Training an Agent to Ground Commands
with Reward and Punishment

James MacGlashan,* Michael Littman,* Robert Loftin,†
Bei Peng,# David Roberts,† Matthew E. Taylor‡
* Computer Science Department, Brown University
† North Carolina State University
‡ School of Electrical Engineering and Computer Science, Washington State University

Abstract
As robots and autonomous assistants become more capable, there will be a greater need for humans to easily convey to agents the complex tasks they want them to carry out. Conveying tasks through natural language provides an intuitive interface that does not require any technical expertise, but implementing such an interface requires methods for the agent to learn a grounding of natural language commands. In this work, we demonstrate how high-level task groundings can be learned from a human trainer providing online reward and punishment. Grounding language to high-level tasks for the agent to solve removes the need for the human to specify low-level solution details in their command. Using reward and punishment for training makes the training procedure simple enough to be used by people without technical expertise and also allows a human trainer to immediately correct errors in interpretation that the agent has made. We present preliminary results from a single user training an agent in a simple simulated home environment and show that the agent can quickly learn a grounding of language such that the agent can successfully interpret new commands and execute them in a variety of different environments.

1 Introduction
As robots and autonomous assistants become more capable, there will be a greater need for humans to easily convey to agents the complex tasks they want them to carry out. Conveying tasks through natural language provides an intuitive interface that does not require any technical expertise, but implementing such an interface requires methods for the agent to learn a grounding of natural language commands. In this work, we propose a method for learning task groundings of natural language commands from online reward and punishment given by a human trainer.

While there is existing work on learning command groundings (see Section 5), most of it is focused on natural language commands that specify the actions the agent needs to take. However, we should not have to explicitly tell an autonomous agent how to complete a task; the agent should solve the problem on its own. For example, if a human wants a robot to bring them a red bag, they should simply be able to say “bring me the red bag,” instead of detailing how the robot needs to navigate to the bag, pick it up, and navigate back to them. Existing approaches for training natural language command groundings focus on using training data that consists of language paired with example demonstrations. However, depending on how complex an agent is, manually controlling it to provide demonstrations may be inconvenient. For instance, if the agent is a robot, teleoperating the robot to provide it demonstrations may 1) be too time consuming and cumbersome without a well developed interface, and 2) require the operator to have sufficient expertise to produce high quality demonstrations. An easier way to train the robot is to reward and punish the robot as it is executing a command it was given, similar to how dogs are trained to follow a command through reward and punishment. A reward and punishment paradigm has the additional advantage that if an agent misunderstands a command, a human can easily correct the agent as the agent is attempting to carry out the command, rather than stopping the agent and then providing a new demonstration and language example.

Our approach to learning natural language command groundings makes use of a generative model of tasks, language, behavior, and trainer feedback. The task, language, and behavior model builds on prior work by MacGlashan et al. (2014) in which tasks are defined as factored reward functions and termination conditions of an MDP, language is modeled using IBM Model 2 (Brown et al. 1990; 1993), and behavior is generated from the optimal policy for a given task. We extend this approach to enable learning from reward and punishment, rather than demonstration, by including in the model trainer feedback. To model trainer feedback, we adopt algorithms proposed by Loftin et al. (2014) for performing task inference from human feedback (without any language grounding) in which feedback is treated as a discrete value (reward, punishment, neutral) and given by a human trainer according to parameters defining their willingness to reward or punish correct and incorrect action selection. The contribution of our work is to show how these two approaches can be combined in a fully interactive system that enables an agent to learn the task groundings of natural language commands.

In our learning algorithm, an agent is first given a command, which induces a probability distribution over the set of possible tasks in the environment. The agent then plans a solution to the most likely task using an “off-the-shelf” MDP planning algorithm and begins following the policy.
After each action execution, the trainer either rewards, punishes, or provides no feedback to the agent, which is evidence for or against each of the possible tasks. If a different task than the one currently being executed becomes more likely than the current one, then the agent switches to following the policy of the new most likely task. After a task has been completed, the final probability distribution over tasks is used to update the language model so that the agent will better understand future commands.

In this work we present preliminary results from a single user training an agent in a simple simulated home environment and show that the agent can quickly learn to correctly interpret and execute new commands.

2 Background

To represent tasks, we make use of the Object-oriented Markov Decision Process (OO-MDP) formalism (Diuk, Cohen, and Littman 2008). OO-MDPs extend the conventional MDP formalism by providing a rich state representation. MDPs are defined by a four-tuple: \((S, A, T, R)\), where \(S\) is a set of states of the world; \(A\) is a set of actions that the agent can take; \(T\) describes the transition dynamics, which specify the probability of the agent transitioning to each state given the current state and the action taken; and \(R\) is a function specifying the reward received by the agent for each transition. MDPs may also have terminal states that cause action to cease once reached. The goal of planning in an MDP is to find a policy—a mapping from states to actions—that maximizes the expected discounted cumulative reward.

An OO-MDP extends the classic MDP formalism by including a set of object classes, each defined by a set of attributes, and a set of propositional functions whose parameters are typed to object classes. A state in an OO-MDP is a set of objects that each belong to one of the possible object classes; each object has its own state which is represented as a value assignment to the attributes of its associated object class. The propositional functions defined in the OO-MDP are functions of the state of its object parameters. For example, in a blocks world, the function \(\text{s2}(b_1, b_2)\) would operate on two block objects and return true when the state of \(b_1\) was adjacent and above the state of \(b_2\). Although an OO-MDP state is fully defined by the union of objects, these propositional functions provide additional high-level information about the state. In this work, we use propositional functions to define abstract task definitions. Note that although OO-MDPs provide a rich state representation, any standard MDP planning algorithm can be used for planning in an OO-MDP.

3 Approach

Our learning algorithm uses a generative model consisting of task definitions, language, behavior, and trainer feedback. Figure 1 shows a diagram of this model, with arrows indicating conditional probabilities. The next section will give an overview of each part of the model, and present the overall interactive learning algorithm used to learn the parameters of the language model. Our running example is the simplified home environment shown in Figure 2.

2 Background

To represent tasks, we make use of the Object-oriented Markov Decision Process (OO-MDP) formalism (Diuk, Cohen, and Littman 2008). OO-MDPs extend the conventional MDP formalism by providing a rich state representation. MDPs are defined by a four-tuple: \((S, A, T, R)\), where \(S\) is a set of states of the world; \(A\) is a set of actions that the agent can take; \(T\) describes the transition dynamics, which specify the probability of the agent transitioning to each state given the current state and the action taken; and \(R\) is a function specifying the reward received by the agent for each transition. MDPs may also have terminal states that cause action to cease once reached. The goal of planning in an MDP is to find a policy—a mapping from states to actions—that maximizes the expected discounted cumulative reward.

An OO-MDP extends the classic MDP formalism by including a set of object classes, each defined by a set of attributes, and a set of propositional functions whose parameters are typed to object classes. A state in an OO-MDP is a set of objects that each belong to one of the possible object classes; each object has its own state which is represented as a value assignment to the attributes of its associated object class. The propositional functions defined in the OO-MDP are functions of the state of its object parameters. For example, in a blocks world, the function \(\text{s2}(b_1, b_2)\) would operate on two block objects and return true when the state of \(b_1\) was adjacent and above the state of \(b_2\). Although an OO-MDP state is fully defined by the union of objects, these propositional functions provide additional high-level information about the state. In this work, we use propositional functions to define abstract task definitions. Note that although OO-MDPs provide a rich state representation, any standard MDP planning algorithm can be used for planning in an OO-MDP.

3 Approach

Our learning algorithm uses a generative model consisting of task definitions, language, behavior, and trainer feedback. Figure 1 shows a diagram of this model, with arrows indicating conditional probabilities. The next section will give an overview of each part of the model, and present the overall interactive learning algorithm used to learn the parameters of the language model. Our running example is the simplified home environment shown in Figure 2.
The language model we use is IBM Model 2 (Brown et al. 1990; 1993), a statistical machine translation model. In machine translation, the goal is to translate a sentence in a source language (French) into a target language (English). The model consists of translation parameters, giving the probability \( \Pr(f_i|e_j) \) of English word \( e_j \) translating to French word \( f_i \); and word alignment parameters, giving the probability \( \Pr(j|m, n, k) \) of the \( i \)th word in a French sentence of length \( k \) corresponding to the \( j \)th word in an English sentence of length \( m \). Given a set of pairs of English and French sentences, these parameters can be learned with Expectation Maximization (Dempster, Laird, and Rubin 1977).

In this work, we use the generative model to translate from an English command \( e \), to a lifted task and variable binding constraints. Because IBM Model 2 is designed to translate between sentences in different languages, the lifted task and variable binding constraints are deterministically transformed into a machine language string \( m \). This transformation iterates over each propositional function in the lifted task and then each propositional function in the variable binding constraints. Each propositional function is turned into a sequence of tokens consisting of the name of the propositional function and the object classes to which each of its parameters are typed. For instance, lifted task 1 and variable binding constraints \( \text{isYellow}(c_1) \land \text{isRed}(r_1) \) would become the machine string “blockInRoom block room isYellow block isRed room.” Once converted into a machine language string, IBM Model 2 can be used to compute the probability \( \Pr(e|m) \) of an English command \( e \) given a machine command \( m \). Translation in the reverse direction is performed using Bayesian inference.

**Trainer Feedback Model**

The trainer feedback model assumes that a trainer will reward, punish, or do nothing (neutral feedback), in response to the agent taking a correct or incorrect action, with respect to the task they are training. The model contains three parameters: \( \mu^+, \mu^-, \) and \( \epsilon \). \( \mu^+ \) is the probability that the trainer will give neutral feedback when a correct action is taken, and \( \mu^- \) is the probability that they will give neutral feedback when an incorrect action is taken. \( \epsilon \) is the probability that the trainer will misinterpret a correct action as incorrect, or vice-versa. Given an MDP, an action is assumed to be correct if it is an optimal action for the MDP in the current state, and incorrect otherwise. Therefore, the probability of the agent receiving reward (\( r \)), punishment (\( p \)), or neutral feedback (\( \emptyset \)) is defined as

\[
\begin{align*}
\Pr(f = r|s, a, t) &= \left\{ \begin{array}{ll}
(1 - \epsilon)(1 - \mu^+) & \text{if } a \in \pi^+ s \\
(1 - \epsilon)(\mu^+) & \text{if } a \notin \pi^+ s
\end{array} \right.
\\
\Pr(f = p|s, a, t) &= \epsilon(1 - \mu^-) & \text{if } a \in \pi^- s \\
\Pr(f = \emptyset|s, a, t) &= \left\{ \begin{array}{ll}
(1 - \epsilon)\mu^+ + \epsilon\mu^- & \text{if } a \in \pi s \\
(1 - \epsilon)\mu^- + \epsilon\mu^+ & \text{if } a \notin \pi s
\end{array} \right.
\end{align*}
\]

where \( a \) is the action the agent took in state \( s \) and \( \pi s \) is the set of optimal actions in state \( s \) for task \( t \). Given a new feedback signal, Bayesian inference can be used to update the belief in each task. Specifically,

\[
\Pr(t|s, a, f) \propto \Pr(t) \Pr(f|s, a, t).
\]
This model is useful in practice because trainers may give more reward than punishment (e.g., rewarding correct actions and doing nothing for incorrect actions), or vice-versa. If the agent knows (or can learn) that the trainer gives more reward or punishment ($\mu^+ \neq \mu^-$), then neutral feedback can provide as much information as explicit feedback (Loftin et al. 2014). Without knowing, however, it is best to set $\mu^+ = \mu^-$, such that neutral feedback is ignored.

**Learning Algorithm**

Our learning algorithm is shown in Algorithm 1. The algorithm takes as input an OO-MDP definition without the reward function and the set of lifted tasks that the agent can solve. The algorithm begins by initializing the IBM Model 2 parameters arbitrarily and creating an initially empty training dataset $D$. The algorithm then loops over successive training sessions that begin with a new input state. Each input state may have a different number of objects present, with variable properties, which effectively makes each input state a new environment. The agent is then given a natural language command $e$ that the trainer wants the agent to complete. Using the current IBM Model 2 parameters, the prior over grounded tasks is updated to the posterior given the command. The agent then uses any “off-the-shelf” MDP planning algorithm to find the policy for each task. This begins a behavior loop in which the agent follows the policy of the most likely grounded task, receives a feedback, updates its belief in each grounded task, and switches to the policy of the new most likely grounded task. This behavior loop continues until the trainer terminates the session. For each possible machine language command $m$, a training instance is added to dataset $D$. Each training instance consists of the machine language command ($m$), the natural language command ($e$), and a weight that is the posterior probability of $m$ given the initial command $e$, and the state, action, and feedback sequence observed during the last behavior loop. This probability is computed by marginalizing over the lifted task, grounded task, and binding constraints:

$$Pr(m|s, a, f, e) \propto \sum_{l, t, c} Pr(l|s) Pr(t|s, l) Pr(f|s, a, t)$$

$$Pr(c|s, l, t) Pr(m|l, e) Pr(e|l, c),$$

where $Pr(f|s, a, t)$ is the probability of feedback sequence $f$, which is computed by treating each feedback as independent of the other state-action-feedback triples in the sequence:

$$Pr(f|s, a, t) = \prod_i Pr(f_i|s_i, a_i, t).$$

Finally, the IBM Model 2 parameters are retrained using the updated dataset $D$. The training algorithm for IBM Model 2 is a weakly supervised Expectation Maximization (EM) algorithm, as in prior work (MacGlashan et al. 2014), similar to the classic EM algorithm for IBM Model 2, except that the contribution of each data instance is weighted, as the intended task must be inferred probabilistically.

**4 Experimental Results**

To test the model, we performed preliminary experiments with a single trainer (one of the authors) training an agent in the same simplified simulated home environment discussed previously. The OO-MDP representing this domain consists of four object classes: AGENT, ROOM, DOOR, and BLOCK. The AGENT class is defined by $x$ and $y$ position attributes; the ROOM class is defined by attributes defining the bounding box of the room and the color the room is painted; the DOOR class is defined by attributes defining its bounding box; and the BLOCK class is defined by $x$ and $y$ position attributes, a color attribute, and a shape attribute. The possible color values are red, green, blue, yellow, and purple; the possible shapes are chair, bag, backpack, and basket (though in these experiments only the chair and bag were used). The agent can deterministically move one unit north, south, east, or west and the agent can move blocks by moving into them. Propositional functions in the OO-MDP include propositional functions for checking if a block is a certain shape (e.g., shapeChair(BLOCK)), checking if a block is a certain color (e.g., isRed(BLOCK), checking if a room is painted a certain color, checking if the agent is in a room (agentInRoom(AGENT, ROOM)), and checking if a block is in a room (blockInRoom(BLOCK, ROOM)).

The possible lifted tasks given to the agent were a task for taking a block to a room, as previously defined in Equation 1, as well as a task for going to a room, which is similar in form except it uses the agentInRoom propositional function. Because these tasks are goal directed, the corresponding goal states were also terminating states. As this domain is deterministic, uniform cost search was used for planning. If the agent reached a terminating state for the task it was completing, then it was forced to always take a special “noop” action that did not change the state.
The training interface, shown in Figure 3, lets the trainer create custom environments, give commands, and then interactively reward and punish the agent. A training session can be ended at any time with the choice of adding it to the language model’s training dataset or ignoring it.

We chose a training sequence of environments/initial states and commands to iteratively teach the agent about a different property of the commands. The sequence of training environments are shown in Figures 4a to 4f (4g is only used for testing after training in the former environments), with environment 4a being used twice in a row and the rest only used once.

The sequence of commands and the associated environment in which they were given are shown in Table 1. Note that in the first four examples, the agent is told to go to a different colored room with no other objects in the environment. This initial set of training commands is to train the agent on the meaning of the room color words. In the following three commands, a single movable object is present, which the agent is asked to take to a room. The first time, it’s a blue chair, then a purple bag, and then a yellow bag. The goal of this training sequence is that the agent should have been given enough examples to disambiguate which words refer to room color, block color, or block shape.

Table 1: Sequence of training commands and environments.

<table>
<thead>
<tr>
<th>#</th>
<th>Command Given</th>
<th>Associated Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>go to the green room</td>
<td>4a</td>
</tr>
<tr>
<td>2</td>
<td>go to the red room</td>
<td>4a</td>
</tr>
<tr>
<td>3</td>
<td>go to the blue room</td>
<td>4b</td>
</tr>
<tr>
<td>4</td>
<td>go to the yellow room</td>
<td>4c</td>
</tr>
<tr>
<td>5</td>
<td>take the blue chair to the yellow room</td>
<td>4d</td>
</tr>
<tr>
<td>6</td>
<td>take the purple bag to the blue room</td>
<td>4e</td>
</tr>
<tr>
<td>7</td>
<td>take the yellow bag to the red room</td>
<td>4f</td>
</tr>
</tbody>
</table>

During each training session, the trainer employed a feedback strategy of giving reward and punishment intermittently for correct and incorrect action selection. The trainer feedback strategy parameters were set to values that reflect this strategy ($\mu^+ = 0.7$, $\mu^- = 0.7$, and $\epsilon = 0.1$). It is worth noting that during training, the agent correctly interpreted the command without any required reward or punishment (though it was given anyway) for training commands 2, 5, 6, and 7. For command 2, it was most likely luck that the agent interpreted the command correctly since it had such limited experience. Although the objects to move in instance 5, 6, and 7 were novel, there was only one possible block to move, which meant that to interpret the command correctly, all the agent needed to know was that it was a block moving task and which color room to take the block, which it should have been able to determine from its previous experiences.

To test if the agent learned to properly associate words with the relevant semantic components after the training period, a single testing environment, shown in Figure 4g, was used and after each test command in this environment, the training session was completed without updating the language model so that any new data did not contaminate the results. Note that this environment is different from all former environments and includes a block shape and color pairs not previously seen. Specifically, this environment included a purple chair and a blue bag. If the language model associated words with the correct semantic components, however, then the agent would be able to properly interpret the commands. The test commands given are shown in Table 2. The first two test commands test whether the agent was able to properly disambiguate which block it was being requested to move, even though the agent had never been asked to move that specific kind of block. The last three test sentences were designed to test whether the agent still understood commands to go to rooms, even in the presence of novel objects. The agent correctly interpreted the test commands in all cases.

Table 2: Sequence of test commands, in environment 4g.

<table>
<thead>
<tr>
<th>#</th>
<th>Test commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>take the purple chair to the green room</td>
</tr>
<tr>
<td>2</td>
<td>take the blue bag to the yellow room</td>
</tr>
<tr>
<td>3</td>
<td>go to the green room</td>
</tr>
<tr>
<td>4</td>
<td>go to the red room</td>
</tr>
<tr>
<td>5</td>
<td>go to the blue room</td>
</tr>
</tbody>
</table>

Another way to gain insight into what the agent learned is to examine the translation parameters of IBM Model 2 after training is completed. Although there is not space to list all the parameters, we provide some interesting examples here. The machine language word “agentInRoom” was most likely to generate the English word “go” (with probability 0.51); “shapeBag” was most likely to generate “bag” (probability 0.32); “roomIsRed” was most likely to generate “red” (probability 0.33); and “blockInRoom” was most likely to generate the word “take” (probability 0.33).

As a final test to demonstrate the utility of modeling different feedback strategies, this same experiment was rerun, except the trainer was only allowed to punish the agent. To reflect this change in strategy, the trainer feedback model...
parameters were changed accordingly\(^1\) \((\mu^+ = 0.7, \mu^- = 0.05)\). During training, the agent only incorrectly interpreted commands one and three, which means for the remaining training sessions, the trainer provided no explicit feedback at all. Despite this lack of explicit feedback, when coupled with the feedback strategy model parameters, the agent was still able to learn the command groundings. On the test commands, the agent correctly interpreted test commands 1-4. The agent did misinterpret test command 5, but switched to the correct behavior after a single punishment.

5 Related Work

The most similar work to ours is work that learns command groundings from demonstrations in the world, rather than using annotated text. For example, previous work that investigates grounding commands from demonstration included work on giving navigational instructions (Vogel and Jurafsky 2010; Chen and Mooney 2011; Grubb et al. 2011), giving action commands to a robot (Duvallet, Kollar, and Stentz 2013; Tellex et al. 2011; 2014), controlling software (Branavan et al. 2009; Branavan, Zettlemoyer, and Barzilay 2010) and playing games (Goldwasser and Roth 2011; Branavan, Silver, and Barzilay 2011). Two critical differences between this body of work and ours is (1) that our work aims to ground language to task descriptions that do not require the human to tell the agent how to complete the task and (2) our learning algorithm grounds language from online reward and punishment delivered by a human.

Two related approaches to training an agent with human reward and punishment are TAMER (Knox and Stone 2009) and COBOT (Isbell et al. 2001). In these algorithms, human feedback is modeled as a reward function which is then used in a typical reinforcement learning algorithm. One disadvantage to this approach is that it cannot make use of different training strategies employed by the trainer; additionally, the modeled reward function is highly dependent on each state in the state space, which does not capture an overall task definition that can easily be connected with language.

Other related work focuses on how to learn the behavior, rather than identify the task that needs to be completed. Rybski et al. (2007) propose a method to train a robot to complete novel tasks from demonstrations that incorporate language with \textit{a priori} grounded utterances. Tenorio-Gonzalez et al. (2010) use language to shape an already known reward function. Hewlett et al. (2010) learn the behavior of specified verb phrases as abstract finite state machines. Future work could incorporate such approaches to reduce the computational complexity of planning.

6 Conclusions

We proposed a learning algorithm for grounding natural language commands to high-level task definitions from online human-delivered reward and punishment. Grounding commands to high-level task definitions is especially useful because it enables the human to convey tasks to an agent without having to tell the agent how to carry it out. As a result, the same command can be given in a variety of different states/environments and the agent will autonomously determine how to carry out the task in each of them. Training with reward and punishment has the advantage that it does not require technical expertise in manually controlling the agent to provide demonstrations.

We presented preliminary results of a single user interactively training an agent in a simplified simulated home environment and demonstrated that after only a few examples the agent was able to correctly interpret novel commands. In the future, we will perform a user study to assess how well the system handles the kind of variable language and feedback that would be provided by novice users who do not understand the underlying system.

A limitation of our approach is that the agent must plan for every possible task that could be performed in the environment. As environments get more complex and more tasks are possible, this will become intractable. In the future, we would like to explore sampling methods to reduce the space of tasks considered and couple that with "scaffolding" where initially training with simple environments allows the agent to easily learn word groundings, and therefore allows the agent to focus on the most likely task groundings for a command in more complex environments.

In this work, we focused on providing the agent no background language knowledge, since being able to learn without any background language knowledge reduces the amount of work required by a designer. However, in practice, background language knowledge could be incorporated to further accelerate learning. For example, a dictionary could be used to generalize the grounding of one observed word to its synonyms. Additionally, if certain words were known to correspond to colors, then they could similarly have their translation probabilities biased toward being generated by color propositional functions.

\(^1\)Alternatively, these parameters could be learned online using the I-SABL algorithm (Loftin et al. 2014).
7 Acknowledgements

This work was supported in part by NSF IIS-1149917 and NSF IIS-1319412.

References


