Multi-NN and Multi-RNN decompose the conditional probability of both the level of abstraction l and the lifted reward function m given the natural language command c as $Pr(l, m \mid c) = Pr(l \mid c) \cdot Pr(m \mid l, c)$, allowing for the explicit calculation of the probability of each level of abstraction given the natural language command. As a result, both Multi-NN and Multi-RNN create separate sets of parameters for each of the separate outputs, *i.e.* separate parameters for each level of abstraction in the underlying hierarchical planner.

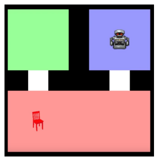
Alternatively, we can directly estimate the joint probability $Pr(l, m \mid c)$. To do so, we propose a different type of RNN model that takes in a natural language command as a sequence of words s (as in Multi-RNN), and directly outputs the joint probability of each tuple (l, m), where l denotes the level of abstraction, and m denotes the lifted reward function at the given level. The Single-RNN objective is to find $\hat{\theta}$ such that:

$$\hat{\theta} = \arg\max_{\theta} \sum_{(n,l,m)} \log Pr(l, m \mid s, \theta) \tag{4.6}$$

With this Single-RNN model, we are able to significantly improve model efficiency compared to the Multi-RNN model, as all levels of abstraction share a single set of parameters. Furthermore, removing the explicit calculation of the level selection probabilities allows for the possibility of positive information transfer between levels of abstraction, which is not necessarily possible with the previous models.

The Single-RNN architecture is shown in Figure 4.2c. We use a single-output RNN, similar to the Multi-RNN architecture, with the key difference being that there is only a *single* output, with each element of the final output vector corresponding to the probability of each tuple of levels of abstraction and reward functions (l, m) given the natural language command c.

To train the model, we minimize the cross-entropy loss of the joint probability term in Eqn. 4.6. Training hyperparameters are identical to Multi-RNN, and dropout is applied to the initial embedding layer and the penultimate hidden layer.



(a) A starting instance of the Cleanup World domain.

Level	Example Command	Reward Function
L_0	Turn and move one spot to the right.	goWest
	Go three down, four over, two up.	agentInRoom agent0 roomIsGreen
L_1	Go to door, enter red room, push chair to green room door.	blockInRegion block0 roomIsGreen
	Go to the door then go into the red room.	agentInRegion agent0 roomIsRed
L_2	Go to the green room.	agentInRegion agent0 roomIsGreen
	Bring the chair to the blue room.	blockInRegion block0 roomIsBlue

(b) Example commands and corresponding reward functions.

Figure 4.3: Amazon Mechanical Turk (AMT) dataset domain and examples.

4.3.3 Grounding Module

In all of our models, the inferred lifted reward function template must be bound to environment-specific variables. The grounding module maps the lifted reward function to a grounded one that can be passed to an MDP planner. In our evaluation domain (see Figure 4.1), it is sufficient for our grounding module to be a lookup table that maps specific environment constraints to object ID tokens. In domains with ambiguous constraints (*e.g.* a "chair" argument where multiple chairs exist), a more complex grounding module could be substituted. For instance, Artzi and Zettlemoyer [6] present a model for executing lambda-calculus expressions generated by a combinatory categorial grammar (CCG) semantic parser, which grounds ambiguous predicates and nested arguments.

4.4 Evaluation

Our evaluation tests the hypothesis that hierarchical structure improves the speed and accuracy of language grounding at multiple levels of abstraction. We measure grounding accuracy and planning speed in simulation with a corpus-based evaluation, and demonstrate our system on a Turtlebot robot.

4.4.1 Mobile-Manipulation Robot Domain

The Cleanup World domain [67, 98], illustrated in Figure 4.3a, is a mobile-manipulator robot domain that is partitioned into rooms (denoted by unique colors) with open doors. Each room may contain some number of objects which can be moved (pushed) by the robot. This problem is modeled after a mobile robot that moves objects around, analogous to a robotic forklift operating in a warehouse or a pick-and-place robot in a home environment. We use an AMDP for the Cleanup World domain as defined in the previous chapter, which imposes a three-level abstraction hierarchy for planning.

The combinatorially large state space of Cleanup World simulates real-world complexity and is ideal for exploiting abstractions. At the lowest level of abstraction L_0, the (primitive) action set available to the robot agent consists of north, south, east, and west actions. Users directing the robot at this level of granularity must specify lengthy step-by-step instructions for the robot to execute. At the next level of abstraction L_1, the state space of Cleanup World only consists of rooms and doors. The robot's position is solely defined by the region (*i.e.* room or door) it resides in. Abstracted actions are *subroutines* for moving either the robot or a specific block to a room or door. It is impossible to transition between rooms without first transitioning through a door, and it is only possible to transition between adjacent regions; any language guiding the robot at L_1 must adhere to these dynamics. Finally, the highest level of abstraction, L_2, removes the concept of doors, leaving only rooms as regions; all L_1 transition dynamics still hold, including adjacency constraints. Subroutines exist for moving either the robot or a block between connected rooms. The full space of subroutines at all levels and their corresponding propositional functions are defined by [47]. Figure 4.3b shows a few collected sample commands at each level and the corresponding level-specific AMDP reward function.

4.4.2 Procedure

We conducted an Amazon Mechanical Turk (AMT) user study to collect natural language samples at various levels of abstraction in Cleanup World. Annotators were shown video demonstrations of ten tasks, always starting from the state shown in Figure 4.3a. For each task, users provided a command that they would give to a robot, to perform the action they saw in the video, while constraining their language to adhere to one of three possible levels in a designated abstraction hierarchy: fine-grained, medium, and coarse. This data provided multiple parallel corpora for the machine translation problem of task grounding. We measured our system's performance by passing each command to the language grounding system and assessing whether it inferred both the correct level of abstraction and the reward function. We also recorded the response time of the system, measuring from when the command was issued to the language model to when the (simulated) robot would have started moving. Accuracy values were computed using the mean of multiple trials of ten-fold cross validation. The space of possible tasks included moving a single step as well as navigating to a particular room, taking a particular object to a designated room, and all combinations thereof.

Unlike MacGlashan et al. [98], the demonstrations shown were not only limited to simple robot navigation and object placement tasks, but also included composite tasks (*e.g.* "Go to the red room, take the red chair to the green room, go back to the red room, and return to the blue room"). Commands reflecting a clear misunderstanding of the presented task, *e.g.* "please robot", were removed from the dataset. Such removals were rare; we removed fewer than 30 commands for this reason, giving a total of 3047 commands. Per level, there were 1309 L_0 commands, 872 L_1 commands, and 866 L_2 commands. The L_0 corpus included more commands since the tasks of moving the robot one unit in each of the four cardinal directions do not translate to higher levels of abstraction.

	Evaluated L_0	Evaluated L_1	Evaluated L_2
Trained L_0	**21.61%**	**17.20%**	21.87%
Trained L_1	9.83%	10.23%	13.90%
Trained L_2	14.94%	12.84%	**31.49%**

(a) IBM2 Reward Grounding Baselines

	Evaluated L_0	Evaluated L_1	Evaluated L_2
Trained L_0	**77.67%**	28.05%	23.26%
Trained L_1	32.79%	**82.99%**	74.65%
Trained L_2	14.19%	58.62%	**87.91%**

(b) Single-RNN Reward Grounding Baselines

Figure 4.4: Task grounding accuracy (averaged over 5 trials) when training IBM2 and Single-RNN models on a single level of abstraction, then evaluating commands from alternate levels. This is similar to the MacGlashan et al. [98] results, as we see that without accounting for abstractions in language, there is a noticeable effect on grounding accuracy.

4.4.3 Robot Task Grounding

We present the baseline task grounding accuracies in Figure 4.4 to demonstrate the importance of inferring the latent abstraction level in language. We simulate the effect of an oracle that partitions all of the collected AMT commands into separate corpora according to the specificity of each command. For this experiment, any L_0 commands that did not exist at all levels of the Cleanup World hierarchy were omitted, resulting in a condensed L_0 dataset of 869 commands. We trained multiple IBM2 and Single-RNN models using data from one distinct level and then evaluated using data from a separate level. Training a model at a particular level of abstraction includes grounding solely to the reward functions that exist at that same level. Reward functions at the evaluation level were mapped to the equivalent reward functions at the training level (*e.g.* L_1 agentInRegion to L_0 agentInRoom). Entries along the diagonal represent the average task grounding accuracy for multiple, random 90-10 splits of the data at the given level. Otherwise, evaluation checked for the correct grounding of the command to a reward function at the training level equivalent to the true reward function at the alternate evaluation level.

Task grounding scores are uniformly quite poor for IBM2; however, IBM2 models trained using L_0 and L_2 data respectively result in models that substantially outperform those trained on alternate levels of data. It is also apparent that an IBM2 model trained on L_1 data fails to identify the features of the level. We conjecture that this is

	Level Selection	Reward Grounding
IBM2	79.87%	27.26%
Multi-NN	93.51%	36.05%
Multi-RNN	95.71%	80.11%
Single-RNN	**95.91**%	**80.46**%

Figure 4.5: Accuracy of 10-Fold Cross Validation (averaged over 3 runs) for each of the models on the AMT Dataset.

caused, in part, by high variance among the language commands collected at L_1 as well as the large number of overlapping, repetitive tokens that are needed for generating a valid machine language instance at L_1. While these models are worse than what MacGlashan et al. [98] observed, we note that we do not utilize a task or behavior model. It follows that integrating one or both of these components would only help prune the task grounding space of highly improbable tasks and improve our performance.

Conversely, Single-RNN shows the expected maximization along diagonal entries that comes from training and evaluating on data at the same level of abstraction. These show that a model limited to a single level of language abstraction is not flexible enough to deal with the full scope of possible commands. Additionally, Single-RNN demonstrates more robust task grounding than statistical machine translation.

The task grounding and level inference scores for the models in Sec. 4.3 are shown in Figure 4.5. Attempting to embed the latent abstraction level within the machine language of IBM2 results in weak level inference. Furthermore, grounding accuracy falls even further due to sparse alignments and the sharing of tokens between tasks in machine language (*e.g.* agentInRoom agent0 room1 at L_0 and agentInRegion agent0 room1 at L_1). The fastest of all the neural models, and the one with the fewest number of parameters overall, Multi-NN shows notable improvement in level inference over the IBM2; however, task grounding performance still suffers, as the bag-of-words representation fails to capture the sequential word dependencies critical to the intent of each command. Multi-RNN again improves upon level prediction accuracy and leverages the high-dimensional representation learned by initial RNN layer to train reliable grounding models specific to each level of abstraction. Finally, Single-RNN has near-perfect level prediction and demonstrates the successful learning of abstraction level as a latent feature within the neural model. By not using an oracle for level inference, there is a slight loss in performance compared to the results obtained in Figure 4.4b; however, we still see improved grounding performance over Multi-RNN that can be attributed to the full sharing of parameters across all training samples allowing for positive information transfer between abstraction levels.

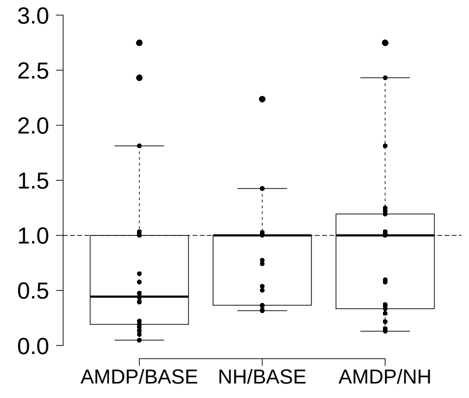
4.4.4 Robot Response Time

Fast response times are important for fluid human-robot interaction, so we assessed the time it would take a robot to respond to natural language commands in our corpus. We measured the time it takes for the system to process a natural language command, map it to a reward function, and then solve the resulting MDP to yield a policy so that the simulated robot would start moving. We used Single-RNN for inference since it was the most accurate grounding model, and only correctly grounded instances were evaluated, so our results are for 2634 of 3047 commands that Single-RNN got correct.

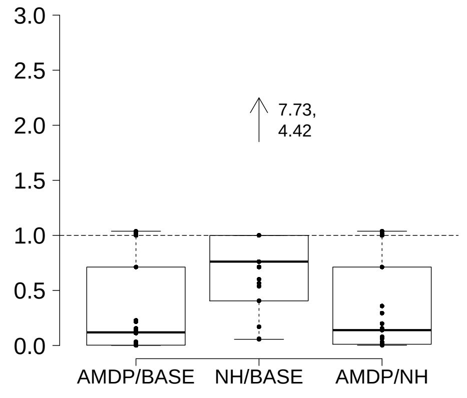
We compared three different planners to solve the MDP:

- **BASE**: A state-of-the-art flat (non-hierarchical) planner, bounded real-time dynamic programming (BRTDP [**?**]).

- **AMDP**: A hierarchical planner for MDPs [47]. At the primitive level of the hierarchy (L_0), **AMDP** also requires a flat planner; we use **BASE** to allow for comparable planning times. Because the subtasks have no compositional structure, a Manhattan-distance heuristic can be used at L_0. While **BASE** technically allows for heuristics, distance-based heuristics are unsuitable for the composite tasks in our dataset. This illustrates another benefit of using hierarchies: to decompose composite tasks into subtasks that are amenable to better heuristics.



(a) Regular domain (2^{14} states)



(b) Large domain (2^{18} states)

Figure 4.6: Relative inference + planning times for different planning approaches on the same correctly grounded AMT commands. For each method pair, values less than 1 indicate the method on the numerator (left of '/') is better. Each data point is an average of 1000 planning trials.

- **NH** (No Heuristic): Identical to **AMDP**, but without the heuristic as a fair comparison against **BASE**.

We hypothesize **NH** is faster than **BASE** (due to use of hierarchy), but not as fast as **AMDP** (due to lack of heuristics).

Since the actual planning times depend heavily on the actual task being grounded (ranging from 5ms for goNorth to 180s for some high-level commands), we instead evaluate the *relative* times used between different planning approaches. Figure 4.6a shows the results for all 3 pairs of planners. For example, the left-most column shows $\frac{\textbf{AMDP}\text{ time}}{\textbf{BASE}\text{ time}}$; the fact that most results were less than 1 indicates that **AMDP** usually outperforms **BASE**. Using Wilcoxon signed-rank tests, we find that each approach in the numerator is significantly faster ($p < 10^{-40}$) than the one in the denominator, *i.e.* **AMDP** is faster than **NH**, which is in turn faster than **BASE**; this is consistent with our hypothesis. Comparing **AMDP** to **BASE**, we find that **AMDP** is twice as fast in over half the cases, 4 times as fast in a quarter of the cases, and can reach 20 times speedup. However, **AMDP** is also slower than **BASE** on 23% of the cases; of these, half are within 5% of **BASE**, but the other half is up to 3 times slower. Inspecting these cases suggests that the slowdown is due to overhead from instantiating multiple planning tasks in the hierarchy; this overhead is especially prominent in relatively small domains like Cleanup World. Note that in the worst case this is less than a 2s absolute time difference.

From a computational standpoint, the primary advantage of hierarchy is space and time abstraction. To illustrate the potential benefit of using hierarchical planners in larger domains, we doubled the size of the original Cleanup domain and ran the same experiments. Ideally, this should have no effect on L_1 and L_2 tasks, since these tasks are agnostic to the discretization of the world. The results are shown in Figure 4.6b, which again are consistent with our hypothesis. Somewhat surprisingly though, while **NH** still outperforms **BASE** ($p < 10^{-150}$), it was much less efficient than **AMDP**, which shows that the hierarchy itself was insufficient; the heuristic also plays an important role. Additionally, **NH** suffered from two outliers, where the planning problem became more complex because the solution was constrained to conform to the hierarchy; this is a well-known tradeoff in hierarchical planning [**?**]. The use of heuristics in **AMDP** mitigated this issue. **AMDP** times almost stayed the same compared to the regular domain, hence outperforming **BASE** and **NH** ($p < 10^{-200}$). The larger domain size also reduced the effect of hierarchical planning overhead: **AMDP** was only slower than **BASE** in 10% of the cases, all within $< 4\%$ of the time it took for **BASE**. Comparing **AMDP** to **BASE**, we find that **AMDP** is 8 times as fast in over half the cases, 100 times as fast in a quarter of the cases, and can reach up to 3 orders of magnitude in speedup. In absolute time, **AMDP** took < 1s on 90% of the tasks; by contrast, **BASE** takes > 20s on half the tasks.

4.4.5 Robot Demonstration

Using the trained grounding model and the corresponding AMDP hierarchy, we tested with a Turtlebot on a small-scale version of the Cleanup World domain. To accommodate the continuous action space of the Turtlebot, the low-level, primitive actions at L_0 of the AMDP were swapped out for move forward, backward, and bidirectional rotation actions; all other levels of the AMDP remained unchanged. The low level commands used closed loop control policies, which were sent to the robot using the Robot Operating System [126]. Spoken commands were provided by an expert human user instructing the robot to navigate from one room to another. These verbal commands were converted from speech to text using Google's Speech API [45] before being grounded with the trained Single-RNN model. The resulting grounding, with both the AMDP hierarchy level and reward function, fed directly into the AMDP planner resulting in almost-instantaneous planning and execution. Numerous commands ranging from the low-level "Go north" all the way to the high-level "Take the block to the green room" were planned and executed using the AMDP with imperceivable delays after the conversion from speech to text. A video demonstration of the end-to-end system is available online[2].

4.5 Discussion

Overall, our best grounding model, Single-RNN, performed very well, correctly grounding commands much of the time; however, it still experienced errors. At the lowest level of abstraction, the model experienced some confusion between robot navigation (agentInRoom) and object manipulation (blockInRoom) tasks. In the dataset, some users explicitly mentioned the desired object in object manipulation tasks while others did not; without explicit mention of the object, these commands were almost identical to those instructing the robot to navigate to a particular room. For example, one com-

[2]https://youtu.be/9bU2oE5RtvU

mand that was correctly identified as instructing the robot to take the chair to the green room in Figure 4.3a is “Go down...west until you hit the chair, push chair north...” A misclassified command for the same task was “Go south...west...north...” These commands ask for the same directions with the same amount of repetition (omitted) but only one mentions the object of interest allowing for the correct grounding. Overall, 83.3% of green room navigation tasks were grounded correctly while 16.7% were mistaken for green room object manipulation tasks.

Another source of error involved an interpretation issue in the video demonstrations presented to users. The robot agent shown to users as in Figure 4.3a faces south and this orientation was assumed by the majority of users; however, some users referred to this direction as north (in the perspective of the robot agent). This confusion led to some errors in the grounding of commands instructing the robot to move a single step in one of the four cardinal directions. Logically, these conflicts in language caused errors for each of the cardinal directions as 31.25% of north commands were classified as south and 15% of east commands were labeled as west.

Finally, there were various forms of human error throughout the collected data. In many cases, users committed typos that actually affected the grounding result (*e.g.* asking the robot to take the chair back to the green room instead of the observed blue room). For some tasks, users often demonstrated some difficulty understanding the abstraction hierarchy described to them resulting in commands that partially belong to a different level of abstraction than what was requested. In order to avoid embedding a strong prior or limiting the natural variation of the data, no preprocessing was performed in an attempt to correct or remove these commands. A stronger data collection approach might involve adding a human validation step and asking separate users to verify that the supplied commands do translate back to the original video demonstrations under the given language constraints as in MacMahon et al. [99].

4.6 Conclusion

We presented a system for interpreting and grounding natural language commands to a mobile-manipulator robot at multiple levels of abstraction. Moreover, our Turtlebot evaluation demonstrates that this system works well in real-world environments and is an encouraging step towards seamless human-robot interaction.

Our proposed language-grounding models significantly outperform the previous state-of-the-art method for mapping natural language commands to reward functions. A major reason for such an improvement is modelling the mapping problem as a classification problem to a specific reward function in the AMDP hierarchy. Classification is a significantly easier problem than the type of sequence to sequence mapping attempted by IBM models. This is because in classification we are trying to map language in one of n bins, but the number of possible outputs for sequence to sequence mapping are infinite. However, the type of functions learned using sequence to sequence mapping are richer and can express more behaviors than classification. For example, we would like to give commands like "go along the greenway," which cannot necessarily be put in a single classification bin, as we will require a bin for every object we can travel along, and then a bin for every possible preposition, adjective and adverb. In the next two chapters we look at sequence to sequence methods for different problems so their outputs are more expressive than classification.

Chapter 5

Grounding Language to Linear Temporal Logic

Parts of this chapter in the dissertation have been previously presented at RSS 2018 as "Sequence-to-Sequence Language Grounding of Non-Markovian Task Specifications," with Dilip Arumugam, Lawson L. S. Wong and Stefanie Tellex [48]. The core idea for this project came from me as natural language commands were more diverse than just goal based commands. I performed the work on the planner, data collection and robot experiments, and Dilip worked on the data collection and language grounding results initially, which I improved later.

The broad spectrum of tasks that humans would like to see interpreted and executed by robots extends beyond realizing a particular goal configuration. A slightly richer space of tasks includes those that aim to induce a constrained or repetitive robot execution. In the former category, a person may instruct a robot in the home to "go down the right side of the hallway and to the bedroom," restricting the space of possible paths the robot might take to reach its destination. Similarly, a command in the latter category may be “watch the sink for dirty dishes and clean any that you see,” implying that the robot should enter a loop of repeatedly scanning for dirty dishes and cleaning any that appear. A robot system that can adequately represent and enable untrained users to express these complex behaviors would constitute an important step forward in the area of human-robot interaction.

Existing approaches such as Linear Temporal Logic (LTL) [125, 100] allow these behaviors to be expressed as formal logical functions, enabling the automatic generation of robot controllers to execute these behaviors. While incredibly powerful, the average, non-expert user cannot be expected to hold the requisite knowledge for expressing arbitrarily complex behaviors via LTL. Consequently, we turn to natural language as a familiar interface through which non-expert users may convey their intent and desires while escaping the need for low-level programming knowledge. Unfortunately, due to the rich underlying semantics of LTL, there is no existing approach that takes open-ended natural language commands and maps directly to general LTL expressions.

This chapter presents an approach for enabling a robot to learn a mapping between English commands and LTL expressions. We prese nt the currently the largest dataset that maps between English and LTL in a model environment. We employ neural sequence-to-sequence learning models to infer the LTL sequence corresponding to a given natural language input sequence. While existing sequence-to-sequence models require large amounts of data to perform tasks at scale (such as English-French neural machine translation), we are not able to easily access such vast quantities of data due to the associated cost of obtaining annotations of natural language with their equivalent LTL logical forms. Consequently, we also outline a data collection pipeline through Amazon Mechanical Turk and employ data augmentation techniques to expand the size of the parallel corpus. Finally, we conduct an analysis of our proposed sequence-to-sequence grounding models and their ability to generalize to novel natural language

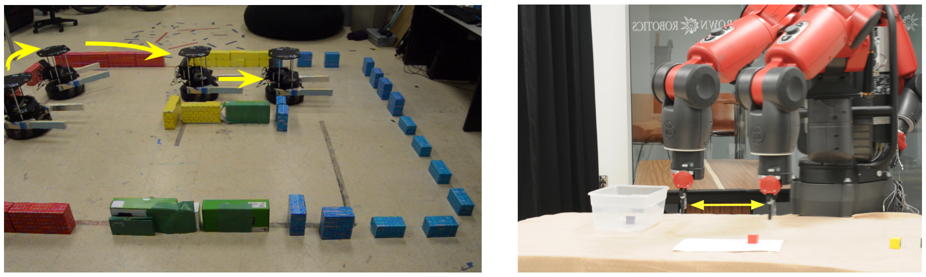

(a) (b)

Figure 5.1: (a)In this robot demonstration we provided a natural language command of “Go through the yellow room to the blue room”. The command is mapped to the LTL formula “$\Diamond(Y \wedge \Diamond B)$”, which implies eventually satisfying the combination of going to yellow and subsequently reaching the blue room. In this work we map from natural language commands to such LTL formulae with sequence-to-sequence approaches. (b) In this demonstration Baxter was given the command to “search the table for a cube and drop it into the bin and repeat.” The language is grounded to the LTL formula “$\Box(S\mathcal{U}\neg A \wedge \Diamond A)$,” which implies always scan until a block is found outside the bin, if found put all blocks in the bin. Here S is the proposition to scan and A is the proposition for all blocks to be in the bin. The figure shows Baxter scanning while satisfying this command.

commands and generate LTL expressions never seen during training.

We evaluate our approach in simulation as well as using a real robot. We collected a corpus of more than 700 sentences mapping to LTL commands to a robot in a simulated 2D environment. We showed that we are able to correctly ground LTL expressions for a wide variety of commands. We demonstrate our approach on a Baxter based pick-and-place domain, and on a mobile robot (the Turtlebot). Our approach extends the space of achievable robot behaviors to include several non-Markovian objectives including repetitive patrolling, going through specified locations, and avoiding specified locations.

5.1 Related Work

The question of how to effectively convert between natural language instructions and robot behavior has been widely studied in previous work [155, 26, 99, 82, 38, 140, 57, 20, 28, 3, 98, 92, 107, 91, 123, 19, 8, 69, 111]. So far, there have been three categories of behavior specifications that these works have mapped natural language to: action sequences, goal states, and LTL specifications. They differ in terms of the following four desiderata:

- Environment-agnostic: The same natural language instruction may lead to different primitive actions being executed across different environments, or if the world is stochastic. Grounding language directly to action sequences is environment-dependent; the other two (goals and LTL) are not.
- Compact representation: Action sequences have length proportional to the trajectory, whereas goals and LTL formulae can be arbitrarily shorter.
- Non-Markovian specifications: Some typical robot behavior cannot be easily expressed as a goal state, such as "put down the knife before moving". This is trivial to specify in an action sequence, and is often expressible in LTL.
- Efficient planning: Action sequences require no planning and are therefore trivially efficient. For reaching goal states, efficient algorithms exist both within classical and decision-theoretic planning (MDPs) [121, 93]. Control policy synthesis for LTL is (doubly) exponential in the formula length in the worst case [32], although it is typically faster in practice, and many efficient fragments and approximations have been proposed.

Since we want to ground language for non-Markovian specifications in potentially novel and stochastic environments, we adopt LTL as our target specification in this work, and will only review grounding language to temporal logic in the remainder of this section.

Kress-Gazit et al. [81, 82] pioneered the area of grounding language to LTL, although the initial work was limited to grounding a fragment termed "structured English". Subsequently, Dzifcak et al. [38] designed a combinatorial categorial grammar

(CCG) that grounds a different fragment of English to the more-general computational tree logic (CTL*); however, this work suffers from the need to manually construct a grammar. Lignos et al. [92] instead used off-the-shelf parsers to interpret the full space of natural language, but do not provide a method to learn from new robot-specific language, and therefore their approach may be limited by the particular corpus used by the existing parser. By contrast, Boteanu et al. [19] collected a crowdsourced corpus of block-sorting instructions to train a Verifiable Distributed Correspondence Graph model that mapped natural language to structured English. In this work, we present a corpus that is an order of magnitude larger, and apply the sequence-to-sequence framework that allows grounding to the full space of LTL formulae (instead of the GR(1) fragment). In an orthogonal direction, Raman et al. [128] demonstrated the benefit of verifying LTL behavior by identifying unsatisfiable parts of LTL formulae and reporting which natural language commands resulted in these failures. Although verifiable behavior is a significant advantage of using LTL [83], we do not focus on verification in this work.

Modeling agent behavior with LTL has been of interest for a while [95, 82]. There have been various variants of LTL developed over the years for different reasons. We describe a few here, for a more detailed treatment please refer Kress-Gazit et al. [84]. To handle uncertainty, a probabilistic version of LTL, Probabilistic Computation Tree Logic [52] has been defined so as to evaluate over states of an MDP. Further, since LTL has an infinite time horizon, approaches have tried to shorten this time horizon so we verify properties to be valid for a certain time horizon [142]. Geometric LTL (GLTL) [94] is another such formulation for specifying goals for an MDP using LTL formuale, such that the formulae are only valid for a chosen time window. In this work, we use GLTL to represent temporal behaviors because it allows planning with traditional MDP planners. However, our use of the neural sequence-to-sequence framework grants us flexibility, and we may also consider other variations of LTL to map natural language to, and perform non-Markovian behaviors. We leave exploration of other LTL variants for target specification from natural language to future work.

5.2 Approach

In order to fully specify our system for converting natural language to robot behavior, we first describe our problem setting and clarify its connection to the GLTL semantic representation inferred by our grounding model. We then go on to outline the recurrent neural network architectures used for mapping between natural language and GLTL expressions.

5.2.1 Problem Setting

Again we formalize the problem of language grounding within the context of an Object-oriented Markov Decision Process (OO-MDP), built upon an MDP with propositions for goal conditions and constraints.

Following the framework introduced by MacGlashan et al. [98] and Howard et al. [57], we treat natural language as the specification of a latent reward function that completes the definition of an otherwise fully-specified MDP. We use a language grounding model to arrive at a more consolidated, semantic representation of that reward function, thereby completing the MDP and allowing it to be passed to an arbitrary planning algorithm for generating robot behavior. More specifically, we think of each natural language command as specifying some latent LTL formula encoding the desired behavior. The OO-MDP propositional functions serve as possible atoms of the LTL expressions, creating an expressive language for defining tasks. In particular, LTL formulae are sufficiently expressive to subsume semantic representations used in previous goal-based language grounding work such as MacGlashan et al. [98] and Arumugam et al. [8].

5.2.2 Geometric linear temporal logic (GLTL)

Linear temporal logic has the following grammatical syntax:

$$\phi := \texttt{atom} \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \Box\phi \mid \Diamond\phi \mid \phi\mathcal{U}\psi \mid \bigcirc\phi$$

$\texttt{atom}$ denotes an atomic proposition; $\neg$, $\wedge$, and $\vee$ are logical "not", "and", and "or"; $\Box$ denotes "always", $\Diamond$ denotes "eventually", $\mathcal{U}$ denotes "until", and $\bigcirc$ denotes "next". Semantic interpretations of temporal logic operators can be found in Manna and Pnueli [100].[1]

Following work done by Bacchus et al. [9] for specifying reward functions over temporal sequences via LTL, Littman et al. [94] introduced the geometric LTL (GLTL) variant that replaces the standard LTL operators with "soft" substitutes. Effectively, these probabilistic operators are equivalent to their hard LTL counterparts but satisfying each operator is restricted to some bounded window of time as determined by a draw from a geometric distribution. More concretely, instead of $\Box p$, representing that p always holds true indefinitely, GLTL would have $\Box_\mu p$ for indicating that p holds for the next $k \sim \mathrm{Geom}(\mu)$ timesteps. Similarly, $\Diamond_\mu p$ and $q\mathcal{U}_\mu p$ correspond to p becoming true in the next k timesteps and q holding true at least until p becomes true in the next k timesteps respectively where, again, $k \sim \mathrm{Geom}(\mu)$. While sacrificing the guarantees and proofs of correctness typical of LTL, the probabilistic nature of GLTL enables learning while specifying rewards in a generalizable, environment-independent fashion. We include more on how GLTL ties into our approach in Section 5.2.1 and, for more information on GLTL itself, please consult [94].

Crucially, Littman et al. [94] connect GLTL back to MDPs in a way that not only specifies the desired behavior without overfitting to a single environment instance by folding the stochastic semantics of the GLTL expression into the stochastic transitions

[1]In this work we do not collect data on behavior requiring the "next" operator, and therefore it never appears in our grounded formulae; however, the same framework can be used if relevant data for "next" is collected.

of the environment, enabling the application of standard reinforcement learning and planning techniques. In particular, each atomic proposition of a GLTL expression has an associated three-state MDP consisting of an initial, accepting, and rejecting state. From the initial state, there is a single action in this *specification MDP* that will transition to the accepting state if the proposition holds and move to the rejecting state otherwise. Each GLTL operator represents some fixed transformation or combination of these MDPs resulting in a new specification MDP. Once an inferred GLTL expression is converted to its corresponding specification MDP, it is combined with the separate environment MDP resulting in an MDP whose solution represents a policy capturing the desired behavior. For the full details of specification MDP construction and combination with an environment MDP, please see Littman et al. [94]. We model the procedure of mapping from natural language to GLTL expressions as a neural machine translation problem.

Although we choose GLTL paired with an MDP to find policies corresponding to LTL expressions, our language grounding system can work with any framework for computing a satisfying controller for the robot, such as those described in the related work [84]. Notice that the switch is commensurate with defining a new machine translation problem with a target language defined by the syntax of the alternative framework.

5.2.3 Mapping Language to GLTL

In order to handle the task of translating from natural language to GLTL expressions, we turn to recent neural-network architectures for sequence learning that have already proven to be incredibly performant in other machine translation tasks. Further, in this work to improve generalizability we use pre-trained GloVe embeddings [124]. We are presented with a sequence taken from some source language $\mathbf{x} = [x_1, \dots, x_N]$ and would like to infer a corresponding sequence of some target language

$\mathbf{y} = [y_1, \ldots, y_M]$. Given a translation model with parameters θ, we look to identify the most likely target sequence and decompose its corresponding probability into the product of partial translation probabilities:

$$p(\mathbf{y}|\mathbf{x}, \theta) = \prod_{m=1}^{M} p(y_m|\mathbf{x}, \mathbf{y}_{<m}, \theta), \tag{5.1}$$

where $\mathbf{y}_{<m} = [y_1, \ldots, y_{m-1}]$.

Treating the space of natural language commands as a source language and GLTL expressions as a target language, we collect a parallel corpus and optimize the neural-network architecture parameters θ (see Section 6.4 for more details on the data collection procedure). Specifically, we leverage the recurrent neural network (RNN) encoder-decoder framework [27, 136] extended by Bahdanau et al. [10].

Widely used across a variety of natural language tasks, RNNs are models that map sequential inputs to high-dimensional output spaces, using recurrent cells that maintain an internal or hidden-state representation of the sequence processed thus far [56, 27, 49]. Neural sequence-to-sequence learning use two distinct RNN models, an encoder and decoder, to map between source and target language sequences. An encoder processes an input sequence and, for each input token, produces an output and an updated hidden state. While the hidden state stores a global summary of the overall sequence, the output encodes local information computed from the current hidden state and the current input token. After all input tokens are fed to the encoder, there is a resulting sequence of encoder outputs and hidden states. Subsequently, a decoder model generates target language sequences by mapping an input target language symbol to a probability distribution over the target language vocabulary [15]. Connecting the two models is the final hidden state of the encoder RNN cell (that is, the hidden state after processing the entire input sequence) which is used to initialize the hidden state of the decoder RNN cell [27, 136]. The encoder hidden state is initialized with an all zeros vector and a special start token is fed as the first input to the decoder in order to initialize decoding.

In both the encoder and decoder of our model, we use the Gated Recurrent Unit (GRU) [27] as the core RNN cell. Briefly, a GRU is one type of RNN cell that, using the previous hidden state s_{t-1} and current input token's embedding x_t, performs the following computations:

$$\text{Reset gate:} \quad r_t = \sigma(\mathbf{W_r} \cdot x_t + \mathbf{U_r} \cdot s_{t-1} + \mathbf{b_r}), \tag{5.2}$$

$$\text{Output:} \quad g_t = \tanh(\mathbf{W_g} \cdot x_t + \mathbf{U_g} \cdot (r_t \odot s_{t-1}) + \mathbf{b_g}), \tag{5.3}$$

$$\text{Update gate:} \quad z_t = \sigma(\mathbf{W_z} \cdot x_t + \mathbf{U_z} \cdot s_{t-1} + \mathbf{b_z}), \tag{5.4}$$

$$\text{State update:} \quad s_t = (1 - z_t) \odot s_{t-1} + z_t \odot g_t, \tag{5.5}$$

resulting in an output vector g_t and a new hidden state s_t where $(\cdot)$ and $(\odot)$ denote matrix-vector and Hadamard products respectively. Intuitively, r_t is a "reset" gate affecting how hidden state information is combined with the current input to produce the cell output, g_t, and z_t is an "update" gate negotiating how much information is preserved within the hidden state. All parameters in bold denote trainable parameters of the cell that are optimized during training of the entire architecture via backpropagation.

Given the GRU encoder and decoder, f_{enc} and f_{dec}, words of the input sequence are first mapped to their corresponding vector representations via an embedding look-up. Intuitively, since the individual tokens can simply be represented as integers, the word embeddings provide a high-dimensional representation that is optimized as part of the neural network and can be used to capture semantic information about each individual word. In our previous experiments [48] we learned the word embeddings as part of our network. However, this work we use pre-trained GloVe embeddings [124]. GloVe embeddings are vector representations for words learned using unsupervised training based on co-occurrence statistics of words. Where previous embedding learning methods were learning embeddings that represented word co-occurrences in a sentence, GloVe looks at co-occurrence statistics with respect to a context. Words with similar co-occurrence statistics for a context would have closer embeddings. This property of the embeddings helped improve performance across different types of linguistic tasks[124]. Our hypothesis here is that the GloVe embedding might help process novel words, and provide some more generalization on the encoder side.

After receiving the word embedding we feed each input embedding, $\hat{\mathbf{x}}_\mathbf{j}$, in sequence producing a corresponding sequence of encoder outputs, $\big[h_1, \ldots, h_N\big]$, and hidden states $\big[s_1, \ldots, s_N\big]$, where each h_j, s_j comes from:

$$h_j, s_j = f_{enc}(s_{j-1}, \hat{\mathbf{x}}_j). \tag{5.6}$$

Once the input sequence has been processed, the final encoder hidden state, s_N is used to initialize the RNN cell of the decoder. During decoding, the previously inferred token of the output sequence is mapped to a probability distribution over the target vocabulary from which the next output token is sampled:

$$p(y_i|\mathbf{x}, \mathbf{y}_{<m}, \theta) = f_{dec}(v_{i-1}, \hat{\mathbf{y}}_{i-1}). \tag{5.7}$$

Here $\hat{\mathbf{y}}_{i-1}$ denotes the embedding for the previously decoded token, $\hat{y}_{i-1}$. In all of our experiments, we use a greedy decoding scheme and treat the token in the distribution with highest probability as the inferred token, $\hat{y}_i$.

Subsequent work by Bahdanau et al. [10] proposed learning a weighting or attention scheme to selectively utilize encoder output information during decoding. The resulting Bahdanau attention mechanism reparameterizes the decoder as a function of the previous token embedding and a context vector. At each step of decoding, the context vector weights the information of the encoder outputs $\big[h_1, \ldots, h_N\big]$ according to:

$$c_i = \sum_{j=1}^{N} \alpha_{ij} h_j, \tag{5.8}$$

where α_i is a weight on the information carried in h_j. The individual attention weights are computed as outputs of a feedforward neural network, $a(v_i, \left[h_1, \ldots, h_N\right])$, with a final softmax activation and where v_i is the current hidden state of the decoder RNN cell. Accordingly, decoding occurs by greedily sampling the next token distribution:

$$p(y_i|\mathbf{x}, \mathbf{y}_{<m}, \theta) = f_{dec}(v_{i-1}, \hat{\mathbf{y}}_{i-1}, c_i). \tag{5.9}$$

Even with alternatives to Bahdanau attention, attention weights are typically computed as a function of the current decoder hidden state and the encoder outputs [96]. Given the rigid structure of GLTL as a target output language, we examine an alternative attention mechanism that computes attention weights purely as a function of decoder parameters. More formally, we propose an *encoder-agnostic* attention mechanism where each attention weight α_i is computed by a feedforward neural network $a(\hat{\mathbf{y}}_{i-1}, v_i)$ that takes the previously decoded token embedding and the current decoder hidden state as inputs. This attention scheme captures the idea that the previously decoded token and current state of the target translation generated thus far offer a clearer signal for weighting the encoder outputs than the encoder outputs themselves. Although seemingly counterintuitive, we consider a particular scenario where the sequence-to-sequence grounding model must generalize to a language command at test time that corresponds to a novel GLTL expression never seen during training. Instead of being subject to the learned mechanics of the source language that may vary dramatically at test time, the encoder-agnostic attention scheme would maintain focus on the GLTL target language side that exhibits significantly less variation and follows a small, well-defined lexicon.

Methods	Domain #1		Domain #2	
	Original	Expanded	Original	Expanded
Seq2Seq	$93.10 \pm 1.60\%$	$95.25 \pm 0.30\%$	$86.09 \pm 1.03\%$	$93.42 \pm 0.96\%$
Seq2Seq + Bahdanau Attention	$\mathbf{93.45 \pm 0.60\%}$	$\mathbf{95.51 \pm 0.11\%}$	$\mathbf{87.15 \pm 0.36\%}$	$93.78 \pm 0.29\%$
Seq2Seq + Encoder-Agnostic Attention	$93.18 \pm 0.94\%$	$94.98 \pm 0.30\%$	$86.47 \pm 0.72\%$	$\mathbf{93.92 \pm 0.44\%}$
Seq2Seq + GloVe + Encoder-Agnostic Att.	$93.25 \pm 0.86\%$	$93.32 \pm 0.78\%$	$84.54 \pm 4.95\%$	$92.34 \pm 1.37\%$

Figure 5.2: Accuracy and standard deviation of 5-fold cross validation for each grounding model and domain, averaged over 3 independent runs.

Unlike the architecture outlined in Bahdanau et al. [10], we found a single, forward RNN encoder was sufficient for our task and opted not to use a bi-directional RNN encoder in favor of reduced training time and sample complexity. All models were implemented in PyTorch [122] and trained using the Adam optimizer [72] with a learning rate of 0.001. Embedding and RNN output sizes were set to 50 and 256 units respectively. Additionally, we made use of dropout regularization [134] before and after both the encoder and decoder GRUs with a keep probability of 0.8. For the pretrainined GloVe embeddings of size 50 we used a pre-trained look-up table of GloVe embeddings[124]. We did not further train this embedding layer of the network. We found that reversing all input sequences provided a small increase in grounding accuracy. For each training sample, a random choice was made between providing the ground truth output token, y, (teacher forcing [150]) and providing the actual decoded symbol, $\hat{y}$, with 0.5 probability.

5.3 Experiments

We now outline the two domains used for evaluating our approach as well as the details of our data collection procedure for training the three presented grounding models.

5.3.1 Mobile-Manipulation Robot Domain

Cleanup World [98] consists of a single robot agent acting within an environment characterized by distinctly colored rooms and movable objects. We chose this domain for our experiments as it allows us to express a wide variety of constrained execution tasks. For the purposes of our experiments, these restrictions took on one of two forms: the robot would need to reach it's target destination by either passing through or explicitly avoiding a particular room in the environment.

5.3.2 Baxter Pick-and Place Domain

In order to showcase instances of repetitive robot execution, we designed a pick-and-place domain where a Baxter robot must patrol the environment waiting for sudden events that trigger some desired behavior. More concretely, the domain is defined by distinct table and bin regions where colored blocks may reside. These blocks have only two attributes: first for location, table or bin; second for color where the blocks can be red or green or blue or yellow. The state of the domain can get very large as the total number of blocks is not restricted or predetermined. Initially, there may not be any blocks present and so the robot must utilize a scan action to survey the area until an appropriately colored block can be picked up and moved from the table to the bin. A scan operation reveals a new block, if it was "placed" on the table. The placement is done by a human user. The pick-and-place action picks a block and moves it to the bin. Note that the reachable state space of this domain grows rapidly as the number of blocks on the table increases, as each block can be at different locations. Given an agent with only two scan and pick-and-place actions at its disposal, we experimented with two core task types: placing all blocks found during scanning into the bin or placing only blocks of a certain chosen color into the bin. These tasks were chosen for their repetitive nature, while also having an alertness property, where some colors are ignored but some are acted upon. Further, these tasks have many real life analogs like a cleaning robot that patrols rooms looking for spills, or an industrial robot search for faults in an assembly line with a repeated pattern.

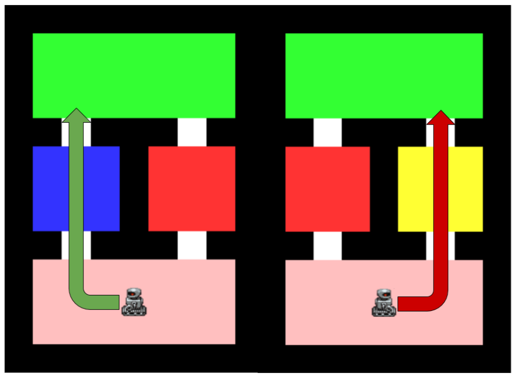


Figure 5.3: Static mobile-manipulation domain images presented to AMT users for annotation. Positive (green) and negative (red) labels were applied to the images so that AMT users could infer the constraint reflected in the robot's behavior.

5.3.3 Data-collection Procedure

An Amazon Mechanical Turk (AMT) user study was conducted in order to collect data for training our neural sequence-to-sequence grounding models mapping natural language commands to GLTL expressions in each of our domains. To construct the parallel corpus, annotators were presented with static images (for Cleanup World) and short video clips (for pick-and-place) depicting a particular robot behavior as shown in our video[2]. Users were then asked to provide a single sentence instruction that would induce the observed behavior. For the mobile-manipulation robot domain, sample images provided to AMT annotators can be seen in Figure 5.3. Specifically, these images were displayed to users with positive (green arrow) and negative (red arrow) labels so that annotators could infer the constraint being placed on the robot's execution.

Curiously, we found that annotators were more inclined to specify positive over negative behavior in their instructions. For instance, an attempt to collect data for behavior matching the command "go to the green and avoid the yellow room" would often result in commands where users would instruct the robot to navigate to the green room utilizing whichever rooms were designated under the positively labeled images. Al-

[2]`https://streamable.com/8lqab`

Figure 5.4: Example commands and corresponding GLTL formulae. R, G, and B are propositions in Cleanup World for testing if the agent is in the Red, Green or Blue rooms respectively. S, A and N_R are propositions for the pick-and-place domain. S tests if the table has been scanned once; A tests if all blocks are in the bin; and N_R test if all blocks expect the red colored ones are in the bin.

Example Command	GLTL Expression
Go to the green room.	$\Diamond G$
Go into the red room.	$\Diamond R$
Enter blue room via green room.	$\Diamond(G \wedge \Diamond B)$
Go through the yellow or red room, and enter the blue room	$\Diamond((R \vee Y) \wedge \Diamond B)$
Go to the blue room but avoid the red room.	$\Diamond B \wedge \neg\Box R$
While avoiding yellow navigate to green.	$\Diamond G \wedge \neg\Box Y$
Scan for blocks and insert any found into bin.	$\Box((S\mathcal{U}\neg A) \wedge \Diamond A)$
Look for and pick up any non red cubes and put them in crate.	$\Box((S\mathcal{U}\neg N_R) \wedge \Diamond N_R)$

though technically correct, these instructions pose an obstacle as the intended, ground-truth GLTL expression (containing, for example, the token for the yellow room) would never include a symbol with semantic meaning associated with the rooms mentioned (for example, the blue or red rooms). In order to address this problem, a manual re-labeling of data was performed such that samples whose instruction did not align to the intended GLTL formula were instead mapped to the corresponding GLTL formula consistent with the instruction. Filtering commands that reflected a clear misunderstanding of the annotation task resulted in parallel corpora consisting of 501 and 345 commands for the mobile-manipulation and patrol domains respectively. In aggregate, these commands reflected a total of 15 and 2 unique GLTL expressions respectively. Examples of natural language commands and their corresponding GLTL formulae can be seen in Figure 5.4.

In order to supply additional data for the mobile-manipulation domain, we utilized a subset of the Cleanup World dataset collected by Arumugam et al. [8] consisting of 356 agent navigation and block manipulation commands, swapping their Markov reward function representation for the GLTL equivalent. Together, the combined dataset denotes the *original dataset* for the mobile manipulation domain used throughout all of our experiments. The pick-and-place domain original dataset received no extra commands. To further expand on the data present for learning across both domains, a synthetic data expansion procedure was applied to both datasets. Excluding the minority of commands pertaining to block manipulation behavior, all other commands were mapped to new commands through the substitution of color words in the natural language and the equivalent swapping of atoms in the corresponding GLTL expressions. For example, the command "move to the red room" and corresponding expression $\Diamond R$ would be mapped to three new language commands (one for each of the blue, green and yellow rooms) along with the corresponding GLTL expressions ($\Diamond B$, $\Diamond G$, $\Diamond Y$). This expansion resulted in two *expanded datasets* for each of the domains consisting of 3382 and 745 commands respectively reflecting a total of 39 and 5 unique GLTL expressions.

5.3.4 Language Grounding

We conducted 5-fold cross validation experiments across three grounding models and present the language grounding accuracy means and standard deviations in Table 5.2. Results were averaged over 3 independent trials with distinct random seeds. For each instance, correctness was determined by comparing the complete, greedily-decoded GLTL formula to ground truth. Training was done using two independent parallel corpora, one for each of the domains outlined in above. Additionally, we report results on both the original datasets, consisting of natural language commands exactly as collected through AMT, and the expanded datasets synthetically generated via the procedure outlined in Section 5.3.3. Notably, we find that all three models exhibit roughly identical performance despite the use of two different attention mechanisms. We suspect that the lack of an effect is due to the dramatically smaller vocabulary size of GLTL by comparison to traditional neural machine translation problems. However, we do find the use of an attention mechanism to have some effect on enabling generalization and the inference logical forms not seen during training.

In order to establish how well each grounding model captures the semantics of natural language and GLTL expressions, we conducted an experiment to assess the capacity for each model to infer novel GLTL expressions. Focusing on the mobile manipulation domain, we randomly sampled varying percentages of the 39 unique GLTL expressions represented across the collected within the expanded corpus of 3382 commands. All samples in the parallel corpus associated with the random sampled commands were used as training data while the entire remainder of the corpus was treated as a held-out test set consisting only of GLTL expressions not seen during training. The results of the experiment are shown in Figure 5.5 with error bars denoting 95% confidence intervals computed over 10 independent trials. On average, we find that our encoder-agnostic attention scheme is slightly more adept at generalizing to novel commands. The results altogether, however, suggest that this type of generalization is still an open challenge for these models that traditionally excel in standard neural machine translation tasks that enjoy access to vast quantities of parallel training data. We believe that adapting these techniques to better operate in the extremely low resource area of robot language learning is an important direction for future research.

5.4 Results

5.4.1 Language Grounding

Our results confirm that a standard neural sequence learning model, without the use of an attention mechanism, is sufficient for achieving highly accurate language grounding to GLTL logical forms. We do, however, witness the benefits of attention when faced with the challenge of generalizing beyond the space of GLTL expressions seen at training time, although, on average, our encoder-agnostic attention scheme does represent

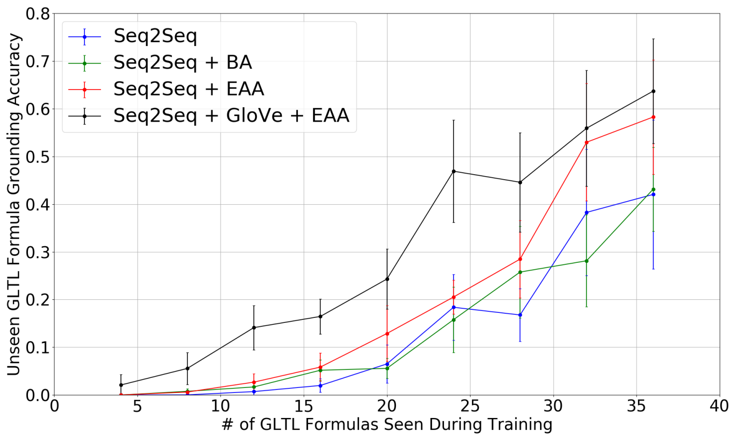


Figure 5.5: Grounding accuracies of various sequence-to-sequence models evaluated on held-out subsets of the training data consisting entirely of novel GLTL expressions. Error bars represent 95% confidence intervals computed over 10 independent runs.

a fairly substantial improvement. There is further improvement when the encoder-agnostic attention scheme is used along with GloVe embeddings. In data-starved conditions with very few annotations pre-trained word embeddings improve results drastically, and clearly beat simple neural sequence models. However, we expect these benefits would transfer if the simple neural sequence approaches were also given the benefit of pre-trained word embeddings. Overall, these results still leave much to be desired. This problem of generalization without any training experience (or zero-shot generalization) within neural sequence-to-sequence models is further studied by Lake and Baroni [88] who conduct a series of experiments assessing various dimensions on generalization within a new domain for mapping navigation commands to action sequences. Notably, their study finds that generalization is only reliable when novel test time constituents are observed within a variety of contexts during training. As these neural-based translation approaches continue to improve, the question of how to make them more amenable to sparse-data, robot-learning settings will become increasingly important.

Beyond the challenge of generalization, there were a few other sources of error across the three grounding models. In the mobile-manipulation domain corpus, the longest language command consisted of 27 tokens. Demonstrating another well-known challenge of sequence learning and RNN models in general, these longer, outlier sequences posed a challenge and often produced invalid GLTL expressions. Despite the use of input sequence reversal to help combat this challenge, increasing sequence length forces the RNN cell to carry information across a larger number of timesteps; after a point, the degradation in the latent semantic information makes accurate translation incredibly difficult. Additionally, we found instances where the models produced a correct GLTL formula type but had the atoms reversed (such as selecting to avoid the target destination room and navigate towards the avoidance room). These commands tended to have minor grammatical errors or lacked certain keywords to indicate the blue *room* or the yellow *area*.

More generally, we note that the previously discussed issues with our approach are only a subset of a much broader space of challenges inherited from neural machine translation. The full space of challenges is perhaps most succinctly and carefully explored in the work of Koehn and Knowles [74] who outline six key deficiencies of general neural machine translation. While these challenges are made quite apparent from the scale of typical translation problems (mapping between millions of vocabulary tokens) a few issues of particular importance to the robotics community include out-of-domain words, low-resource translation, low-frequency words, and long sentences. Note that, in this work, we have already demonstrated and discussed the difficulties of low-resource translation (that is, translation with limited training data) alongside low-frequency words. Given the high-cost of acquiring annotations, robotics datasets are often built within a particular context and geared towards a specific domain. As the demand for these systems grows and requires a single system to operate across domains (for instance, the home and the workplace), the inability for current translation models to handle identical words with context-sensitive translations (or out-of-domain words) will prove to be a bottleneck. Moreover, as commands naturally grow in complexity and encode an increasing number of tasks, so too will the corresponding output sequences, resulting in decreased translation/grounding accuracy. Given the clear mutual interest, a rich direction of future work includes collaborating with the natural language processing community to develop solutions to these problems and bring them to bear on robotics domains.

5.4.2 Cleanup Robot Demonstrations

To further demonstrate the efficacy of our approach, we translated our mobile-manipulation domain into the physical world with a Turtlebot agent. The problem space focused on constrained execution tasks requiring the robot the enter a goal room while either avoiding or passing through specific rooms. As in simulation, our physical set-up consisted of four rooms uniquely identified by color and the agent's position in the world was tracked by a motion capture system. Using the Robot Operating System (ROS) [126] speech to text API, we converted speech utterances to natural language text that could be passed to a trained instance of our grounding model with Bahdanau attention, producing a GLTL formula. Treating the Cleanup World MDP as the environment MDP and identifying the specification MDP of the GLTL formula, we combine the two MDPs and apply a standard MDP planning algorithm to generate robot behavior in real time. The planning is real time as the formulae for these tasks are not very long and GLTL allows fast planning, most of the delays observed in the video are networking delays when using speech-to-text. The primitive north, south, east, west actions of the Cleanup Domain agent were converted into a series of low level ROS Twist messages using hand-written controllers. A video of our robot accurately responding to user commands in real time is online[3].

[3] `https://streamable.com/5uy0e`

5.4.3 Baxter Pick-and-Place Domain

We implemented the pick-and-place domain on the Baxter robot platform and observed the behavior for two command types.

The first command type was to move any block that is placed on the table to the bin. This behavior was highly consistent across repeated trials in simulation. The agent would scan, and a block might be placed during this scan operation. The agent would detect the placed block during the scan operation, and would choose to place the block in one of the following time steps. Conversely, we found that the behavior for selectively placing blocks of a specific color in the bin experienced difficulties during the execution of the inferred GLTL formula.

As described in Section 5.1 a GLTL expression holds true only for certain time steps, as planning within MDPs for an infinite time horizon compromises learnability and performing value backups for planning. Specifically for this task the agent placed non-red blocks into the bin and continued looking for more blocks. We noticed that the agent initially (and correctly) does not pick any red blocks it finds; however, as the number of blocks revealed to the robot increases, it becomes more likely that the robot picks up a red block, as the formula and its corresponding behavior would only be true for a certain number of time steps dependent on the geometric discounting. With a discount factor of 0.99 we noticed that the agent would pick up a red block after about $5-10$ blocks were revealed to the agent when scanning. However, the agent constantly placed more non-red blocks in the bin, even when the number of red blocks was more than the number of rest of the blocks. An easy thing to increase the duration of placing non-red blocks would be to increase the discount factor, however this would lead to increased planning time, as the agent is planning over a longer horizon.

This simulation domain shows that the GLTL formulation can easily handle behaviors that include repetitive subtasks while being able to react to certain, pre-conditioned events. The experiment also helps us understand some of the limitations of GLTL, as an approximation of LTL for a chosen time horizon.

For the implementation of these behaviors on the robot we used fiducials in the form of April Tags [120]. We used the software stack Ein [119] as an interface to command the robot and its inverse kinematics based planner. The fiducials were observed from a wrist camera. To get accurate position estimates through the fiducials we slowed down the movement of the arm during the scanning process. However, the robot itself was being controlled at 10Hz. The actual state of the world, that is, the position and orientation of the blocks was abstracted and the LTL planner only planned over the location of the blocks, bin or table, and the color of the blocks. The transcription, mapping to LTL, and planning is all done real time with this setup. The real time demos are available online[4].

5.5 Conclusion

This chapter demonstrates an approach for mapping between English commands and LTL expressions through neural sequence-to-sequence learning models. We presented techniques for data augmentation, pre-trained word embeddings and a novel attention mechanism that enables the system to map between novel English commands and novel LTL expressions not encountered at training time so long as the constituent LTL atoms have been previously observed. We demonstrated this approach within two domains for mobile-manipulation and pick-and-place tasks as well as on physical robots.

One of the fundamental issues with neural sequence-to-sequence methods is their lack of generalization to unseen data. This issue stems for their inability to learn sequences and functions compositionally. This phenomenon has been studied before by Lake and Baroni [88] and Kim et al. [71]. Further work needs to be performed to introduce compositional learning mechanisms in neural networks. In the next chapter we explicitly look at a non-neural method to learn compositional mappings from

[4]`https://streamable.com/5uy0`

natural language to behavior.

Chapter 6

Learning to Parse Natural Language to Grounded Reward Functions with Weak Supervision

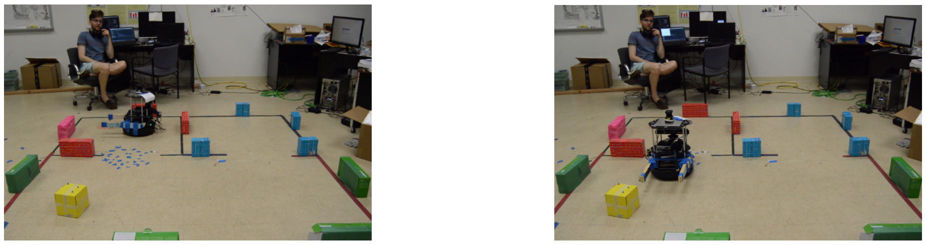
Large parts of this chapter in the dissertation have been previously presented at ICRA 2018 as "Learning to Parse Natural Language to Grounded Reward Functions with Weak Supervision," with Edward C. Williams, Mina Rhee and Stefanie Tellex [149]. The core idea for this project came from both Edward and me. Edward was experimenting with CCG parsers to get quantifiers to work in language grounding for goals. I came up with a way to validate such a weakly supervised parser. I worked with Edward on specifying grammar, data collection, the baseline experiments and the robot demo. I worked on this project as it provided an alternative to con-

nectionist approaches to learning language grounding. It helped us understand the importance of compositionality and the limitations of old school parser learning techniques.

Natural language provides an expressive and accessible method for specifying goals to robotic agents. In this chapter, we represent these goal conditions as lambda-calculus goal-based reward functions under the MDP [13] formalism, which give the agent positive reward for reaching the goal state. Planning then aims to maximize the sum of rewards the agent receives while attempting to reach the goal condition optimally. Planning in the MDP formalism is useful for robotics as it allows for a natural method to deal with stochasticity. The use of reward functions as an intermediate representation allows a separation between the language interpretation and planning components.

We learn a mapping from natural language into the space of lambda-calculus expressions using a semantic parser. This mapping is modeled as a weighted linear Combinatory Categorial Grammar (CCG) [30, 135] parser, which pairs words in a lexicon with lambda-calculus representations of their meaning. More importantly, a CCG provides a mechanism for composing lambda-calculus expressions representing sentences from subexpressions representing their constituent words and phrases. Our lambda-calculus representation interfaces with Object Oriented MDP (OO-MDP) [36] states while enabling the use of complex commands involving comparators, object attributes, relationships between objects, and nested referring expressions.

Existing approaches for learning to map sentences to reward functions require fully supervised data, including complete reward function specifications paired with natural language commands [98, 8]. This requires manual annotation of natural language datasets, and the resulting learned models have limited ability to generalize to unseen tasks. Existing CCG-based approaches that map natural language to trajectory specifications [6, 7] require executing planning during learning to validate proposed logical expressions, which is computationally expensive and does not necessarily allow plan-



(a) State at task initiation

(b) State at task completion

Figure 6.1: The figure shows an example pair of states of the Turtlebot before and after it performed the command to “go to the largest room.” The robot moves from the red room to the large green room, marked using the colored blocks. The robot plans the trajectory using a reward function generated from the natural language command on the fly using a CCG parser we learned using weak supervision from language.

ning in Markov Decision Processes domains. Our method allows for efficient learning using only pre- and post-condition states as annotation, instead of complete reward functions or their derivations, which are used to validate reward functions without executing a planner.

Ours is the first approach to produce compositional reward functions using a CCG semantic parser, and then use the reward function for the purposes of planning on an agent in an MDP. We trained and evaluated our model on data collected from Amazon Mechanical Turk (Figure 6.2) on simulated mobile manipulation tasks in Cleanup Domain [98]. Our results demonstrate effective learning of a weakly supervised parser with an F1 score of 0.82 on a corpus of 23 behaviors. We compare against baselines that use planning to validate parses during learning, and achieve comparable performance with orders of magnitude improvement in computation time during learning. Moreover, we present a robot demonstration of our method in a Turtlebot mobile-manipulation task using 6 weakly annotated demonstrations to learn two behaviors, and also show generalization to unseen tasks as shown in Figure 6.1.

6.1 Related Work

Zettlemoyer and Collins [155] presented a supervised method to learn CCG parsers given natural language annotated with lambda-calculus logical forms. Krishnamurthy and Mitchell [85] demonstrated a weakly supervised method for learning CCG semantic parsers given a syntactically parsed sentence and a knowledge base. Artzi and Zettlemoyer [6] extended this idea of weak supervision to planning in a navigation task by mapping natural language to a logical form specifying a trajectory. The trajectories produced by the CCG parser while learning were compared to demonstration trajectories for a validation signal. However, these plans were not generated on an MDP, and the semantic representations of language directly represented sequences of actions rather than goal conditions. Planning on a physical robot requires modeling stochasticity, which is model with MDPs, to allow agents to recover from failure. As we are only interested in goal based commands, it is not necessary to validate using trajectory information during learning. Instead, we can learn from pre- and post- condition states provided as demonstrations for a natural language command.

Tellex et al. [140] described a method using syntactic parse trees of language to create probabilistic graphical models (PGMs) that can create trajectories that match the natural language phrase. This process of generating trajectories is expensive, hence Howard et al. [57] used the PGMs to generate constraints that be used to plan, separating the language interpretation and planning components of the system. Our method also produces constraints on the end state of a robot's trajectory. While Howard et al. [57] assumes access to a pre-trained syntactic, our method needs pre-defined logical expressions to test attributes of objects. A related line of research (MacGlashan et al. [98], Arumugam et al. [8]) translates natural language to sets of reward functions so that agents can perform planning within an MDP. These methods model the process of mapping natural language to reward functions as sequence to sequence translations [98] and multi-class classification [8] using a relatively simple semantic representation and without the ability to learn and interpret nested referential expressions. We instead represent the task as semantic parsing with a highly compositional semantic representation.

Inverse Reinforcement Learning (IRL) [116] is a method for learning a reward function in an MDP using demonstration trajectories and a feature representation of the state space. In our work, we learn a compositional reward function from pre-specified predicates, natural language, and pre- and post- condition states provided as supervision. IRL learns a richer class of reward functions than the goal-based reward functions we learn in this work. However, our use of natural language to learn compositional reward functions allows for generalizability of the learned reward functions as described in Section 6.6. Another series of approaches map natural language directly to policies in an MDP, learned from demonstrations provided at training time [111, 64]. However, these methods have not been demonstrated to work with small training corpora, which is enabled in our case by the seed lexicon provided to our semantic parser.

6.2 Task Domain

Markov Decision Processes (MDPs) [13] model stochasticity, which is an important factor when trying to plan or learn with robots in the physical world. Hence, we represent our robot manipulation and navigation domain as an MDP. In such a domain, we can define goal-state reward functions as functions that, for a given set of terminal goal states $G \subset \mathcal{E}$, produce the following output:

$$R_G(s,a) = \begin{cases} 1, & \text{if } s \in G \\ 0, & \text{otherwise} \end{cases}$$

We define our set of tasks as those that can be executed by planning over goal-state reward functions in the Cleanup World object manipulation domain [98]. We specify this MDP using Object Oriented MDPs (OO-MDP) [36], which provides a factored representation of an MDP problem. The factorization of the environment and actions is done using objects present in the environment, which is convenient as humans naturally describe world states with respect to objects present in it. Cleanup Domain [98] is an MDP environment generator with an agent, and different numbers of rooms and objects. The object have types, such as “basket”, “chair”, and “block” and attributes, such as “color” and “location”. The agent possesses location and orientation attributes. Rooms have attributes of “color”, “location” and “size”. In this work we use a three room configuration of Cleanup world with changing attributes for rooms and objects and the agent. The agent can take the actions of *north, south, east* and *west*. We have previously in Chapter 3 shown the stochastic nature of this domain, with doors that can lock and unlock and objects that are lost by the agent.

6.3 Method

To map natural language to grounded reward functions, we define a compositional semantic representation that defines sets of goal states from relationships between objects in our OO-MDP domain. We then induce a weighted linear CCG parser [30] mapping natural language into our semantic representation, using pre- and post- condition states as supervision, rather than fully annotating language with semantics. Our learning algorithm is derived from the validation-based perceptron learner of Artzi and Zettlemoyer [6].

6.3.1 Semantic Representation and Execution

To bridge the gap between natural language and OO-MDP reward functions, we define a lambda-calculus semantic representation for language that defines a set of goal states in the OO-MDP domain. Lambda calculus, a computational formalism based on function application and variable binding, is well-explored as a semantic representation for natural language [24, 155, 156, 6]. Its utility for natural language representation comes from its expression of the compositionality of natural language while providing an interface between language and computation.

We make the assumption that natural language commands to our agent define a configuration of the world that the user would like the agent to produce, by re-arranging objects in the world or moving to a different location, is reflected in the nature of our representation. Natural language is represented as a proposition function over states in the MDP. For instance, a complete lambda-calculus expression used as a reward function for the task in Figure 6.2, "go to the chair", is $near(the(\lambda x.agent(x)), the(\lambda y.red(y) \wedge chair(y)))$, which acts as as the goal-state reward function in Section 6.2. This function is composed of relational operators ("*in*"), definite determiners ("*the*"), and noun phrases describing properties of objects. We adopt much of the notation from [6, 5, 155, 135] for our lambda-calculus functions. However, we eschew the neo-Davidsonian event semantics used by Artzi and Zettlemoyer [6], as our tasks are purely represented by state configurations. As such, we deliberately avoid modeling components of trajectories in our semantic representations, which preserves our efficient separation of language and planning components of the system, at a cost to the expressiveness of our semantic representation. Rather than implementing our CCG semantic representations as database queries [155] or trajectory specifications in a custom-built navigation domain [6], we implement our lambda-calculus functions as operating on objects in OO-MDP states.

We primarily model five components of natural language necessary to completing our tasks:

Nouns

We model nouns as single-argument lambda-calculus functions that map OO-MDP objects in a given state to Boolean values. For example, the phrase "block" would be represented as a function $\lambda x.block(x)$. When this phrase is evaluated in an OO-MDP state on a particular object o, it returns true if o is a block.

Adjectives

Adjectival language, such as "green block," are modeled as conjunctions of these single-argument proposition functions. The function $\lambda x.green(x) \wedge block(x)$ checks two attributes of the object provided as its argument. As an OO-MDP object is parameterized using object classes and attributes, these single-argument proposition functions can be implemented as lookup operations on object instances.

Definite Determiners

We model the definite determiner "the" as a function that maps a proposition function to an object that satisfies the given proposition function, following Artzi and Zettlemoyer [6]. This can be represented as a search over a set of objects in an OO-MDP state. We note that definite determiners are evaluated with respect to the initial state of the task, as we assume object references should be resolved at the world-state in which the natural language command was issued.

Comparators

Comparators, such as "the biggest" or "the smallest," are modeled as $argmax$ and $argmin$ operators over the entire space of objects with respect to a proposition function over objects and a numerical property of objects. The operator searches over all objects that satisfy the provided proposition function with respect to the comparison being made, and chooses the object that maximizes the numerical property. In our model, we only provided the argmax with a $size$ operator, although in principle the operator could be any single-argument function that returns a number from an object. Comparators, like definite determiners, are also resolved with respect to the initial world-state.

Relations

Spatial relationships between two objects are modeled as lambda-calculus functions of arity two. The function $\lambda x.\lambda y.in(x, y)$, for example, uses the spatial dimensions of the object provided as the second argument to determine if it contains the first argument. All relationship proposition functions produce Boolean outputs when evaluated in a given OO-MDP state. Currently, we model four spatial relationships: containment (in), adjacency (near), and directional adjacency to the right or left of the referent. All lambda-calculus functions were implemented as pre-specified JScheme [2] predicates that operate on states and objects in the BURLAP [97] reinforcement learning library.

6.3.2 Parser Learning

To map natural language to elements of our semantic representation, we learn a weighted linear CCG parser [30] that maps natural language commands $x \in X$ to a logical form $y \in Y$ in our semantic representation. We collect a data set where each element is of the form (x_i, S_i), where $x_i \in X$ is a natural language command, and S_i is a set of pairs of MDP states, as described in Section 6.4. To learn this parser without providing semantic representations as annotations directly, we define a validation function that determines if a given semantic parse will produce the correct behavior as described by our training data. We additionally use a modified form of the coarse lexical generation procedure of Artzi and Zettlemoyer [6] to produce new CCG lexical entries from training data. To learn parser weights, we use the validation-driven perceptron learning algorithm of Artzi and Zettlemoyer [6] as implemented in the Cornell Semantic Parsing Framework (SPF) [4].

Parser

Our learning objective is to learn a set of parser weights $\theta \in R^d$ for a weighted linear CCG parser [30] with a d-dimensional feature representation $\Phi(x, y)$. This parser uses a variant of the dynamic-programming CKY algorithm to produce the highest scoring parse $\hat{y}$ from a natural language command x. To perform training and inference, we use the linear CCG parser implemented in the Cornell SPF [4]. We added additional features to our model to induce correct behavior in situations where similar language describes different tasks depending on context. For example, the phrase "go to" either implies a containment or adjacency relationship depending on the object being referred to. Earlier iterations of the parser commonly confused our *in* and *near* predicates. To resolve this, we added features recording pairwise appearances of specific predicates.

Parse Validation

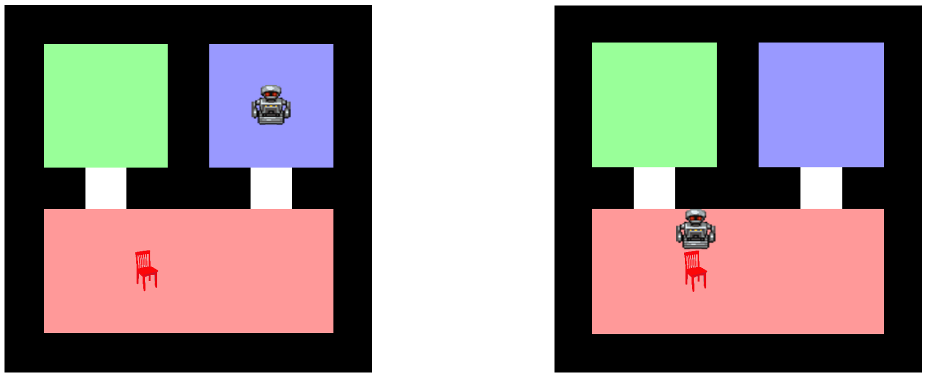
To facilitate our validation-driven perceptron learning, we define a parse validation function similar to those of Artzi and Zettlemoyer [6]. We define the function $\mathcal{V}(y, S) \in \{0, 1\}$, which takes as input a semantic parse y and set of pre- and post- condition state pairs $S = (S_i, S_f)$.

$$\mathcal{V}(y, S) = \begin{cases} 1, & y(S_i) = 0 \land y(S_f) = 1 \\ 0, & \text{otherwise.} \end{cases}$$

We validate over all state pairs provided with a given natural language command. This function ensures that the semantic parse y correctly defines the desired set of goal states. As our tasks define sets of goal states, this is sufficient to check if a parse is valid without invoking a planner. For comparison, we tested against a baseline approach that executed planning with proposed proposition functions during validation, in Section 6.5.

Coarse Lexical Generation

To generate new lexical entries for words and phrases not present in the seed lexicon, we adapt the coarse lexical generation algorithm of Artzi and Zettlemoyer [6] to our



(a) State at task initiation (b) State at task completion

Figure 6.2: The figure shows an example pair used to collect data. Here we ask the users to give a command to the robot that will result in the pre- and post condition behavior shown in the pair of images.

validation procedure. The algorithm generates new proposed lexicon entries from all possible combinations of factored lexical entries (see Kwiatkowski et al. [87] for a detailed description of the lexicon) and words in a given training examples, then discards entries leading to parses that fail to validate.

6.4 Data Collection

Our complete dataset contains 23 tasks, which include moving to any of the three rooms present in our domain, navigating to and relocating objects in the domain, all of which may be specified using room and object attributes, comparators (“the biggest room”), or in reference to relationships between objects in our domain (“go the room that the block is in”). Each task describes a set of goal states that the user desires the agent in our domain to reach. For each task, we generated five pre- and post- condition state pairs (Figure 6.2) using our simulator.

We then gathered training data using using the Amazon Mechanical Turk (AMT) platform. Users were shown three pairs of pre- and post- condition states, sampled from the total five, all representing the same task in different domain configurations.

Example Sentences Collected from AMT
“Move to the green room”
“Go by the red chair”
“Move to stand next to the chair”
“Move close to the chair”
“Stand in the blue room”

Table 6.1: Example sentences collected from the AMT HIT where the users were shown a set of pre-and post condition states and asked to give a command that would instruct the robot to complete the task.

We chose the number of pairs to minimize cognitive load while maximizing the generalizability of the produced language. They were then asked to provide a single command that would instruct the robot to complete the task in every domain configuration. This provided multiple pre- and post- condition pairs for each training example, to incentivize both the learning algorithm and AMT users to produce outputs that are task- rather than configuration-specific. Some example commands gathered from AMT are shown in Table 6.1. We collected a total of 2211 commands.

Many commands received from AMT users contained incorrect commands, many of which implied that the users only examined one of our three pairs of pre- and post-condition states. Some commands were vague and did not describe any specific task, while others only applied to the first pair of images shown to users. Including these commands in our data set actively misleads our parser, encouraging it to produce incorrect parses. We removed 174 such commands from the dataset to produce a pruned dataset.

6.5 Experiments and Results

We performed three sets of experiments on our corpus, collected as described in Section 6.4. First, we trained and tested our model on the full corpus collected from AMT, and a subset of the corpus manually pruned of incorrect and misleading commands. In the second experiment we tested our method against a baseline method that executes a planner during training, similar to earlier weakly supervised CCG parser learning approaches [6, 7]. This experiment was performed with a smaller training corpus, to allow planners with long horizons to validate the parser learning in a reasonable time-frame. For our third experiment, we reduced the level of supervision (that is, the number of demonstration state pairs,) available during learning, while leaving the test data unchanged.

Initialization of the parser learner

We provided the learning algorithm with a seed lexicon as demonstrated in Artzi and Zettlemoyer [6], Artzi et al. [7], Zettlemoyer and Collins [155]. The seed lexicon is used to initialize the coarse lexical generation procedure described in Section 6.3.2. This seed lexicon contained words and phrases paired with syntactic categories and the lambda-calculus meaning representations elaborated upon in Section 6.3.1 This seed lexicon contained simple noun phrases and adjectives such as “chair” and “red,”, comparators such as “the largest”, and imperatives such as “move to” or “go to,” represented with corresponding lambda-calculus logical forms. In addition to using these lexical entries during parsing, the coarse lexical generation procedure used during learning produces new lexical entries from words found in the training data and meaning representations in the seed lexicon. We used a beam width of 75 for lexicon generation during training, which was performed over 15 epochs. At both training and test time, the CCG parser used a beam width of 150.

6.5.1 Evaluation on AMT Corpus

We performed training on two permutations of the data set collected from AMT: the raw dataset, including many incorrect and misleading commands as described in Section 6.4, and a pruned dataset with incorrect commands manually removed. Testing was

Training Set	Demos/Cmd	Best F1
Raw AMT	3	0.64
Pruned	3	**0.82**
Pruned	2	0.75
Pruned	1	0.71

Table 6.2: F1 score on held-out test data, using varying levels of supervision and dataset quality.

performed by validating against state pairs provided with the held-out test data. The full dataset contained 1833 training commands and 678 test commands, each paired with 3 pre- and post- condition demonstration state pairs. The pruned dataset contained 1619 training commands and 418 test commands, each paired with 3 demonstrations as well. We observed a F1 score of 0.64 on the held-out test data of the raw dataset, and expected it to increase when errors were removed from the dataset as described in Section 6.4. This pattern was observed, with an F1 score of 0.82 on the pruned dataset.

6.5.2 Ablation Experiments

To test our parser's robustness to ambiguity in the demonstrations provided during learning, we performed learning with varying levels of supervision and tested against the same held-out test corpus. As our full corpus contains three pre- and post- condition state pairs as demonstration for each command, to incentivize the learner to produce compact and accurate proposition functions, we trained models with one and two commands provided as supervision and compared their performance. We hypothesized that providing fewer states as supervision would render the parser vulnerable to ambiguities in demonstrations. For example, if a provided command is "go near the block" but the learner is only provided with a demonstration in which the agent moves near the block but into the blue room, the proposition function $in(the(\lambda x.agent(x)), the(\lambda y.blue(y) \wedge y.room(y))$ would validate as correct, leading the parser to develop incorrect lexical entries and model weights. We observed that performance relative to the model pro-

Planning horizon	Time taken to learn in s	F1 score
1 step	1201953 ($\sim$ 20 mins)	0.0
10 steps	24024701 ($\sim$ 6.67 hours)	0.667
20 steps	47597724 ($\sim$ 13.22 hours)	0.667
Our method (no planning)	87.18	0.72

Table 6.3: Timing results and F1 scores on a dataset of a train and test split of 50 and 20 commands. The planning based parsing baseline spends most of the learning time computing plans for incorrect parses while learning to parse. The planning based methods spend more time learning and perform poorly when compared to our method.

vided with three demonstrations decreased progressively when demonstrations were removed, as shown in the final two rows of Table 6.2.

6.5.3 Baselines

To compare against a planning-driven baseline, we created a dataset of 50 training and 20 test commands, containing 3 behaviors from our dataset. We produced full demonstration trajectories for these commands. We created the small dataset due to time constraints, as baseline methods take can upwards of 15 hours to run on a 50-command dataset. Much of the decrease in speed comes from attempting to validate invalid parses produced during training. The baseline method uses a planning based validation similar to that used in Artzi and Zettlemoyer [6]. In our baseline, plans are generated for every logical expression constructed and matched to demonstration trajectories in the training data. Parses are valid if the final state in the trajectory produced by the planner and the training trajectory satisfy the logical expression generated by the parser. Our method uses the first and final states of trajectories for parse validation during training, circumventing the repeated planning problem by ensuring that the logical expressions generated satisfy the terminal state of the dataset, without any planning. Using goal-state reward functions as task representations enables this simplification and the corresponding gain in efficiency.

The planning times can be made arbitrarily costly by increasing the horizon of planning, we present results for three different horizons, $1, 10, 15$, all of which take more time and preform worse than our method. The results are shown in Table 6.3. The 1 step planning horizon is insufficient for the agent to reach the goal states in our dataset which can be $\sim 10 - 15$ steps away. However, 10 and 20 step planning horizons find correct parses with a comparable or slightly lower F1 score than our method.

6.6 Robot Demonstration

To interface our parser with a Turtlebot robot, we use reward functions produced by the parser to produce plans in a physical mobile manipulation domain. The domain is a modeled as an MDP after Cleanup World [98]. The configuration of the three rooms can be changed. The states of the robot and the block are being tracked by a motion capture rig. The robot's actions are *forward*, *turn left*, *turn right*, which are executed on the robot as a series of Twist messages over ROS [126]. We show a user asking the robot to get to the right of the block, taking the block to the red room and moving to the largest room, and the robot performing these tasks appropriately. The video of the demonstrations is online[1]

When performing the task "get to the right of the block" the authors expected the agent to circle the block and thus arrive on its right side. However the planner computed that a more efficient plan would be to move the block so the agent reaches the goal of being to the right of the block. This behavior shows that there is more subtlety to language than our semantic representation captures, moving to the right of an object assumes that the object or the landmark has to be stationary. We do not model any landmarks or objects as stationary in our domain or our semantic representation, which can be added in future work. Moreover, this behavior also shows the importance of modeling tasks with MDPs so as to plan optimally.

[1]https://youtu.be/YChlga1wwAc

Next we demonstrate the performance of our parser using limited training data and its ability to extend its model to tasks not seen at training time. We train our parser for two tasks: "go to the green room" and "move next to the block." We provide the parser with 6 sentences and a total of 6 pairs of pre- and post condition states. We collect this data from our Cleanup Domain simulator. Next we learn a parser for this data using our method using the parameters described in Section 6.5. We then use this parser to plan on the robot for the seen tasks of "going to the green room" and "going next to the block". More importantly, the parser successfully executes the unseen task of "going to the blue room." The seed lexicon provided to the parser during training contains the words: "blue", "green" and "red" along with their symbols *blue*, *green* and *red* respectively. The parser learns to infer that the symbols are adjectives used to describe objects and transfers this knowledge to unseen data, allowing the robot to plan for unseen commands with a very small amount of data. This small data set test shows the strength of our method to teach behavior to a robot that is generalizable with a handful annotations.

6.7 Discussion

We modeled goal-based tasks as reward functions within an MDP formalism. Mapping to reward functions allows the robot to plan optimally in real world stochastic environments. However, the tasks presented in this work are a subset of the possible tasks that humans can specify using natural language. For example tasks that specify a trajectory constraint such as "walk carefully along the river" cannot be modeled as a goal state. Mapping language to event based semantic representations such as those of Artzi and Zettlemoyer [6] would allow us to model such language. However, event-based semantic representations requires tracking event history, which weakens the Markov assumption that enable efficient planning.

During data collection users described tasks using nested referential expressions in a way that we did not foresee. For example, users asked the agent to go the room that the chair was in, instead of asking the agent to go near the chair. Our learned parser composed nested referential expressions for these language commands to produce the right behavior while accurately representing the literal meaning of the language. However we noticed that our model had issues disambiguating between *in* and *near* symbols because the language that the users provided was similar when describing the tasks of going into a room and moving close to an object. For example: the phrase “go to” was used to command the robot to both enter a room and move near a block. We introduced new co-occurrence features to capture dependencies between pairs of predicates. This incentivized the parser to learn, for example, that an agent cannot get into a block. This leads to the correct behavior by the agent, but an incorrect representation of the sentences that use nested referential expressions in lambda calculus. This demonstrates that the parser's behavior can be altered in unexpected ways by the addition or removal of features.

Automated generation of reward functions from supervised or unsupervised methods is an important element of teaching behaviors to reinforcement learning or planning agents. We have focused on an approach that uses language and goal states to learn reward functions. In the weakly supervised setting Guu et al. [51] used algorithmic methods to reduce errors caused by ambiguity in demonstrations, which we instead address by providing more demonstrations to the parser to add clarity to the training commands provided. Other approaches have learned reward functions in an unsupervised setting from videos [133]. Future work could combine language and videos in an unsupervised setting.

6.8 Conclusion

This chapter presented a method for learning a parser that maps natural language commands to reward functions using a CCG parser via weak supervision. We showed that this parser can be learned using modifications of existing semantic parser learning algorithms, and its outputs are executable as goal-state reward functions with off the shelf planners.

Such a parser learning method required extensive domain knowledge to set-up the seed lexicon, and the modelling of the language components nouns, adjectives, determiners, comparators and relational operators as compositional functions to make reward functions. This expert knowledge is not easily obtained or transferred to different domains. The parser learning method can also be brittle to the types of language outside the corpus, and produce reward functions that are not interpretable. Neural sequence-to-sequence methods as described in the previous chapter generalize to unseen words in the corpus by using pre-trained embeddings such as GloVe [124]. Further, the sequence-to-sequence methods are easier to setup as they do not require extensive knowledge of the grammar. However, parser based methods learn compositional functions, which is not possible with neural sequence-to-sequence methods. For the next chapter we will not pre-specify any symbols or predicates within the problem domain and learn the predicates and their mappings to language from scratch. This allows us to learn predicates for planning along with language groundings from demonstrations which will be very useful to help train robots with few instances of data.

Chapter 7

Mapping Language to Transferable Symbols for Instruction Following

Large parts of this chapter in the dissertation are currently in submission for review at a robotics conference. This work was performed with Eric Rosen, George D. Konidaris and Stefanie Tellex. The core idea for this project came from me. I wanted to learn language groundings without any symbolic specifications in a continuous state–action space setting with an objective to perform tasks similar in nature to complex LTL expressions and not just goal based commands. Eric helped me with the data collection and the robot demo, and George advised me on skill and symbol learning requirements for this project and Stefanie advised me on the experimental pipeline and type of domains to choose. The entirety of the language grounding, skills and symbol learning pipeline from idea to *execution is my work. This work adds to the previous chapters by learning hierarchical structures and mapping them to language commands with demonstration pairs; a setting that is more realistic in human-robot collaboration scenarios.*

Humans can easily learn novel tasks given paired demonstrations and instructions. For example, we can teach a child how to set a table by showing the task as a demonstration and giving natural language instructions. Studies show that children can learn novel concepts and ground novel words to describe known or unseen objects very quickly; sometimes even with one sample [101]. Robots should also be able to learn these generalizable concepts or symbols or abstractions with few demonstrations. Such learning would allow our robots to learn symbols continuously, and these learned symbols can then be used to plan for novel instructions given by the human user.

Previous end-to-end approaches [3, 107] perform a direct mapping from language to the discrete action space of the agent, and have been shown to be effective in simulations. Some end-to-end approaches [16, 17] map language to learned locations on a simulated map, which is more robust, but still use thousands of demonstrations to learn behaviors. Other methods for instruction following on robots have used handcrafted state abstractions [48, 8] or learned the abstractions first with data outside of the demonstration pairs of language and trajectory [140, 57]. Our work addresses this problem by first learning transferable abstractions or symbols from demonstration trajectories, and then mapping the accompanying language to those symbols. Such an approach allows the agent to be independent of the length of the trajectories or the stochasticity of the environment, and allows us to exploit out-of-the box planning algorithms to find plans to reach goal conditions. Following Konidaris et al. [79], we define a symbol as a set of skill termination states. We learn these termination conditions by first segmenting the demonstration trajectory into underlying skills. Next we cluster the termination states of these skills to create a termination condition for these skills.

These termination conditions are symbols we will use to ground language to plans. The symbols themselves can be considered as termination conditions for an option, with or without a policy. Moreover, these symbols are learned in the agent's frame of reference and not a global frame. This allows transfer as the symbols learned are lifted and not grounded to just a location on a map.

At runtime, we map from language to these learned symbols. Then the robot can use an off-the-shelf planner combined with the options to find a policy to execute the command by chaining options together, mediated by the symbols. Because the symbols are in the agent's frame of reference, they generalize beyond the environments seen during training. We then ground natural language to these learned symbols to plan for the desired behavior.

We implement our approach of instruction following in three domains: TurtleBot simulator, a driving domain using a high dimensional LIDAR dataset collected on a car [147], and navigation on a mobile robot in an indoor scenario. We present the accuracy results of grounding natural language to our learned symbols in the first two domains. We then demonstrate our system end-to-end, with language grounding accuracy results and planning on a real mobile robot domain. These behaviors are learned using a handful of trajectory demonstrations paired with natural language and can generalize to novel maps. To the best of our knowledge this is the first work that learns symbolic abstractions and their mappings to language from demonstrations.

7.1 Related Work

Instruction following is a supervised learning problem where the agent needs to predict trajectory that would satisfy an input natural language command. The agent is trained using instructions paired with valid trajectories for the instructions. Semantic parsing was initially used to solve instruction following problems [99, 6]. This requires goal conditions and sub-goal conditions to be pre-specified. For example, if an agent was asked to "go to the chair," the agent would need a pre-specified goal condition that returns true when the agent is next to the chair. These hand specified goal conditions are labor intensive to design especially on real robots because they need to map from high-dimensional data from sensors to a symbol corresponding to being near the chair.

Some progress was made in learning symbols from data for instruction following by collecting the data for the symbols separately from the trajectories [140, 57]. MacGlashan et al. [98] learned mapping from language to a reward function learned via Inverse Reinforcement learning. However the abstractions such as objects and rooms were pre-specified in their domains, and not learned from scratch, and the reward function itself considered the permutation of these objects in the world. Matuszek et al. [103] learned mappings from sentences to attributes learned from sensor data directly. This work is the closest in spirit to what we are trying to achieve. However instead of learning object attributes we are learning abstractions for planning from scratch and grounding language to these learned abstractions.

Some works have tried to use neural approaches for sequence to sequence mapping from language directly to trajectories [3, 107]. Blukis et al. [16] map language to learned locations on a simulated map using an end to end approach, which is a robust approach to language grounding to goals. However, their methods still requires thousands of interactions with the agent as they learn the policy end to end. We believe that our approach would require fewer demonstrations and would generalize better. Other end-to-end learning methods have used reinforcement leaning [55, 25], however they use beyond millions episodes to learn simple behaviors. There are also approaches that map language to pre-specified symbols using neural networks [48]. Mapping to symbols in these approaches allows robots to plan more robustly to sensor noise and environment stochasticity. However, the symbol spaces must be pre-defined with expert knowledge. For these mappings they translate natural language to the pre-specified symbol space using neural translation approaches like Seq2Seq [136], or Transformers [143]. In this work instead we want to learn the symbols or abstraction from the trajectories and then map language to these learned symbols.

There have been many approaches to learn skills from trajectories [117, 75]. Konidaris et al. [79] learns symbols from these skills. These skills have been seen as a way to help agents learn in a reinforcement learning setting [77] or allow the robot to follow a demonstration with multiple learned skills [117]. We follow a supervised learning approach to learning skills with demonstrations as demonstrated in Niekum et al. [117] and Konidaris et al. [75]. We convert these skills to symbols and plan over them as described by Konidaris et al. [79]. We differ from the previous approaches as the ordering of our skill terminations is defined by natural language at runtime.

7.2 Problem Definition

The inputs to this system consists of multiple egocentric trajectories ($\{\tau\}_{i=0}^{N}$) paired with natural language instructions for each trajectory $\{I\}_{i=0}^{N}$. Each trajectory is a sequence of states observed when an agent performs and action, that is, $\tau = \{(s_0, a_0), (s_1, a_1)....(s_T, a_T)\}$. Each trajectory is paired with an instruction I_n given by a human user.

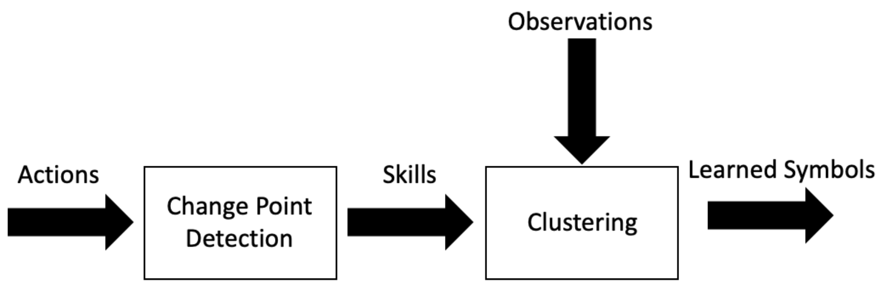
After training, the outputs of the system are a set of symbols $\sigma_i \in \Sigma$ which can be used for planning in the world, and a mapping from natural language to a sequence of these symbols. At runtime, the robot is placed in a novel environment and given a natural language command it has not previously encountered. It uses its trained model to map from the language to a sequence of symbols. Then, the robot plans to reach each symbol in the output sequence in order until the entire symbol sequence has been executed. In this paper the state space consists of LIDAR data that the robot observes in the world.

7.3 Mapping Language to Plans via Learned Symbolic Abstractions

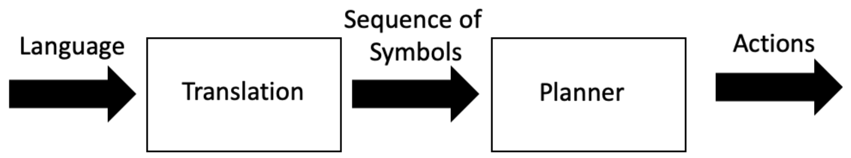
Our approach requires robot trajectories, in the form of actions and observations in egocentric frame, as well as paired natural language commands during learning. During learning we learn an abstraction of the world in the form of symbols. These symbols are the termination conditions of the underlying skills that create the demonstration trajectories. To learn symbols we first recognize these skills using change point detection. We then segment demonstrations into their underlying skills, and then learn skill termination conditions using clustering and classification. For instruction following we translate natural language to a sequence of these symbols, that is skill terminations, which can then be planned over. During runtime we plan over the learned symbols to achieve desired behavior (Fig. 7.1). We now discuss different parts of our approach.

7.3.1 Symbols

A symbol σ is a classifier to identify the termination set of a particular skill. We assume that the complex behaviours demonstrated by the demonstration trajectories can be broken down into reusable skills. Hence a trajectory demonstration is an ordering of skill termination conditions, or symbols. We map natural language to the sequence of symbols present in the demonstration trajectories. We need to learn only the termination conditions of the skills, which will be referred to as symbols in this paper. The skill policy can be learned from the segmented demonstration using a framework like Dynamic Movement Primitives (DMPs) [60], or we can plan to skill terminations if the transition model of the world is known. These symbols can also be thought of as options termination conditions, however since we are using the labels of these option termination conditions for translation, we refer to them as symbols.



(a)



(b)

Figure 7.1: The pipelines for our methodology for learning symbols and grounding language to them for instruction following: a) To learn symbols, actions are segmented using change point detection to generate skills. Then the observations that occur at skill termination are clustered to generate our learned symbols; b) At run time, natural language is inputted to our translation model, which outputs a sequence of learned symbols. The outputted symbols are put into a planner, which outputs robot actions. Note that the translation model is trained by using language and learned symbol pairs.

7.3.2 Egocentric Representation

The classifier of each symbol is trained with states observed in egocentric space, but these symbols can be grounded in the global frame of reference by checking if the egocentric state of a location in the map or pose of the robot is similar to the egocentric state observed during learning.

The egocentric representation in our case has states in the form of Light Detection and Ranging (LIDAR) data, and the actions in the form of torque input to the agent. For other platforms the state can be an ego-centric camera input. The agent-space is non-Markov in nature as multiple states might look the same to the agent. For example, in the LIDAR setting, in a long corridor the state space of the LIDAR might hardly change as the agent is moving forward. Since multiple states look the same in agent-space, it is easier to learn symbols in this space that can then be transferred to different locations within a map that look the same.

There is a separate global frame of reference [77], which is the state of the external world under the given instruction. In this state space there is a map of the world, and the robot's cardinal location is known in the map. This state is Markov, and allows planning. We learn the abstract symbols in local egocentric frame, and plan in a global Markov state space, which in this case is a map. We ground symbols learned in agent-space onto the physical map of the world in problem-space and then perform planning over them. Previously, James et al. [63] have learned symbols that can be ported and generalized to different locations within a map with some global frame variables, like location on the map, as part of the agent's state. This allows the instruction giver to use sentences like the "go through the door on the right." We do not learn portable symbols in this work, but know that portability of the learned symbols is an important problem. For this work we assume the transition dynamics of the world to be known to allow point to point navigation on the map.

7.3.3 Change Point Detection

Change point detection is used to detect a change in the underlying statistics of a time series. We specifically use a hierarchical Bayesian formalism for change point detection so as to detect repeated instances of the same latent skill. We assume that our trajectory demonstrations are unstructured, but have underlying latent skills that are repeated across multiple demonstrations.

We choose Hierarchical Dirichlet Process - Hidden Markov Model (HDP-HMM) [139] to model our trajectories. This formulation gives us the advantage of not only recognizing the change points themselves, but also provides mapping between trajectory segments and the underlying latent skills, allowing us to recognize when the same skill is being repeated across multiple demonstrations. Such an approach has been used previously by Niekum et al. [117], Ranchod et al. [129] and Konidaris et al. [75].

The data in our case consists of the observed actions $y = [y_0, y_1, ...y_{T-1}, y_T]$ chosen by the demonstrator, where the data is d dimensional. There are underlying latent skills which come from $k \in \{1, 2, ...\}$ possible labels. Under the HDP-HMM process latent skills are assumed to be generated from a first order Markov process represented by $z = \{z_0, z_1, ...z_T\}$. The latent first order Markov process is parameterized with initial state probabilities (π_0) and transition probabilities $(\{\pi_k\}_{k=1}^{\infty})$ from one latent skill to another. There is an additional parameter of $(\{\theta_k\}_{k=1}^{\infty})$ to govern the observation noise when observing the output trajectory, which is parameterzied by the hyperparameter λ. The mathematical formulation for a HDP-HMM as described by Fox et al. [44] is given by:

$$y_t | z_t = k \sim \mathcal{N}(x_t | A_k x_{t-1}, \Sigma_k); \qquad z_{t+1} | \{\pi_k\}_{k=1}^{\infty}, z_t \sim \pi_{z_t};$$

$$\pi_k | \alpha, \kappa, \beta \sim \text{DP}(\alpha + \kappa, (\alpha\beta + \kappa\delta_\kappa)/(\alpha + \kappa)); \qquad \beta | \gamma \sim \text{GEM}(\gamma),$$

where DP is a Dirichlet process and β is a random probability measure setting the

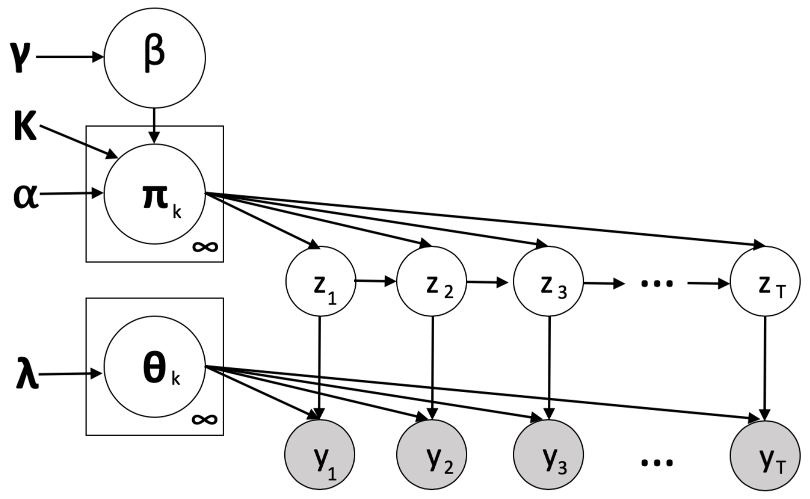


Figure 7.2: The HDP-HMM plate model used in our work. Z indicates latent states, in our case the skills, Y indicates observations, in our case the actions, π indicates initial and transition probabilities between latent states.

transition probabilities between skills. GEM refers to Griffiths, Engen and McCloskey distribution, which is used to generate β using a stick breaking model parameterized by γ. Parameter β in turn parameterized the transition probabilities π_k along with the hyperparameters of α and κ. The goal of such a model is to estimate the joint probability $Pr(\beta, \pi, \theta, \{z\}_{t=1}^{T} | \{x\}_{t=0}^{T})$, which is the joint probability of the DP parameters and hidden states given the trajectory. The estimation of the joint probability is generally performed using variational inference. The Viterbi algorithm [43] is then used to label each time step with a latent skill using the joint probability distribution. The observed sequence y here is treated as an auto-regressive process. For a more complete mathematical treatment of HDP-HMM please refer to Fox et al. [44] and Hughes [58]. The plate model for the HDP-HMM model is shown in Figure 7.2. We used the BNPY toolbox by Hughes and Sudderth [59] for performing this HDP-HMM change point detection. HDP-HMM labels the trajectory segments with the underlying latent skills. These trajectory segments can be used to learn a skill policy using DMPs[60].

In this work we are only segmenting the action trajectory of the agent, and not the state-action trajectory. This is different from methods demonstrated in Konidaris et al.

[75] and Niekum et al. [117], where state and action information is used for segmentation. This is because our state data is not just the robot's joint space, but the actual high dimensional state of the environment visible to the agent, which computationally intractable with the current state of the art Bayesian changepoint detection methods.

7.3.4 Clustering

Each of the latent skill learned with changepoint detection could potentially have multiple termination conditions. We use clustering as an unsupervised method to learn possible termination sets of each learned skill. More precisely, the number of effect sets of a skill are not known in advance. For example, the going forward skill might terminate when the agent reaches a wall, or when the agent reaches the next room. The termination sets for both of these skills must be modelled separately. However the trainer need not know how many such effect sets exist, hence we use clustering to recover possible termination sets. We use DBSCAN [40] as the clustering method. We prefer it as it is a non-parametric method that does not need a pre-specified number of clusters or the diameters of possible clusters. We only specify a distance metric, a minimum number of support points for a cluster, and a noise parameter. The noise parameter allows the algorithm to identify outliers. We choose the same parameter for all skills in a domain, that we pick such that we do not have any small clusters of termination states for any skill.

7.3.5 Classification

We then take each cluster and create a classifier with the points associated with the cluster. We choose Single Class SVMs [131] as they are a robust classifier against noise, which is prevalent in LIDAR data. Single Class SVMs were developed as an outlier detection method, and in our case, outlier states are those that do not satisfy the statistics of our termination states. The classifiers output a true label if the tested points can belong to the cluster of points used to train it, and a false label if the it is more likely that the points are outliers. We use a sigmoid kernel as features for classification, as it helps distinguish LIDAR distance data well. We choose best practice methods to choose hyperparameters for the classifier, which are dependent on the number of dimensions of the data and the variance of the data. After we train classifiers associated with each cluster, we attempt to merge classifiers that have the same output response for each skill. This is likely as the clustering process is unsupervised and produces more clusters that are indistinguishable from each other. To compare the output responses of the classifiers we classify the points used to train one classifier using the other classifier, and vice-versa. If both classifiers label more than 85% of the points belonging to the other cluster as inliers, we then merge the cluster, as their inlier responses are similar, and the clusters have a large overlap, hence they might as well be the same symbol.

7.3.6 Translation

Machine translation is posed as mapping an input sequence to an output sequence. For our problem, we aim to learn a mapping from natural language instructions, defined as a sequence of strings $i = [i_0, i_1, ..., i_{l-1}, i_l]$, to a sequence of symbols, that is, termination conditions $[\sigma_0, .., \sigma_n]$.

The output target sentence is the order in which the symbols appear in the demonstration trajectory. We then train the deep Seq2Seq recurrent neural network [136] to take the sequences and output the paired sequence of skills using the negative log likelihood loss. The embedding vector size is 50 dimensional, and the hidden vector size is 256. Seq2Seq utilizes two recurrent neural networks, one which acts as an encoder E and the other as a decoder D. The encoder is trained to take the input sequence and map it to a vector of fixed dimension, and the decoder is trained to take the output and decode to the translated sequence. Our Seq2Seq model has Badhanau attention [10] to provide soft alignment between input and output symbol sequences. We also use GloVe [124] pre-trained embeddings as they improve Seq2Seq performance in low data scenarios.

At runtime we plan over the sequence of output symbols predicted by the Seq2Seq model in order, that is, we plan for the first symbol in the sequence, and then the next and so on until the terminal symbol. The policy for each symbol as mentioned previously can be a DMP or an out of the box planner.

7.4 Experiment

In order to evaluate our algorithm, we tested our method on three different robot datasets: A simulated Turtlebot domain, a real-world car LIDAR dataset collected from the Naval Postgraduate School (NPS) [147], and a robotics dataset we collected ourselves with a Kinova Movo mobile manipulator going through an office building.

7.4.1 Simulated Turtlebot

We created the domain of Corridor world with corridors and intersections, so as to resemble an office building. This domain was created in Gazebo, with friction and other sources of noise. The Turtlebot agent has a simulated noisy Hokuyo LIDAR, which provides the perceivable state space information of the agent. The simulated Turtlebot functions at 30 Hz, and actions were sent to the robot with Twist messages. The action data was a 3 dimensional encoding the twist velocity commands of up and left and right.

We collected a total of 26 trajectories paired with 26 language annotations, one language annotation per demonstration, collected from an expert user. These trajectories consisted of $5 - 6$ demonstrations each, for 5 different behaviours of: turn right, turn left, go straight until end of corridor, go straight and turn left at the intersection, and go straight and turn right at the intersection. We first segmented the skills and produced the output the order of skills for every demonstration.

Different trajectory demonstrations get segmented into skills as shown in Fig. 7.3c. The Hokuyo LIDAR on the robot returned ranges for the frontward facing 270 degrees of the Turtlebot. We then clustered the termination states of these skills to create symbols. We used this order of these symbols as our target language for translation. We then took the terminating LIDAR states from each of the segmented skills, and inputted them into our clustering algorithm which produced 4 symbols: intersection, end of corridor, left, and right. The DBSCAN hyperparameters used for clustering were eps$= 15$ and minimum support $= 5$. These 4 clusters produce a classifier each as the symbols.

We then used the input language instructions as the source language for translation, and the order of skills from the segment trajectories as the output target language. We trained our network 5 times, and ran 5-fold cross validation for testing each time. We observed an average cross-validation result of $52.8\% \pm 4.57$. The random baseline for this test is 20% as there are 5 output behaviours trained.

There are two causes for these low numbers. First is the lack of training language data which hurts when an out of vocabulary word is used. Secondly we have multiple behaviours mapping to words like ”right” and ”left,” cause the accuracy to drop. We crowd-source more language data for the other experiments.

7.4.2 NPS car dataset

The NPS car dataset [147] consists of a car driving around a real neighborhood. The car has a 32-Laser Velodyne HDL-32e LIDAR and three Logitech c920 HD RGB cameras. The Velodyne functions at 10 Hz and is collected from a full 360 degree view with a minimum distance of 0.9 meters and a maximum range of 130 meters. The RGB cameras function at 30 Hz, and have a full-HD resolution of $1920x1080$. We used this dataset because it had a lot of urban driving with plenty of turns, instead of highway driving.

However, no action or IMU data was recorded in this dataset, and hence we used the Loam Velodyne [1] ROS package to perform mapping and state-estimation of the car from the Velodyne data. We then used the difference between sequential poses to compute the magnitudes and directions at each time step, which we treated as our 2 dimensional action data for going forward and turning along the vertical axis.

We collected a total of 10 trajectories, and then uploaded the frontward facing RGB video of these trajectories to Amazon Mechanical Turk (AMT), and requested users to write down one sentence they would use to command the robot to perform the observed behavior. From this, we collected 300 language annotations for 3 different behaviors: turn left after getting to the intersection, turn right after getting to the intersection, and going straight until the intersection. We cleaned the language dataset to 270 examples as the other 30 sentences were not related to the task at all. We segmented the skills, and then used the order of these skills as our target language for translation. For symbol learning, we converted the Velodyne data into a 1-D 360 that we use as our input states. We then took the terminating laser scans from each skill segment and clustered them using DBSCAN. This produced 3 symbols for the terminations of: turning left, turning right, and going straight. The clustering hyperparameters used for DBSCAN were $\epsilon = 20$ and minimum support $= 10$. These 3 clusters produce 3 classifiers as symbols.

We then used the crowd-sourced language instructions as our source language for translation, and the output sequence of symbols from the trajectory segmented as our target language. We trained our network 5 times, and ran 5-fold cross validation for testing each time. We observed an average cross-validating result of 96.51 ± 0.46. The random baseline for this test is 33% because there are 3 potential output behaviours trained.

Our behaviors were short in length, which made learning a mapping from language easy. We would have preferred to learn longer horizon behaviors however our mapping and state-estimation package accumulated errors over longer periods of time, limiting our train and test behaviors to be relatively small to our other experiments. This would not have been an issue if we had access to high-quality IMU or action data. Note that we did not perform planning in this domain; we only learned symbols with clustering and learned a mapping from language to these symbols.

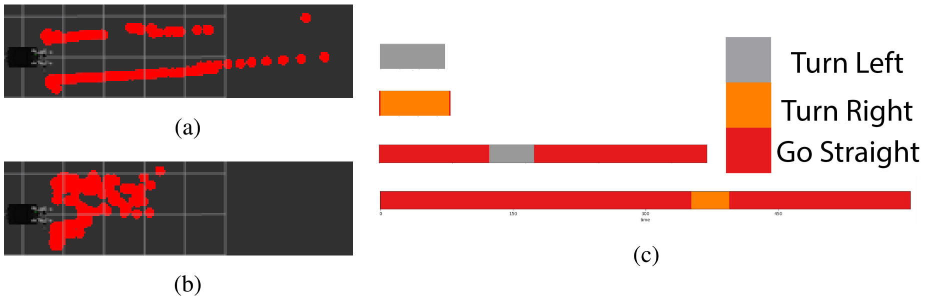


Figure 7.3: Example images of our learned symbols (mobile robot domain) and skill segments (simulated Turtlebot): a) Median of the states used to learn the open intersection symbol; b) Median of the states used to learn the end of corridor symbol; c) Skill segmentation for the behaviors of turn left, turn right, go to the intersection and turn left, and go to the intersection and turn right demonstrated on the Turtlebot in the simulated world of Corridor world.

7.4.3 Mobile Robot domain

We wanted to learn symbols on a real mobile robot, map language to it, and perform planning in an unseen environment. For this, we used a Kinova Movo robot with a built in LIDAR sensor and frontward facing RGB camera. The LIDAR sensor operates at 15 Hz, and actions were sent to the robot with Twist messages at 45 Hz. The action data was 2 dimensional encoding the linear x velocity and the angular z velocity.

For training, we went to 5 different locations and collected a total for 75 trajectories. For language collection, we uploaded the robot frontward facing RGB camera videos to AMT, and requested users to write down one sentence they would say to instruct the robot to perform the observed behavior. From this, we collected 913 language annotations for 5 different behaviors of going straight and taking left, going straight and taking right, in place holonomic left, in place holonomic right and going to the intersection. In the context of segmentation, the trajectories are noisy because the algorithm labels even small left and right movements as left and right turning skills. Hence we remove any latent skill that is shorter than half a second, as these movements might just involve course correction while providing demonstrations. The algorithm produced 3 different latent skills of turning left, turning right and going straight. The output now is a sequence of skills per demonstration trajectory.

For symbol learning, we filtered the LIDAR to only use the frontward facing $90°$ data, as the corridors were narrow and the LIDAR returned a lot of noise from the side sensors. We then took the terminal LIDAR states from each of the 3 segmented skills, and input them into our clustering algorithm, DBSCAN, which produced 5 different clusters with hyperparameters of $\epsilon = 88$ and minimum support $= 5$. We learned one class SVMs using states in each of these clusters representing end of corridor, intersection, right terminal corridor, and two left terminal corridor symbols. Example images of our learned symbols can be seen in Fig. 7.3a, 7.3b.

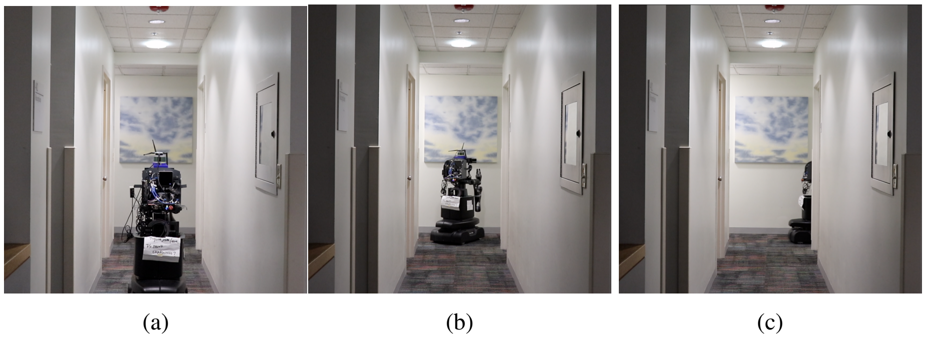

(a) (b) (c)

Figure 7.4: Images of mobile robot planning in unseen environment from the command "Go to the end of the corridor and take a right": a) robot going to intersection; b) robot taking a right; c) robot going to end of corridor

Using our trajectory and language dataset, we trained our language translation model on 5000 epochs, with 50 dimensional pre-trained GloVe embedding on the encoder side, with hidden state dimension being 256. The translation accuracy was $75.71 \pm 0.37\%$. A baseline random policy has translation accuracy 20%. We then brought the robot to an unseen environment, and tested three different language commands: “go to the end of the corridor and take a right”, “left at the end of the hallway”, and “go to the end of the corridor”. In all three cases, our translation model produced both a valid sequence of symbols for the given map, as well as the necessary symbols for accomplishing the instruction. Images of the robot performing planning for one of the behaviors can be seen in Figure. 7.4, and videos of all three behaviors in the test environment can be found in our supplemental video[1].

The translation accuracy is not as high as we would like it to be as there are multiple symbols that recognize when the robot has turned left. This is because we collected only 10 trajectories of the robot turning left. In half the instances the robot landed in an open corridor and in other it the robot was in a corridor which was not very long. Due to the large disparity in depth between the corridors we trained on, the terminal states for the left skill was bimodal and produced two separate symbols that the clustering algorithm was not able to merge together. We believe this discrepancy caused our model to under-perform.

This creates two symbols for left that cannot be merged as the robot should see these corridors differently in the world. We were able to merge our symbols for turning right as the agent had over 20 trajectories to learn the meaning of right, and by sampling different corridors the agent was able to generalize the symbol termination across different depths. We believe a much better method for translating from natural language to a set of learned symbols might be via inference, or conditioning the translations on the map of the world that the robot is encountering. In such cases it would be easy for the agent to establish that the current map has only certain types of symbols, and therefore the left symbol and the grounding process would be straightforward.

[1]`https://streamable.com/zsf59`

7.5 Conclusion

In this work, we show how to automatically learn abstract symbols, map language to them, and transfer the skills to new environments for planning. This is the first work to demonstrate such an end-to-end pipeline that learns symbols and groundings from language and trajectory demonstration pairs. The fact that our symbols are represented as termination classifiers over robot observations in egocentric frame allows them to be both interpretable and generalizable to unseen environments. Furthermore, by first translating high-dimensional natural language into a drastically smaller space of learned symbols, our method requires very few demonstrations to learn behaviors, which is crucial in robot applications that have large state spaces. Our experiments across three different datasets demonstrate that our method enables real robot platforms to follow open-ended language commands with little data.

Chapter 8

Discussion

In this thesis we presented approaches to ground language to abstractions. To this end we had four specific contributions:

- A fast and reactive hierarchical planner specifically designed for the large state-action spaces encountered in task and motion planning.
- Language grounding for different types of behaviors or plans, from simple goal conditions to richer temporal constraints.
- using different mechanisms for language grounding: simple classification, deep sequence-to-sequence methods, traditional grammar based parser learning.
- Learning goal and sub-goal conditions of a plan from scratch and mapping language to these symbols. This should allows us to learn hierarchical structures helpful for planning via demonstrations.

Initially we grounded language to pre-specified abstractions. While specifying these abstractions is easy for small domains, it is impossible to specify these abstractions for all tasks. We also believe that for language grounding via classification is perhaps an easier solution than most other approaches such as sequence to sequence mappings or learning a parser. However, classification is only suitable for scenarios in which all classes are known a priori, which necessitates learning a new classifier to learn any new groundings. On the other hand we have found sequence to sequence methods too brittle to predict novel combinations of reward functions. Other methods like learning a CCG parser require us to specify a lot of the semantic and grammatical rules, which require a lot of expert domain knowledge, even if these approaches lead to some generalization. Hence we believe more research needs to be performed on learning compositional methods to ground novel translations or reward functions.

Our final result learned both the abstractions and the language grounding to these abstractions from data. This result is a proof of existence for learning abstraction from trajectory data, and mapping language to these sub-goals. Our next goal is to learn more complex tasks perhaps on a pick and place task with a mobile robot. While learning all abstractions from scratch might require many demonstrations, which can make such an approach impractical, we believe that in an ideal scenario plenty of these abstractions are available in advance. These abstractions already exist in computer vision literature in the form of labelled object classifiers and caption generators [86]. We believe that the right solution is a mixture of pre-specified abstractions and the capability of an artificial agent to learn new abstractions on demand.

In the future we would like to quantify the quality of an abstraction hierarchy. We believe that such quantification would help in constructing better hierarchies. Moreover, we would like to introduce inference based planning to allow question asking, and improvement of a pre-specified plan, especially if the pre-specified plan is incorrect or partially specified. Furthermore, we believe the dialog between a human and an agent will play an important role in learning language groundings and object affordances.

Bibliography

[1] Laser Odometry and Mapping (Loam) Velodyne. `http://wiki.ros.org/loam_velodyne`.

[2] Ken R. Anderson, Timothy J. Hickey, and Peter Norvig. The JScheme language and implementation, 2013. URL `http://jscheme.sourceforge.net/jscheme/main.html`.

[3] Jacob Andreas and Dan Klein. Alignment-based compositional semantics for instruction following. In *Conference on Empirical Methods in Natural Language Processing*, pages 1165–1174, 2015.

[4] Yoav Artzi. Cornell SPF: Cornell Semantic Parsing Framework, 2016. URL `https://github.com/lil-lab/spf`.

[5] Yoav Artzi and Luke Zettlemoyer. Bootstrapping semantic parsers from conversations. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 421–432, 2011.

[6] Yoav Artzi and Luke Zettlemoyer. Weakly supervised learning of semantic parsers for mapping instructions to actions. In *Annual Meeting of the Association for Computational Linguistics*, 2013.

[7] Yoav Artzi, Dipanjan Das, and Slav Petrov. Learning compact lexicons for ccg semantic parsing. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pages 1273–1283, October 2014.

[8] Dilip Arumugam, Siddharth Karamcheti, Nakul Gopalan, Lawson L. S. Wong, and Stefanie Tellex. Accurately and efficiently interpreting human-robot instructions of varying granularities. In *Robotics: Science and Systems XIII*, 2017.

[9] Faheim Bacchus, Craig Boutilier, and Adam Grove. Rewarding behaviors. In *National Conference on Artificial Intelligence*, pages 1160–1167, 1996.

[10] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*, 2014.

[11] Bram Bakker, Zoran Zivkovic, and Ben Krose. Hierarchical dynamic programming for robot path planning. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2756–2761, 2005.

[12] Jennifer L. Barry, Leslie Pack Kaelbling, and Tomás Lozano-Pérez. Deth: Approximate hierarchical solution of large markov decision processes. In *International Joint Conference on Artificial Intelligence*, pages 1928–1935, 2011.

[13] R. Bellman. A Markovian decision process. *Indiana University Mathematics Journal*, 6: 679–684, 1957.

[14] Richard Bellman. Dynamic programming and lagrange multipliers. *Proceedings of the National Academy of Sciences*, 42(10):767–769, 1956.

[15] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *Journal of Machine Learning Research*, pages 1137–1155, 2000.

[16] Valts Blukis, Nataly Brukhim, Andrew Bennett, Ross A Knepper, and Yoav Artzi. Following high-level navigation instructions on a simulated quadcopter with imitation learning. pages 505–518, 2018.

[17] Valts Blukis, Dipendra Misra, Ross A Knepper, and Yoav Artzi. Mapping navigation instructions to continuous control actions with position-visitation prediction. *Robotics: Science and Systems XIV*, 2018.

[18] Mario Bollini, Stefanie Tellex, Tyler Thompson, Nicholas Roy, and Daniela Rus. Interpreting and executing recipes with a cooking robot. In *Experimental Robotics*, pages 481–495, 2013.

[19] Adrian Boteanu, Thomas Howard, Jacob Arkin, and Hadas Kress-Gazit. A model for verifiable grounding and execution of complex natural language instructions. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2649–2654, 2016.

[20] Daniel J. Brooks, Constantine Lignos, Cameron Finucane, Mikhail S. Medvedev, Ian Perera, Vasumathi Raman, Hadas Kress-Gazit, Mitch Marcus, and Holly A. Yanco. Make it so: Continuous, flexible natural language interaction with an autonomous robot. In *AAAI Conference on Artificial Intelligence Workshop on Grounding Language for Physical Systems*, 2012.

[21] Peter F. Brown, John Cocke, Stephen Della Pietra, Vincent J. Della Pietra, Frederick Jelinek, John D. Lafferty, Robert L. Mercer, and Paul S. Roossin. A statistical approach to machine translation. *Computational Linguistics*, 16:79–85, 1990.

[22] Peter F. Brown, Stephen Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19:263–311, 1993.

[23] Emma Brunskill and Lihong Li. PAC-inspired option discovery in lifelong reinforcement learning. In *International Conference on Machine Learning*, pages 316–324, 2014.

[24] Bob Carpenter. *Type-Logical Semantics*. MIT Press, 1997.

[25] Devendra Singh Chaplot, Kanthashree Mysore Sathyendra, Rama Kumar Pasumarthi, Dheeraj Rajagopal, and Ruslan Salakhutdinov. Gated-attention architectures for task-oriented language grounding. In *Thirty-Second AAAI Conference on Artificial Intelligence*, pages 2819–2826, 2018.

[26] David L. Chen and Raymond J. Mooney. Learning to interpret natural language navigation instructions from observations. In *AAAI Conference on Artificial Intelligence*, 2011.

[27] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gúlçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Empirical Methods in Natural Language Processing*, 2014.

[28] Istvan Chung, Oron Propp, Matthew R Walter, and Thomas M Howard. On the performance of hierarchical distributed correspondence graphs for efficient symbol grounding of robot instructions. In *International Conference on Intelligent Robots and Systems*, pages 5247–5252, 2015.

[29] Junyoung Chung, Çaglar Gúlçehre, Kyunghyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *Neural Information Processing Systems Workshop on Deep Learning*, 2014.

[30] Stephen Clark and James R Curran. Wide-coverage efficient statistical parsing with ccg and log-linear models. *Computational Linguistics*, pages 493–552, 2007.

[31] L Stephen Coles. Talking with a robot in english. In *IJCAI*, pages 587–596, 1969.

[32] Costas Courcoubetis and Mihalis Yannakakis. The complexity of probabilistic verification. *Journal of the Association for Computing Machinery*, pages 857–907, 1995.

[33] Thomas G. Dieterrich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal on Artificial Intelligence Research*, pages 227–303, 2000.

[34] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, pages 269–271, 1959.

[35] Carlos Diuk, Alexander L Strehl, and Michael L Littman. A hierarchical approach to efficient reinforcement learning in deterministic domains. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 313–319, 2006.

[36] Carlos Diuk, Andre Cohen, and Michael L. Littman. An object-oriented representation for efficient reinforcement learning. In *International Conference on Machine Learning*, pages 240–247, 2008.

[37] Richard O Duda and Peter E Hart. Use of the hough transformation to detect lines and curves in pictures. Technical report, 1971.

[38] Juraj Dzifcak, Matthias Scheutz, Chitta Baral, and Paul Schermerhorn. What to do and how to do it: Translating natural language directives into temporal and dynamic logic representation for goal management and action execution. In *IEEE International Conference on Robotics and Automation*, 2009.

[39] Kutluhan Erol, James A Hendler, and Dana S Nau. Umcp: A sound and complete procedure for hierarchical task-network planning. In *Artificial Intelligence Planning Conference*, 1994.

[40] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *In Proceedings of Knowledge Discovery and Data Mining*, pages 226–231, 1996.

[41] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, pages 189–208, 1971.

[42] Centre for Disease Control and Prevention. Cdc's developmental milestones, 2019. URL `http://web.archive.org/web/20190609063726/https://www.cdc.gov/ncbddd/actearly/pdf/checklists/Checklists-with-Tips_Reader_508.pdf`.

[43] G David Forney. The Viterbi Algorithm. *Proceedings of the IEEE.*

[44] Emily B Fox, Erik B Sudderth, Michael I Jordan, Alan S Willsky, et al. A sticky hdp-hmm with application to speaker diarization. *The Annals of Applied Statistics*, 5(2A): 1020–1056, 2011.

[45] Google. Google Speech API. `https://cloud.google.com/speech/`, 2017. Accessed: 2017-01-30.

[46] Nakul Gopalan, Marie desJardins, Michael L. Littman, James MacGlashan, Shawn Squire, Stefanie Tellex, John Winder, and Lawson L.S. Wong. Planning with abstract

markov decision processes. In *Abstraction in Reinforcement Learning Workshop at ICML*, 2016.

[47] Nakul Gopalan, Marie desJardins, Michael L. Littman, James MacGlashan, Shawn Squire, Stefanie Tellex, Robert John Winder, and Lawson L. S. Wong. Planning with abstract Markov decision processes. In *International Conference on Automated Planning and Scheduling*, pages 480–488, 2017.

[48] Nakul Gopalan, Dilip Arumugam, Lawson L.S. Wong, and Stefanie Tellex. Sequence-to-sequence language grounding of non-markovian task specifications. In *Robotics: Science and Systems*, 2018.

[49] Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

[50] Maria Teresa Guasti. *Language acquisition: The growth of grammar*. MIT press, 2017.

[51] Kelvin Guu, Panupong Pasupat, Evan Zheran Liu, and Percy Liang. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. pages 1051–1062, 2017.

[52] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal aspects of computing*, pages 512–535, 1994.

[53] Stevan Harnad. The symbol grounding problem. *Physica D: Nonlinear Phenomena*, pages 335–346, 1990.

[54] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, pages 100–107, 1968.

[55] Karl Moritz Hermann, Felix Hill, Simon Green, Fumin Wang, Ryan Faulkner, Hubert Soyer, David Szepesvari, Wojciech Marian Czarnecki, Max Jaderberg, Denis Teplyashin, et al. Grounded language learning in a simulated 3d world. *arXiv preprint arXiv:1706.06551*, 2017.

[56] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, pages 1735–1780, 1997.

[57] Thomas M. Howard, Stefanie Tellex, and Nicholas Roy. A natural language planner interface for mobile manipulators. In *IEEE International Conference on Robotics and Automation*, 2014.

[58] Michael C Hughes. Reliable and scalable variational inference for bayesian nonparametric models. 2014.

[59] Michael C Hughes and Erik B Sudderth. Bnpy: Reliable and scalable variational inference for bayesian nonparametric models. In *In Proceedings of the NIPS Probabilistic Programimming Workshop*, pages 8–13, 2014.

[60] Auke Jan Ijspeert, Jun Nakanishi, and Stefan Schaal. Movement imitation with nonlinear dynamical systems in humanoid robots. In *IEEE International Conference on Robotics and Automation*, pages 1398–1403. IEEE, 2002.

[61] Auke Jan Ijspeert, Jun Nakanishi, Heiko Hoffmann, Peter Pastor, and Stefan Schaal. Dynamical movement primitives: learning attractor models for motor behaviors. *Neural computation*, pages 328–373, 2013.

[62] Mohit Iyyer, Varun Manjunatha, Jordan L. Boyd-Graber, and Hal Daumé. Deep unordered composition rivals syntactic methods for text classification. In *Annual Meeting of the Association for Computational Linguistics*, pages 1681–1691, 2015.

[63] Steven James, Benjamin Rosman, and George Konidaris. Learning portable representations for high-level planning. *arXiv preprint arXiv:1905.12006*, 2019.

[64] Michael Janner, Karthik Narasimhan, and Regina Barzilay. Representation learning for grounded spatial reasoning. *arXiv preprint arXiv:1707.03938*, 2017.

[65] Nicholas K. Jong and Peter Stone. Hierarchical model-based reinforcement learning: R-max+ MAXQ. In *International Conference on Machine Learning*, 2008.

[66] Nicholas K Jong, Todd Hester, and Peter Stone. The utility of temporal abstraction in reinforcement learning. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems*, pages 299–306, 2008.

[67] Andreas Junghanns and Jonathan Schaeeer. Sokoban: a challenging single-agent search problem. In *International Joint Conference on Artificial Intelligence Workshop on Using Games as an Experimental Testbed for AI Reasearch*, 1997.

[68] Leslie Pack Kaelbling and Tomás Lozano-Pérez. Hierarchical planning in the now. In *Workshops at the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[69] Siddharth Karamcheti, Edward C. Williams, Dilip Arumugam, Mina Rhee, Nakul Gopalan, Lawson L.S. Wong, and Stefanie Tellex. A tale of two DRAGGNs: A hybrid approach for interpreting action-oriented and goal-oriented instructions. In *Annual Meeting of the Association for Computational Linguistics Workshop on Language Grounding for Robotics*, 2017.

[70] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3128–3137, 2015.

[71] Junkyung Kim, Matthew Ricci, and Thomas Serre. Not-so-clevr: learning same–different relations strains feedforward neural networks. *Interface focus*, 2018.

[72] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[73] Ross A Knepper, Stefanie Tellex, Adrian Li, Nicholas Roy, and Daniela Rus. Single assembly robot in search of human partner: Versatile grounded language generation. In *2013 8th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 167–168, 2013.

[74] Philipp Koehn and Rebecca Knowles. Six challenges for neural machine translation. 2017.

[75] George Konidaris, Scott Kuindersma, Roderic Grupen, and Andrew Barto. Robot learning from demonstration by constructing skill trees. *The International Journal of Robotics Research*, pages 360–375, 2012.

[76] George D. Konidaris. Constructing abstraction hierarchies using a skill-symbol loop. In *International Joint Conference on Artificial Intelligence*, pages 1648–1654, 2016.

[77] George D. Konidaris and Andrew G Barto. Building portable options: Skill transfer in reinforcement learning. In *IJCAI*, volume 7, pages 895–900, 2007.

[78] George D. Konidaris, Sarah Osentoski, and Philip Thomas. Value function approximation in reinforcement learning using the fourier basis. In *Twenty-fifth AAAI conference on artificial intelligence*, 2011.

[79] George D. Konidaris, Leslie Pack Kaelbling, and Tomas Lozano-Perez. From skills to symbols: Learning symbolic representations for abstract high-level planning. *Journal of Artificial Intelligence Research*, pages 215–289, 2018.

[80] George Dimitri Konidaris. *Autonomous robot skill acquisition*. University of Massachusetts Amherst, 2011.

[81] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. From structured english to robot motion. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2717–2722, 2007.

[82] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Translating structured english to robot controllers. *Advanced Robotics*, pages 1343–1359, 2008.

[83] Hadas Kress-Gazit, Georgios E Fainekos, and George J Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE Transactions on Robotics*, pages 1370–1381, 2009.

[84] Hadas Kress-Gazit, Morteza Lahijanian, and Vasumathi Raman. Synthesis for robots: Guarantees and feedback for robot behavior. *Annual Review of Control, Robotics, and Autonomous Systems*, 1(1), 2018.

[85] Jayant Krishnamurthy and Tom M. Mitchell. Weakly supervised training of semantic parsers. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 754–765, 2012.

[86] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 91–99, 2012.

[87] Tom Kwiatkowski, Luke Zettlemoyer, Sharon Goldwater, and Mark Steedman. Lexical generalization in CCG grammar induction for semantic parsing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 1512–1523. Association for Computational Linguistics, 2011.

[88] Brenden M. Lake and Marco G Baroni. Still not systematic after all these years: On the compositional skills of sequence-to-sequence recurrent networks. *CoRR*, abs/1711.00350, 2017.

[89] Steven M LaValle and James J Kuffner. Randomized kinodynamic planning. *The International Journal of Robotics Research*, pages 378–400, 2001.

[90] Lihong Li, Thomas J. Walsh, and Michael L. Littman. Towards a unified theory of state abstraction for MDPs. In *International Symposium on Artificial Intelligence and Mathematics*, 2006.

[91] Percy Liang. Learning executable semantic parsers for natural language understanding. *Communications of the ACM*, pages 68–76, 2016.

[92] Constantine Lignos, Vasumathi Raman, Cameron Finucane, Mitchell Marcus, and Hadas Kress-Gazit. Provably correct reactive control from natural language. *Autonomous Robots*, pages 89–105, 2015.

[93] Michael L. Littman, Thomas L. Dean, and Leslie Pack Kaelbling. On the complexity of solving Markov decision problems. In *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*, pages 394–402, 1995.

[94] Michael L. Littman, Ufuk Topcu, Jie Fu, Charles Lee Isbell, Min Wen, and James MacGlashan. Environment-independent task specifications via GLTL. *CoRR*, abs/1704.04341, 2017.

[95] Savvas G Loizou and Kostas J Kyriakopoulos. Automatic synthesis of multi-agent motion tasks based on ltl specifications. pages 153–158, 2004.

[96] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Empirical Methods in Natural Language Processing*, pages 1412–1421, 2015.

[97] James MacGlashan. Brown–UMBC Reinforcement Learning and Planning (BURLAP)–Project Page. http://burlap.cs.brown.edu/, 2014.

[98] James MacGlashan, Monica Babeş-Vroman, Marie DesJardins, Michael L. Littman, Smaranda Muresan, Shawn Squire, Stefanie Tellex, Dilip Arumugam, and Lei Yang. Grounding English commands to reward functions. In *Robotics: Science and Systems*, 2015.

[99] Matt MacMahon, Brian Stankiewicz, and Benjamin Kuipers. Walk the talk: Connecting language, knowledge, and action in route instructions. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference*, pages 1475–1482, 2006.

[100] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification*. Springer-Verlag New York, 1992.

[101] Ellen M Markman. *Categorization and naming in children: Problems of induction*. Mit Press, 1991.

[102] B. Marthi, S. J. Russell, and J. Wolfe. Angelic hierarchical planning: Optimal and online algorithms. In *International Conference on Automated Planning and Scheduling*, pages 222–231, 2008.

[103] Cynthia Matuszek, Nicholas FitzGerald, Luke Zettlemoyer, Liefeng Bo, and Dieter Fox. A joint model of language and perception for grounded attribute learning. *International Conference on Machine Learning*, 2012.

[104] Cynthia Matuszek, Evan Herbst, Luke Zettlemoyer, and Dieter Fox. Learning to parse natural language commands to a robot control system. In *Experimental Robotics*, pages 403–415, 2013.

[105] Amy McGovern, Richard S. Sutton, and Andrew H Fagg. Roles of macro-actions in accelerating reinforcement learning. *Grace Hopper Celebration of Women in Computing*, 1997.

[106] H. Brendan McMahan, Maxim Likhachev, and Geoffrey J. Gordon. Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In *International Conference on Machine Learning*, pages 569–576, 2005.

[107] Hongyuan Mei, Mohit Bansal, and Matthew R Walter. Listen, attend, and walk: neural mapping of navigational instructions to action sequences. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 2772–2778, 2016.

[108] Tomas Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[109] Tomas Mikolov, Stefan Kombrink, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. Extensions of recurrent neural network language model. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 5528–5531, 2011.

[110] Tomas Mikolov, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[111] Dipendra Misra, John Langford, and Yoav Artzi. Mapping instructions and visual observations to actions with reinforcement learning. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1004–1015, 2017.

[112] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, pages 529–533, 2015.

[113] Richard Montague. Universal grammar. *Theoria*, pages 373–398, 1970.

[114] Richard Montague. The proper treatment of quantification in ordinary english. In *Approaches to natural language*, pages 221–242. 1973.

[115] Negin Nejati, Pat Langley, and Tolga Konik. Learning hierarchical task networks by observation. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 665–672, 2006.

[116] Andrew Y. Ng and Stuart J. Russell. Algorithms for inverse reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 663–670, 2000.

[117] Scott Niekum, Sarah Osentoski, George Konidaris, and Andrew G Barto. Learning and generalization of complex tasks from unstructured demonstrations. In *IEEE/RSJ Intelligent Robots and Systems*, pages 5239–5246, 2012.

[118] Nils J. Nilsson. Shakey the robot. *AI Center, SRI International, Tech. Rep.*, 1984.

[119] John Oberlin and Stefanie Tellex. Ein 1.0. `http://h2r.github.io/ein/`. Accessed: 2018-12-29.

[120] Edwin Olson. Apriltag: A robust and flexible multi-purpose fiducial system. Technical report, University of Michigan APRIL Laboratory, May 2010.

[121] Christos Papadimitriou and John N. Tsitsiklis. The complexity of Markov decision processes. *Mathematics of Operations Research*, pages 441–450, 1987.

[122] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *Neural Information Processing Systems Workshop on The Future of Gradient-based Machine Learning Software & Techniques*, 2017.

[123] Rohan Paul, Jacob Arkin, Nicholas Roy, and Thomas M. Howard. Efficient grounding of abstract spatial concepts for natural language interaction with robot manipulators. In *Robotics: Science and Systems*, 2016.

[124] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[125] Amir Pnueli. The temporal logic of programs. In *IEEE Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[126] Morgan Quigley, Josh Faust, Tully Foote, and Jeremy Leibs. ROS: an open-source robot operating system. In *IEEE International Conference on Robotics and Automation Workshop on Open Source Software*, 2009.

[127] Vasumathi Raman and Hadas Kress-Gazit. Analyzing unsynthesizable specifications for high-level robot behavior using LTLMoP. In *International Conference on Computer-Aided Verification*, pages 663–668, 2011.

[128] Vasumathi Raman, Constantine Lignos, Cameron Finucane, Kenton Lee, Mitch Marcus, and Hadas Kress-Gazit. Sorry Dave, I'm Afraid I Can't Do That: Explaining Unachievable Robot Tasks Using Natural Language. In *Robotics: Science and Systems*, 2013.

[129] Pravesh Ranchod, Benjamin Rosman, and George Konidaris. Nonparametric bayesian reward segmentation for skill discovery using inverse reinforcement learning. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 471–477. IEEE, 2015.

[130] Melrose Roderick, Christopher Grimm, and Stefanie Tellex. Deep abstract q-networks. In *International Conference on Autonomous Agents and MultiAgent Systems*, pages 131–138, 2018.

[131] Bernhard Schölkopf, Robert C Williamson, Alex J Smola, John Shawe-Taylor, and John C Platt. Support vector method for novelty detection. In *Advances in neural information processing systems*, pages 582–588, 2000.

[132] Stephanie Seneff. Tina: A natural language system for spoken language applications. *Computational linguistics*, pages 61–86, 1992.

[133] Pierre Sermanet, Kelvin Xu, and Sergey Levine. Unsupervised perceptual rewards for imitation learning. *CoRR*, abs/1612.06699, 2016.

[134] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, pages 1929–1958, 2014.

[135] Mark Steedman. *The Syntactic Process*. MIT Press, 2000.

[136] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[137] Richard S Sutton, , and Andrew G Barto. *Introduction to reinforcement learning*, volume 2. Cambridge: MIT Press, 1998.

[138] Richard S Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, pages 181–211, 1999.

[139] Yee W Teh, Michael I Jordan, Matthew J Beal, and David M Blei. Sharing clusters among related groups: Hierarchical dirichlet processes. In *Advances in neural information processing systems*, pages 1385–1392, 2005.

[140] Stefanie Tellex, Thomas Kollar, Steven Dickerson, Matthew R. Walter, Ashis Gopal Banerjee, Seth Teller, and Nicholas Roy. Understanding natural language commands for robotic navigation and mobile manipulation. In *AAAI Conference on Artificial Intelligence*, 2011.

[141] Stefanie Tellex, Ross Knepper, Adrian Li, Daniela Rus, and Nicholas Roy. Asking for help using inverse semantics. In *Robotics Science and Systems*, 2014.

[142] Cristian-Ioan Vasile, Derya Aksaray, and Calin Belta. Time window temporal logic. *Theoretical Computer Science*, pages 27–54, 2017.

[143] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[144] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3156–3164, 2015.

[145] voicebot.ai. U.S. Smart Speaker Ownership, 2019. URL `https://web.archive.org/web/20190604225900/https://voicebot.ai/2019/03/07/u-s-smart-speaker-ownership-rises-40-in-2018-to-66-4-million-and-amazon-echo-maintains-market-share-lead-says-new-report-from-voicebot/`.

[146] David HD Warren and Fernando CN Pereira. An efficient easily adaptable system for interpreting natural language queries. *Computational Linguistics*, pages 110–122, 1982.

[147] Andrew K Watson. Automated creation of labeled pointcloud datasets in support of machine-learning based perception. Technical report, Naval Postgraduate School Monterey United States, 2017.

[148] Sandra R Waxman and Jeffrey L Lidz. *Early world learning*. Wiley Online Library.

[149] Edward C. Williams, Nakul Gopalan, Mina Rhee, and Stefanie Tellex. Learning to parse natural language to grounded reward functions with weak supervision. In *International Conference on Robots and Automation*, 2018.

[150] Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, pages 270–280, 1989.

[151] John Winder, , Shawn Squire, Matthew Landen, Stephanie Milani, and Marie desJardins. Towards planning with hierarchies of learned markov decision processes. In *In ICAPS-2017 Integrated Execution of Planning and Acting Workshop*, 2017.

[152] Terry Winograd. Procedures as a representation for data in a computer program for understanding natural language. Technical report, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1971.

[153] William A Woods. Progress in natural language understanding: an application to lunar geology. In *Proceedings of national computer conference and exposition*, pages 441–450. ACM, 1973.

[154] Tatsuro Yamada, Shingo Murata, Hiroaki Arie, and Tetsuya Ogata. Dynamical linking of positive and negative sentences to goal-oriented robot behavior by hierarchical RNN. In *International Conference on Artificial Neural Networks*, 2016.

[155] Luke Zettlemoyer and Michael Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, pages 658–666, 2005.

[156] Luke Zettlemoyer and Michael Collins. Online learning of relaxed CCG grammars for parsing to logical form. In *Proceedings of the Joint Conference on Empirical Methods in Natural Lanugage Processing and Computational Natural Language Learning*, pages 678–687, 2007.