Reading Between the Lines: Learning to Map High-level Instructions to Commands

S.R.K. Branavan, Luke S. Zettlemoyer, Regina Barzilay
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{branavan, lsz, regina}@csail.mit.edu

Abstract

In this paper, we address the task of mapping *high-level instructions* to sequences of commands in an external environment. Processing these instructions is challenging—they posit goals to be achieved without specifying the steps required to complete them. We describe a method that fills in missing information using an automatically derived environment model that encodes states, transitions, and commands that cause these transitions to happen. We present an efficient approximate approach for learning this environment model as part of a policy-gradient reinforcement learning algorithm for text interpretation. This design enables learning for mapping high-level instructions, which previous statistical methods cannot handle.[1]

1 Introduction

In this paper, we introduce a novel method for mapping high-level instructions to commands in an external environment. These instructions specify goals to be achieved without explicitly stating all the required steps. For example, consider the first instruction in Figure 1 — "open control panel." The three GUI commands required for its successful execution are not explicitly described in the text, and need to be inferred by the user. This dependence on domain knowledge makes the automatic interpretation of high-level instructions particularly challenging.

The standard approach to this task is to start with both a manually-developed model of the environment, and rules for interpreting high-level instructions in the context of this model (Agre and Chapman, 1988; Di Eugenio and White, 1992; Di Eugenio, 1992; Webber et al., 1995). Given both the model and the rules, logic-based inference is used to automatically fill in the intermediate steps missing from the original instructions.

Our approach, in contrast, operates directly on the textual instructions in the context of the interactive environment, while requiring no additional information. By interacting with the environment and observing the resulting feedback, our method automatically learns both the mapping between the text and the commands, and the underlying model of the environment. One particularly noteworthy aspect of our solution is the interplay between the evolving mapping and the progressively acquired environment model as the system learns how to interpret the text. Recording the state transitions observed during interpretation allows the algorithm to construct a relevant model of the environment. At the same time, the environment model enables the algorithm to consider the consequences of commands before they are executed, thereby improving the accuracy of interpretation. Our method efficiently achieves both of these goals as part of a policy-gradient reinforcement learning algorithm.

We apply our method to the task of mapping software troubleshooting guides to GUI actions in the Windows environment (Branavan et al., 2009; Kushman et al., 2009). The key findings of our experiments are threefold. First, the algorithm can accurately interpret 61.5% of high-level instructions, which cannot be handled by previous statistical systems. Second, we demonstrate that explicitly modeling the environment also greatly improves the accuracy of processing low-level instructions, yielding a 14% absolute increase in performance over a competitive baseline (Branavan et al., 2009). Finally, we show the importance of constructing an environment model relevant to the language interpretation task — using textual

[1]Code, data, and annotations used in this work are available at http://groups.csail.mit.edu/rbg/code/rl-hli/

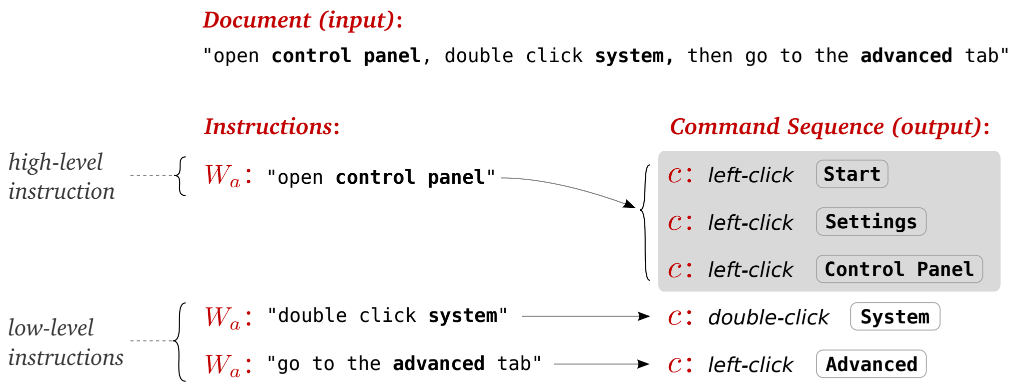
Document (input):
"open control panel, double click system, then go to the advanced tab"
Instructions:
Command Sequence (output):
high-level instruction
W_a: "open control panel"
c: left-click Start
c: left-click Settings
c: left-click Control Panel
low-level instructions
W_a: "double click system"
c: double-click System
W_a: "go to the advanced tab"
c: left-click Advanced

Figure 1: An example mapping of a document containing high-level instructions into a candidate sequence of five commands. The mapping process involves segmenting the document into individual instruction word spans W_a, and translating each instruction into the sequence $\vec{c}$ of one or more commands it describes. During learning, the correct output command sequence is not provided to the algorithm.

instructions enables us to bias exploration toward transitions relevant for language learning. This approach yields superior performance compared to a policy that relies on an environment model constructed via random exploration.

2 Related Work

Interpreting Instructions Our approach is most closely related to the reinforcement learning algorithm for mapping text instructions to commands developed by Branavan et al. (2009) (see Section 4 for more detail). Their method is predicated on the assumption that each command to be executed is explicitly specified in the instruction text. This assumption of a direct correspondence between the text and the environment is not unique to that paper, being inherent in other work on grounded language learning (Siskind, 2001; Oates, 2001; Yu and Ballard, 2004; Fleischman and Roy, 2005; Mooney, 2008; Liang et al., 2009; Matuszek et al., 2010). A notable exception is the approach of Eisenstein et al. (2009), which learns how an environment operates by reading text, rather than learning an explicit mapping from the text to the environment. For example, their method can learn the rules of a card game given instructions for how to play.

Many instances of work on instruction interpretation are replete with examples where instructions are formulated as high-level goals, targeted at users with relevant knowledge (Winograd, 1972; Di Eugenio, 1992; Webber et al., 1995; MacMahon et al., 2006). Not surprisingly, automatic approaches for processing such instructions have relied on hand-engineered world knowledge to reason about the preconditions and effects of environment commands. The assumption of a fully specified environment model is also common in work on semantics in the linguistics literature (Lascarides and Asher, 2004). While our approach learns to analyze instructions in a goal-directed manner, it does not require manual specification of relevant environment knowledge.

Reinforcement Learning Our work combines ideas of two traditionally disparate approaches to reinforcement learning (Sutton and Barto, 1998). The first approach, *model-based learning*, constructs a model of the environment in which the learner operates (e.g., modeling location, velocity, and acceleration in robot navigation). It then computes a policy directly from the rich information represented in the induced environment model. In the NLP literature, model-based reinforcement learning techniques are commonly used for dialog management (Singh et al., 2002; Lemon and Konstas, 2009; Schatzmann and Young, 2009). However, if the environment cannot be accurately approximated by a compact representation, these methods perform poorly (Boyan and Moore, 1995; Jong and Stone, 2007). Our instruction interpretation task falls into this latter category,[2] rendering standard model-based learning ineffective.

The second approach – model-free methods such as *policy learning* – aims to select the opti-

[2]For example, in the Windows GUI domain, clicking on the *File* menu will result in a different submenu depending on the application. Thus it is impossible to predict the effects of a previously unseen GUI command.

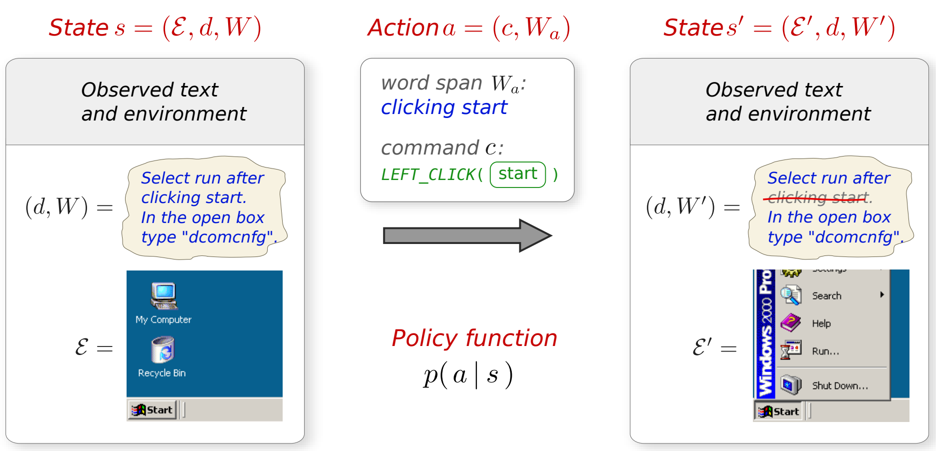
State $s = (\mathcal{E}, d, W)$
Action $a = (c, W_a)$
State $s' = (\mathcal{E}', d, W')$
Observed text and environment
word span W_a: clicking start
command c: LEFT_CLICK(start)
$(d, W) =$
Select run after clicking start. In the open box type "dcomcnfg".
$\mathcal{E} =$
My Computer
Recycle Bin
Start
Policy function $p(a \mid s)$
Observed text and environment
$(d, W') =$
Select run after ~~clicking start~~. In the open box type "dcomcnfg".
$\mathcal{E}' =$
Windows 2000 Pro
Search
Help
Run...
Shut Down...
Start

Figure 2: A single step in the instruction mapping process formalized as an MDP. State s is comprised of the state of the external environment $\mathcal{E}$, and the state of the document (d, W), where W is the list of all word spans mapped by previous actions. An action a selects a span W_a of unused words from (d, W), and maps them to an environment command c. As a consequence of a, the environment state changes to $\mathcal{E}' \sim p(\mathcal{E}'|\mathcal{E}, c)$, and the list of mapped words is updated to $W' = W \cup W_a$.

mal action at every step, without explicitly constructing a model of the environment. While policy learners can effectively operate in complex environments, they are not designed to benefit from a learned environment model. We address this limitation by expanding a policy learning algorithm to take advantage of a partial environment model estimated during learning. The approach of conditioning the policy function on future reachable states is similar in concept to the use of *post-decision state* information in the approximate dynamic programming framework (Powell, 2007).

3 Problem Formulation

Our goal is to map instructions expressed in a natural language document d into the corresponding sequence of commands $\vec{c} = \langle c_1, \ldots, c_m \rangle$ executable in an environment. As input, we are given a set of raw instruction documents, an environment, and a reward function as described below.

The *environment* is formalized as its states and transition function. An *environment state* $\mathcal{E}$ specifies the objects accessible in the environment at a given time step, along with the objects' properties. The environment state transition function $p(\mathcal{E}'|\mathcal{E}, c)$ encodes how the state changes from $\mathcal{E}$ to $\mathcal{E}'$ in response to a command c.[3] During learning, this function is not known, but samples from it can be collected by executing commands and observing the resulting environment state. A real-valued *reward function* measures how well a command sequence $\vec{c}$ achieves the task described in the document.

We posit that a document d is composed of a sequence of instructions, each of which can take one of two forms:

- *Low-level instructions*: these explicitly describe single commands.[4] E.g., "double click system" in Figure 1.
- *High-level instructions*: these correspond to a sequence of one or more environment commands, none of which are explicitly described by the instruction. E.g., "open control panel" in Figure 1.

4 Background

Our innovation takes place within a previously established general framework for the task of mapping instructions to commands (Branavan et al., 2009). This framework formalizes the mapping process as a *Markov Decision Process* (MDP) (Sutton and Barto, 1998), with actions encoding individual instruction-to-command mappings, and states representing partial interpretations of the document. In this section, we review the details of this framework.

[3]While in the general case the environment state transitions maybe stochastic, they are deterministic in the software GUI used in this work.

[4]Previous work (Branavan et al., 2009) is only able to handle low-level instructions.

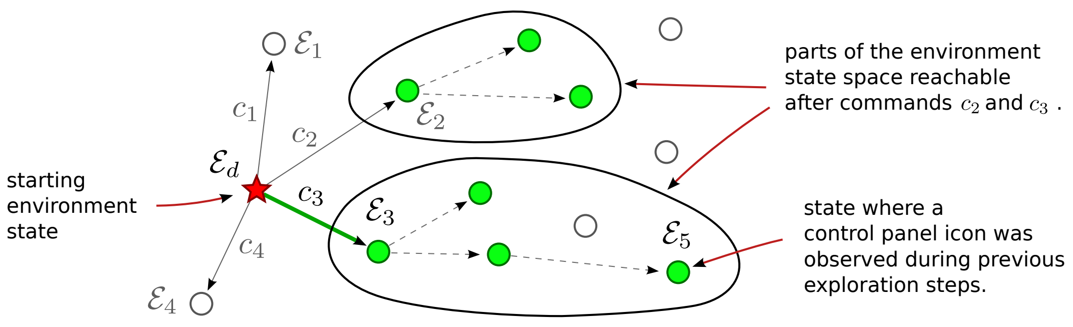
$\mathcal{E}_1$
c_1
c_2
$\mathcal{E}_2$
$\mathcal{E}_d$
c_3
c_4
$\mathcal{E}_3$
$\mathcal{E}_5$
$\mathcal{E}_4$
starting environment state
parts of the environment state space reachable after commands c_2 and c_3 .
state where a control panel icon was observed during previous exploration steps.

Figure 3: Using information derived from future states to interpret the high-level instruction "open control panel." $\mathcal{E}_d$ is the starting state, and c_1 through c_4 are candidate commands. Environment states are shown as circles, with previously visited environment states colored green. Dotted arrows show known state transitions. All else being equal, the information that the control panel icon was observed in state $\mathcal{E}_5$ during previous exploration steps can help to correctly select command c_3.

States and Actions A document is interpreted by incrementally constructing a sequence of actions. Each action selects a word span from the document, and maps it to one environment command. To predict actions sequentially, we track the states of the environment and the document over time as shown in Figure 2. This *mapping state* s is a tuple $(\mathcal{E}, d, W)$ where $\mathcal{E}$ is the current environment state, d is the document being interpreted, and W is the list of word spans selected by previous actions. The mapping state s is observed prior to selecting each action.

The *mapping action* a is a tuple (c, W_a) that represents the joint selection of a span of words W_a and an environment command c. Some of the candidate actions would correspond to the correct instruction mappings, e.g., (c = *double-click* system, W_a = "double click system"). Others such as (c = *left-click* system, W_a = "double click system") would be erroneous. The algorithm learns to interpret instructions by learning to construct sequences of actions that assign the correct commands to the words.

The interpretation of a document d begins at an initial mapping state $s_0 = (\mathcal{E}_d, d, \emptyset)$, $\mathcal{E}_d$ being the starting state of the environment for the document. Given a state $s = (\mathcal{E}, d, W)$, the space of possible actions $a = (c, W_a)$ is defined by enumerating sub-spans of unused words in d and candidate commands in $\mathcal{E}$.[5] The action to execute, a, is selected based on a *policy function* $p(a|s)$ by finding $\arg\max_a p(a|s)$. Performing action a in state $s = (\mathcal{E}, d, W)$ results in a new state s' according to the distribution $p(s'|s, a)$, where:

$$\begin{aligned} a &= (c, W_a), \\ \mathcal{E}' &\sim p(\mathcal{E}'|\mathcal{E}, c), \\ W' &= W \cup W_a, \\ s' &= (\mathcal{E}', d, W'). \end{aligned}$$

The process of selecting and executing actions is repeated until all the words in d have been mapped.[6]

A Log-Linear Parameterization The policy function used for action selection is defined as a log-linear distribution over actions:

$$p(a|s;\theta) = \frac{e^{\theta \cdot \phi(s,a)}}{\sum_{a'} e^{\theta \cdot \phi(s,a')}}, \tag{1}$$

where $\theta \in \mathbb{R}^n$ is a weight vector, and $\phi(s,a) \in \mathbb{R}^n$ is an n-dimensional feature function. This representation has the flexibility to incorporate a variety of features computed on the states and actions.

Reinforcement Learning Parameters of the policy function $p(a|s;\theta)$ are estimated to maximize the expected future reward for analyzing each document $d \in D$:

$$\theta = \arg\max_{\theta} \; E_{p(h|\theta)} \left[r(h) \right], \tag{2}$$

where $h = (s_0, a_0, \ldots, s_{m-1}, a_{m-1}, s_m)$ is a *history* that records the analysis of document d, $p(h|\theta)$ is the probability of selecting this analysis given policy parameters θ, and the reward $r(h)$ is a real valued indication of the quality of h.

[5] Here, command reordering is possible. At each step, the span of selected words W_a is not required to be adjacent to the previous selections. This reordering is used to interpret sentences such as "Select exit after opening the File menu."

[6] To account for document words that are not part of an instruction, c may be a *null* command.

5 Algorithm

We expand the scope of learning approaches for automatic document interpretation by enabling the analysis of *high-level instructions*. The main challenge in processing these instructions is that, in contrast to their low-level counterparts, they correspond to sequences of one or more commands. A simple way to enable this one-to-many mapping is to allow actions that do not consume words (i.e., $|W_a| = 0$). The sequence of actions can then be constructed incrementally using the algorithm described above. However, this change significantly complicates the interpretation problem – we need to be able to predict commands that are not directly described by any words, and allowing action sequences significantly increases the space of possibilities for each instruction. Since we cannot enumerate all possible sequences at decision time, we limit the space of possibilities by learning which sequences are likely to be relevant for the current instruction.

To motivate the approach, consider the decision problem in Figure 3, where we need to find a command sequence for the high-level instruction "open control panel." The algorithm focuses on command sequences leading to environment states where the control panel icon was previously observed. The information about such states is acquired during exploration and is stored in a *partial environment model* $q(\mathcal{E}'|\mathcal{E}, c)$.

Our goal is to map high-level instructions to command sequences by leveraging knowledge about the long-term effects of commands. We do this by integrating the partial environment model into the policy function. Specifically, we modify the log-linear policy $p(a|s; q, \theta)$ by adding *look-ahead features* $\phi(s, a, q)$ which complement the local features used in the previous model. These look-ahead features incorporate various measurements that characterize the potential of future states reachable via the selected action. Although primarily designed to analyze high-level instructions, this approach is also useful for mapping low-level instructions.

Below, we first describe how we estimate the partial environment transition model and how this model is used to compute the look-ahead features. This is followed by the details of parameter estimation for our algorithm.

5.1 Partial Environment Transition Model

To compute the look-ahead features, we first need to collect statistics about the environment transition function $p(\mathcal{E}'|\mathcal{E}, c)$. An example of an environment transition is the change caused by clicking on the "start" button. We collect this information through observation, and build a partial environment transition model $q(\mathcal{E}'|\mathcal{E}, c)$.

One possible strategy for constructing q is to observe the effects of executing random commands in the environment. In a complex environment, however, such a strategy is unlikely to produce state samples relevant to our text analysis task. Instead, we use the training documents to guide the sampling process. During training, we execute the command sequences predicted by the policy function in the environment, caching the resulting state transitions. Initially, these commands may have little connection to the actual instructions. As learning progresses and the quality of the interpretation improves, more promising parts of the environment will be observed. This process yields samples that are biased toward the content of the documents.

5.2 Look-Ahead Features

We wish to select actions that allow for the best follow-up actions, thereby finding the analysis with the highest total reward for a given document. In practice, however, we do not have information about the effects of all possible future actions. Instead, we capitalize on the state transitions observed during the sampling process described above, allowing us to incrementally build an environment model of actions and their effects.

Based on this transition information, we can estimate the usefulness of actions by considering the properties of states they can reach. For instance, some states might have very low immediate reward, indicating that they are unlikely to be part of the best analysis for the document. While the usefulness of most states is hard to determine, it correlates with various properties of the state. We encode the following properties as look-ahead features in our policy:

- The highest reward achievable by an action sequence passing through this state. This property is computed using the learned environment model, and is therefore an approximation.

- The length of the above action sequence.
- The average reward received at the environment state while interpreting any document. This property introduces a bias towards commonly visited states that frequently recur throughout multiple documents' correct interpretations.

Because we can never encounter all states and all actions, our environment model is always incomplete and these properties can only be computed based on partial information. Moreover, the predictive strength of the properties is not known in advance. Therefore we incorporate them as separate features in the model, and allow the learning process to estimate their weights. In particular, we select actions a based on the current state s and the partial environment model q, resulting in the following policy definition:

$$p(a|s;q,\theta) = \frac{e^{\theta\cdot\phi(s,a,q)}}{\sum\limits_{a'} e^{\theta\cdot\phi(s,a',q)}}, \tag{3}$$

where the feature representation $\phi(s,a,q)$ has been extended to be a function of q.

5.3 Parameter Estimation

The learning algorithm is provided with a set of documents $d \in D$, an environment in which to execute command sequences $\vec{c}$, and a reward function $r(h)$. The goal is to estimate two sets of parameters: 1) the parameters θ of the policy function, and 2) the partial environment transition model $q(\mathcal{E}'|\mathcal{E},c)$, which is the observed portion of the true model $p(\mathcal{E}'|\mathcal{E},c)$. These parameters are mutually dependent: θ is defined over a feature space dependent on q, and q is sampled according to the policy function parameterized by θ.

Algorithm 1 shows the procedure for joint learning of these parameters. As in standard policy gradient learning (Sutton et al., 2000), the algorithm iterates over all documents $d \in D$ (steps 1, 2), selecting and executing actions in the environment (steps 3 to 6). The resulting reward is used to update the parameters θ (steps 8, 9). In the new joint learning setting, this process also yields samples of state transitions which are used to estimate $q(\mathcal{E}'|\mathcal{E},c)$ (step 7). This updated q is then used to compute the feature functions $\phi(s,a,q)$ during the next iteration of learning (step 4). This process is repeated until the total reward on training documents converges.

```
Input: A document set $D$,
       Feature function $\phi$,
       Reward function $r(h)$,
       Number of iterations $T$
Initialization: Set $\theta$ to small random values.
                Set $q$ to the empty set.

1  for $i = 1 \cdots T$ do
2    foreach $d \in D$ do
       Sample history $h \sim p(h|\theta)$ where
         $h = (s_0, a_0, \cdots, a_{n-1}, s_n)$ as follows:
       Initialize environment to document specific
       starting state $\mathcal{E}_d$
3      for $t = 0 \cdots n-1$ do
4        Compute $\phi(a, s_t, q)$ based on latest $q$
5        Sample action $a_t \sim p(a|s_t; q, \theta)$
6        Execute $a_t$ on state $s_t$: $s_{t+1} \sim p(s|s_t, a_t)$
7        Set $q = q \cup \{(\mathcal{E}', \mathcal{E}, c)\}$ where $\mathcal{E}', \mathcal{E}, c$ are the
         environment states and commands from $s_{t+1}$,
         $s_t$, and $a_t$
       end
8      $\Delta \leftarrow$
       $\sum_t \left[\phi(s_t, a_t, q) - \sum_{a'} \phi(s_t, a', q)\, p(a'|s_t; q, \theta)\right]$
9      $\theta \leftarrow \theta + r(h)\Delta$
     end
   end
Output: Estimate of parameters $\theta$
```

Algorithm 1: A policy gradient algorithm that also learns a model of the environment.

This algorithm capitalizes on the synergy between θ and q. As learning proceeds, the method discovers a more complete state transition function q, which improves the accuracy of the look-ahead features, and ultimately, the quality of the resulting policy. An improved policy function in turn produces state samples that are more relevant to the document interpretation task.

6 Applying the Model

We apply our algorithm to the task of interpreting help documents to perform software related tasks (Branavan et al., 2009; Kushman et al., 2009). Specifically, we consider documents from Microsoft's Help and Support website.[7] As in prior work, we use a virtual machine set-up to allow our method to interact with a Windows 2000 environment.

Environment States and Actions In this application of our model, the environment state is the set of visible user interface (UI) objects, along

[7]http://support.microsoft.com/

with their properties (e.g., the object's label, parent window, etc). The environment commands consist of the UI commands *left-click*, *right-click*, *double-click*, and *type-into*. Each of these commands requires a UI object as a parameter, while *type-into* needs an additional parameter containing the text to be typed. On average, at each step of the interpretation process, the branching factor is 27.14 commands.

Reward Function An ideal reward function would be to verify whether the task specified by the help document was correctly completed. Since such verification is a challenging task, we rely on a noisy approximation: we assume that each sentence specifies at least one command, and that the text describing the command has words matching the label of the environment object. If a history h has at least one such command for each sentence, the environment reward function $r(h)$ returns a positive value, otherwise it returns a negative value. This environment reward function is a simplification of the one described in Branavan et al. (2009), and it performs comparably in our experiments.

Features In addition to the look-ahead features described in Section 5.2, the policy also includes the set of features used by Branavan et al. (2009). These features are functions of both the text and environment state, modeling local properties that are useful for action selection.

7 Experimental Setup

Datasets Our model is trained on the same dataset used by Branavan et al. (2009). For testing we use two datasets: the first one was used in prior work and contains only low-level instructions, while the second dataset is comprised of documents with high-level instructions. This new dataset was collected from the Microsoft Help and Support website, and has on average 1.03 high-level instructions per document. The second dataset contains 60 test documents, while the first is split into 70, 18 and 40 document for training, development and testing respectively. The combined statistics for these datasets is shown below:

Total # of documents	188
Total # of words	7448
Vocabulary size	739
Avg. actions per document	10

Reinforcement Learning Parameters Following common practice, we encourage exploration during learning with an ϵ-greedy strategy (Sutton and Barto, 1998), with ϵ set to 0.1. We also identify *dead-end* states, i.e. states with the lowest possible immediate reward, and use the induced environment model to encourage additional exploration by lowering the likelihood of actions that lead to such dead-end states.

During the early stages of learning, experience gathered in the environment model is extremely sparse, causing the look-ahead features to provide poor estimates. To speed convergence, we ignore these estimates by disabling the look-ahead features for a fixed number of initial training iterations.

Finally, to guarantee convergence, stochastic gradient ascent algorithms require a learning rate schedule. We use a modified *search-then-converge* algorithm (Darken and Moody, 1990), and tie the learning rate to the ratio of training documents that received a positive reward in the current iteration.

Baselines As a baseline, we compare our method against the results reported by Branavan et al. (2009), denoted here as BCZB09.

As an upper bound for model performance, we also evaluate our method using a reward signal that simulates a fully-supervised training regime. We define a reward function that returns positive one for histories that match the annotations, and zero otherwise. Performing policy-gradient with this function is equivalent to training a fully-supervised, stochastic gradient algorithm that optimizes conditional likelihood (Branavan et al., 2009).

Evaluation Metrics We evaluate the accuracy of the generated mapping by comparing it against manual annotations of the correct action sequences. We measure the percentage of correct actions and the percentage of documents where every action is correct. In general, the sequential nature of the interpretation task makes it difficult to achieve high action accuracy. For example, executing an incorrect action early on, often leads to an environment state from which the remaining instructions cannot be completed. When this happens, it is not possible to recover the remaining actions, causing cascading errors that significantly reduce performance.

	Low-level instruction dataset		High-level instruction dataset		
	action	document	action	high-level action	document
BCZB09	0.647	0.375	0.021	0.022	0.000
BCZB09 + annotation	$*$ 0.756	0.525	0.035	0.022	0.000
Our model	**0.793**	**0.517**	$*$ **0.419**	$*$ **0.615**	$*$ **0.283**
Our model + annotation	0.793	0.650	$*$ 0.357	0.492	0.333

Table 1: Accuracy of the mapping produced by our model, its variants, and the baseline. Values marked with $*$ are statistically significant at $p < 0.01$ compared to the value immediately above it.

8 Results

As shown in Table 1, our model outperforms the baseline on the two datasets, according to all evaluation metrics. In contrast to the baseline, our model can handle high-level instructions, accurately interpreting 62% of them in the second dataset. Every document in this set contains at least one high-level action, which on average, maps to 3.11 environment commands each. The overall action performance on this dataset, however, seems unexpectedly low at 42%. This discrepancy is explained by the fact that in this dataset, high-level instructions are often located towards the beginning of the document. If these initial challenging instructions are not processed correctly, the rest of the actions for the document cannot be interpreted.

As the performance on the first dataset indicates, the new algorithm is also beneficial for processing low-level instructions. The model outperforms the baseline by at least 14%, both in terms of the actions and the documents it can process. Not surprisingly, the best performance is achieved when the new algorithm has access to manually annotated data during training.

We also performed experiments to validate the intuition that the partial environment model must contain information relevant for the language interpretation task. To test this hypothesis, we replaced the learned environment model with one of the same size gathered by executing random commands. The model with randomly sampled environment transitions performs poorly: it can only process 4.6% of documents and 15% of actions on the dataset with high-level instructions, compared to 28.3% and 41.9% respectively for our algorithm. This result also explains why training with full supervision hurts performance on high-level instructions (see Table 1). Learning directly from annotations results in a low-quality environment model due to the relative lack of exploration, hurting the model's ability to leverage the look-ahead features.

High-level instruction
- open **device manager**

Extracted low-level instruction paraphrase
- double click **my computer**
- double click **control panel**
- double click **administrative tools**
- double click **computer management**
- double click **device manager**

High-level instruction
- open the **network tool** in **control panel**

Extracted low-level instruction paraphrase
- click **start**
- point to **settings**
- click **control panel**
- double click **network and dial-up connections**

Figure 4: Examples of automatically generated paraphrases for high-level instructions. The model maps the high-level instruction into a sequence of commands, and then translates them into the corresponding low-level instructions.

Finally, to demonstrate the quality of the learned word–command alignments, we evaluate our method's ability to paraphrase from high-level instructions to low-level instructions. Here, the goal is to take each high-level instruction and construct a text description of the steps required to achieve it. We did this by finding high-level instructions where each of the commands they are associated with is also described by a low-level instruction in some other document. For example, if the text "open control panel" was mapped to the three commands in Figure 1, and each of those commands was described by a low-level instruction elsewhere, this procedure would create a paraphrase such as "click start, left click setting, and select control panel." Of the 60 high-level instructions tagged in the test set, this approach found paraphrases for 33 of them. 29 of these paraphrases were correct, in the sense that they describe all the necessary commands. Figure 4 shows some examples of the automatically extracted paraphrases.

9 Conclusions and Future Work

In this paper, we demonstrate that knowledge about the environment can be learned and used effectively for the task of mapping instructions to actions. A key feature of this approach is the synergy between language analysis and the construction of the environment model: instruction text drives the sampling of the environment transitions, while the acquired environment model facilitates language interpretation. This design enables us to learn to map high-level instructions while also improving accuracy on low-level instructions.

To apply the above method to process a broad range of natural language documents, we need to handle several important semantic and pragmatic phenomena, such as reference, quantification, and conditional statements. These linguistic constructions are known to be challenging to learn – existing approaches commonly rely on large amounts of hand annotated data for training. An interesting avenue of future work is to explore an alternative approach which learns these phenomena by combining linguistic information with knowledge gleaned from an automatically induced environment model.

Acknowledgments

The authors acknowledge the support of the NSF (CAREER grant IIS-0448168, grant IIS-0835445, and grant IIS-0835652) and the Microsoft Research New Faculty Fellowship. Thanks to Aria Haghighi, Leslie Pack Kaelbling, Tom Kwiatkowski, Martin Rinard, David Silver, Mark Steedman, Csaba Szepesvari, the MIT NLP group, and the ACL reviewers for their suggestions and comments. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors, and do not necessarily reflect the views of the funding organizations.

References

Philip E. Agre and David Chapman. 1988. What are plans for? Technical report, Cambridge, MA, USA.

J. A. Boyan and A. W. Moore. 1995. Generalization in reinforcement learning: Safely approximating the value function. In *Advances in NIPS*, pages 369–376.

S.R.K Branavan, Harr Chen, Luke Zettlemoyer, and Regina Barzilay. 2009. Reinforcement learning for mapping instructions to actions. In *Proceedings of ACL*, pages 82–90.

Christian Darken and John Moody. 1990. Note on learning rate schedules for stochastic optimization. In *Advances in NIPS*, pages 832–838.

Barbara Di Eugenio and Michael White. 1992. On the interpretation of natural language instructions. In *Proceedings of COLING*, pages 1147–1151.

Barbara Di Eugenio. 1992. Understanding natural language instructions: the case of purpose clauses. In *Proceedings of ACL*, pages 120–127.

Jacob Eisenstein, James Clarke, Dan Goldwasser, and Dan Roth. 2009. Reading to learn: Constructing features from semantic abstracts. In *Proceedings of EMNLP*, pages 958–967.

Michael Fleischman and Deb Roy. 2005. Intentional context in situated natural language learning. In *Proceedings of CoNLL*, pages 104–111.

Nicholas K. Jong and Peter Stone. 2007. Model-based function approximation in reinforcement learning. In *Proceedings of AAMAS*, pages 670–677.

Nate Kushman, Micah Brodsky, S.R.K. Branavan, Dina Katabi, Regina Barzilay, and Martin Rinard. 2009. Wikido. In *Proceedings of HotNets-VIII*.

Alex Lascarides and Nicholas Asher. 2004. Imperatives in dialogue. In P. Kuehnlein, H. Rieser, and H. Zeevat, editors, *The Semantics and Pragmatics of Dialogue for the New Millenium*. Benjamins.

Oliver Lemon and Ioannis Konstas. 2009. User simulations for context-sensitive speech recognition in spoken dialogue systems. In *Proceedings of EACL*, pages 505–513.

Percy Liang, Michael I. Jordan, and Dan Klein. 2009. Learning semantic correspondences with less supervision. In *Proceedings of ACL*, pages 91–99.

Matt MacMahon, Brian Stankiewicz, and Benjamin Kuipers. 2006. Walk the talk: connecting language, knowledge, and action in route instructions. In *Proceedings of AAAI*, pages 1475–1482.

C. Matuszek, D. Fox, and K. Koscher. 2010. Following directions using statistical machine translation. In *Proceedings of Human-Robot Interaction*, pages 251–258.

Raymond J. Mooney. 2008. Learning to connect language and perception. In *Proceedings of AAAI*, pages 1598–1601.

James Timothy Oates. 2001. *Grounding knowledge in sensors: Unsupervised learning for language and planning*. Ph.D. thesis, University of Massachusetts Amherst.

Warren B Powell. 2007. *Approximate Dynamic Programming*. Wiley-Interscience.

Jost Schatzmann and Steve Young. 2009. The hidden agenda user simulation model. *IEEE Trans. Audio, Speech and Language Processing*, 17(4):733–747.

Satinder Singh, Diane Litman, Michael Kearns, and Marilyn Walker. 2002. Optimizing dialogue management with reinforcement learning: Experiments with the njfun system. *Journal of Artificial Intelligence Research*, 16:105–133.

Jeffrey Mark Siskind. 2001. Grounding the lexical semantics of verbs in visual perception using force dynamics and event logic. *Journal of Artificial Intelligence Research*, 15:31–90.

Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement Learning: An Introduction*. The MIT Press.

Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 2000. Policy gradient methods for reinforcement learning with function approximation. In *Advances in NIPS*, pages 1057–1063.

Bonnie Webber, Norman Badler, Barbara Di Eugenio, Libby Levison Chris Geib, and Michael Moore. 1995. Instructions, intentions and expectations. *Artificial Intelligence*, 73(1-2).

Terry Winograd. 1972. *Understanding Natural Language*. Academic Press.

Chen Yu and Dana H. Ballard. 2004. On the integration of grounding language and learning objects. In *Proceedings of AAAI*, pages 488–493.