Reinforcement Learning for Mapping Instructions to Actions

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

Abstract

In this paper, we present a reinforcement learning approach for mapping natural language instructions to sequences of executable actions. We assume access to a reward function that defines the quality of the executed actions. During training, the learner repeatedly constructs action sequences for a set of documents, executes those actions, and observes the resulting reward. We use a policy gradient algorithm to estimate the parameters of a log-linear model for action selection. We apply our method to interpret instructions in two domains — Windows troubleshooting guides and game tutorials. Our results demonstrate that this method can rival supervised learning techniques while requiring few or no annotated training examples.[1]

1 Introduction

The problem of interpreting instructions written in natural language has been widely studied since the early days of artificial intelligence (Winograd, 1972; Di Eugenio, 1992). Mapping instructions to a sequence of executable actions would enable the automation of tasks that currently require human participation. Examples include configuring software based on how-to guides and operating simulators using instruction manuals. In this paper, we present a reinforcement learning framework for inducing mappings from text to actions without the need for annotated training examples.

For concreteness, consider instructions from a Windows troubleshooting guide on deleting temporary folders, shown in Figure 1. We aim to map this text to the corresponding low-level commands and parameters. For example, properly interpreting the third instruction requires clicking on a tab, finding the appropriate option in a tree control, and clearing its associated checkbox.

- Click start, point to search, and then click for files or folders.
- In the search results dialog box, on the tools menu, click folder options.
- In the folder options dialog box, on the view tab, under advanced settings, click *show hidden files and folders*, and then click to clear the *hide file extensions for known file types* check box.
- Click apply, and then click ok.
- In the search for files or folders named box, type msdownld.tmp.
- In the look in list, click my computer, and then click search now.
- In the search results pane, right-click msdownld.tmp and then click delete on the shortcut menu, a *confirm folder delete* message appears.
- Click yes.

Figure 1: A Windows troubleshooting article describing how to remove the "msdownld.tmp" temporary folder.

In this and many other applications, the validity of a mapping can be verified by executing the induced actions in the corresponding environment and observing their effects. For instance, in the example above we can assess whether the goal described in the instructions is achieved, i.e., the folder is deleted. The key idea of our approach is to leverage the validation process as the main source of supervision to guide learning. This form of supervision allows us to learn interpretations of natural language instructions when standard supervised techniques are not applicable, due to the lack of human-created annotations.

Reinforcement learning is a natural framework for building models using validation from an environment (Sutton and Barto, 1998). We assume that supervision is provided in the form of a reward function that defines the quality of executed actions. During training, the learner repeatedly constructs action sequences for a set of given documents, executes those actions, and observes the resulting reward. The learner's goal is to estimate a

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

policy — a distribution over actions given instruction text and environment state — that maximizes future expected reward. Our policy is modeled in a log-linear fashion, allowing us to incorporate features of both the instruction text and the environment. We employ a policy gradient algorithm to estimate the parameters of this model.

We evaluate our method on two distinct applications: Windows troubleshooting guides and puzzle game tutorials. The key findings of our experiments are twofold. First, models trained only with simple reward signals achieve surprisingly high results, coming within 11% of a fully supervised method in the Windows domain. Second, augmenting unlabeled documents with even a small fraction of annotated examples greatly reduces this performance gap, to within 4% in that domain. These results indicate the power of learning from this new form of automated supervision.

2 Related Work

Grounded Language Acquisition Our work fits into a broader class of approaches that aim to learn language from a situated context (Mooney, 2008a; Mooney, 2008b; Fleischman and Roy, 2005; Yu and Ballard, 2004; Siskind, 2001; Oates, 2001). Instances of such approaches include work on inferring the meaning of words from video data (Roy and Pentland, 2002; Barnard and Forsyth, 2001), and interpreting the commentary of a simulated soccer game (Chen and Mooney, 2008). Most of these approaches assume some form of parallel data, and learn perceptual co-occurrence patterns. In contrast, our emphasis is on learning language by proactively interacting with an external environment.

Reinforcement Learning for Language Processing Reinforcement learning has been previously applied to the problem of dialogue management (Scheffler and Young, 2002; Roy et al., 2000; Litman et al., 2000; Singh et al., 1999). These systems converse with a human user by taking actions that emit natural language utterances. The reinforcement learning state space encodes information about the goals of the user and what they say at each time step. The learning problem is to find an optimal policy that maps states to actions, through a trial-and-error process of repeated interaction with the user.

Reinforcement learning is applied very differently in dialogue systems compared to our setup. In some respects, our task is more easily amenable to reinforcement learning. For instance, we are not interacting with a human user, so the cost of interaction is lower. However, while the state space can be designed to be relatively small in the dialogue management task, our state space is determined by the underlying environment and is typically quite large. We address this complexity by developing a policy gradient algorithm that learns efficiently while exploring a small subset of the states.

3 Problem Formulation

Our task is to learn a mapping between documents and the sequence of actions they express. Figure 2 shows how one example sentence is mapped to three actions.

Mapping Text to Actions As input, we are given a document d, comprising a sequence of sentences $(u_1, \ldots, u_\ell)$, where each u_i is a sequence of words. Our goal is to map d to a sequence of actions $\vec{a} = (a_0, \ldots, a_{n-1})$. Actions are predicted and executed sequentially.[2]

An action $a = (c, R, W')$ encompasses a *command* c, the command's *parameters* R, and the words W' specifying c and R. Elements of R refer to *objects* available in the *environment state*, as described below. Some parameters can also refer to words in document d. Additionally, to account for words that do not describe any actions, c can be a null command.

The Environment The environment state $\mathcal{E}$ specifies the set of objects available for interaction, and their properties. In Figure 2, $\mathcal{E}$ is shown on the right. The environment state $\mathcal{E}$ changes in response to the execution of command c with parameters R according to a transition distribution $p(\mathcal{E}'|\mathcal{E}, c, R)$. This distribution is *a priori* unknown to the learner. As we will see in Section 5, our approach avoids having to directly estimate this distribution.

State To predict actions sequentially, we need to track the state of the document-to-actions mapping over time. A *mapping state* s is a tuple $(\mathcal{E}, d, j, W)$, where $\mathcal{E}$ refers to the current environment state; j is the index of the sentence currently being interpreted in document d; and W contains words that were mapped by previous actions for

[2] That is, action a_i is executed before a_{i+1} is predicted.

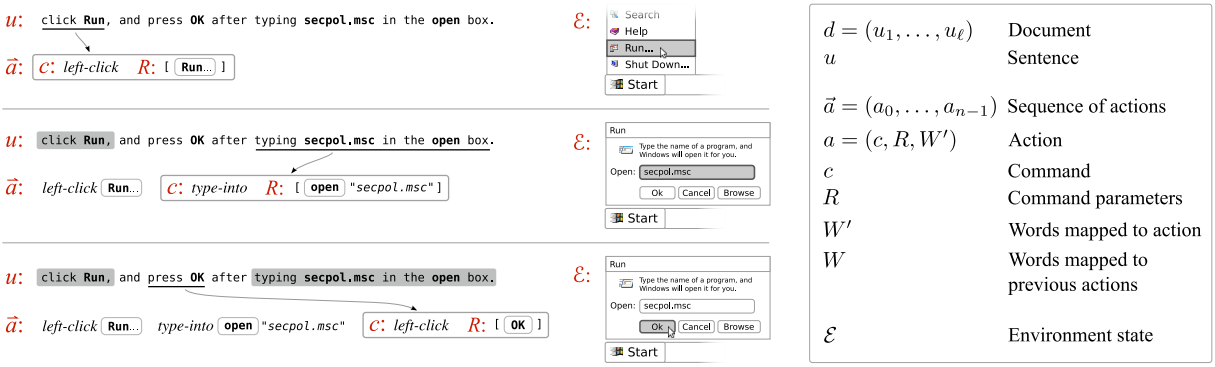


Figure 2: A three-step mapping from an instruction sentence to a sequence of actions in Windows 2000. For each step, the figure shows the words selected by the action, along with the corresponding system command and its parameters. The words of W' are underlined, and the words of W are highlighted in grey.

the same sentence. The mapping state s is observed after each action.

The initial mapping state s_0 for document d is $(\mathcal{E}_d, d, 0, \emptyset)$; $\mathcal{E}_d$ is the unique starting environment state for d. Performing action a in state $s = (\mathcal{E}, d, j, W)$ leads to a new state s' according to distribution $p(s'|s, a)$, defined as follows: $\mathcal{E}$ transitions according to $p(\mathcal{E}'|\mathcal{E}, c, R)$, W is updated with a's selected words, and j is incremented if all words of the sentence have been mapped. For the applications we consider in this work, environment state transitions, and consequently mapping state transitions, are deterministic.

Training During training, we are provided with a set D of documents, the ability to sample from the transition distribution, and a *reward function* $r(h)$. Here, $h = (s_0, a_0, \ldots, s_{n-1}, a_{n-1}, s_n)$ is a *history* of states and actions visited while interpreting one document. $r(h)$ outputs a real-valued score that correlates with correct action selection.[3] We consider both immediate reward, which is available after each action, and delayed reward, which does not provide feedback until the last action. For example, *task completion* is a delayed reward that produces a positive value after the final action only if the task was completed successfully. We will also demonstrate how manually annotated action sequences can be incorporated into the reward.

[3]In most reinforcement learning problems, the reward function is defined over state-action pairs, as $r(s, a)$ — in this case, $r(h) = \sum_t r(s_t, a_t)$, and our formulation becomes a standard finite-horizon Markov decision process. Policy gradient approaches allow us to learn using the more general case of history-based reward.

The goal of training is to estimate parameters θ of the action selection distribution $p(a|s, \theta)$, called the *policy*. Since the reward correlates with action sequence correctness, the θ that maximizes expected reward will yield the best actions.

4 A Log-Linear Model for Actions

Our goal is to predict a sequence of actions. We construct this sequence by repeatedly choosing an action given the current mapping state, and applying that action to advance to a new state.

Given a state $s = (\mathcal{E}, d, j, W)$, the space of possible next actions is defined by enumerating subspans of unused words in the current sentence (i.e., subspans of the jth sentence of d not in W), and the possible commands and parameters in environment state $\mathcal{E}$.[4] We model the policy distribution $p(a|s; \theta)$ over this action space in a log-linear fashion (Della Pietra et al., 1997; Lafferty et al., 2001), giving us the flexibility to incorporate a diverse range of features. Under this representation, the policy distribution is:

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

where $\phi(s, a) \in \mathbb{R}^n$ is an n-dimensional feature representation. During test, actions are selected according to the mode of this distribution.

[4]For parameters that refer to words, the space of possible values is defined by the unused words in the current sentence.

5 Reinforcement Learning

During training, our goal is to find the optimal policy $p(a|s;\theta)$. Since reward correlates with correct action selection, a natural objective is to maximize expected future reward — that is, the reward we expect while acting according to that policy from state s. Formally, we maximize the *value function*:

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

where the history h is the sequence of states and actions encountered while interpreting a single document $d \in D$. This expectation is averaged over all documents in D. The distribution $p(h|\theta)$ returns the probability of seeing history h when starting from state s and acting according to a policy with parameters θ. This distribution can be decomposed into a product over time steps:

$$p(h|\theta) = \prod_{t=0}^{n-1} p(a_t|s_t;\theta)p(s_{t+1}|s_t,a_t). \tag{3}$$

5.1 A Policy Gradient Algorithm

Our reinforcement learning problem is to find the parameters θ that maximize V_θ from equation 2. Although there is no closed form solution, *policy gradient* algorithms (Sutton et al., 2000) estimate the parameters θ by performing stochastic gradient ascent. The gradient of V_θ is approximated by interacting with the environment, and the resulting reward is used to update the estimate of θ. Policy gradient algorithms optimize a non-convex objective and are only guaranteed to find a local optimum. However, as we will see, they scale to large state spaces and can perform well in practice.

To find the parameters θ that maximize the objective, we first compute the derivative of V_θ. Expanding according to the product rule, we have:

$$\frac{\partial}{\partial\theta}V_\theta(s) = E_{p(h|\theta)}\left[r(h)\sum_t \frac{\partial}{\partial\theta}\log p(a_t|s_t;\theta)\right], \tag{4}$$

where the inner sum is over all time steps t in the current history h. Expanding the inner partial derivative we observe that:

$$\frac{\partial}{\partial\theta}\log p(a|s;\theta) = \phi(s,a) - \sum_{a'}\phi(s,a')p(a'|s;\theta), \tag{5}$$

which is the derivative of a log-linear distribution.

Equation 5 is easy to compute directly. However, the complete derivative of V_θ in equation 4 is intractable, because computing the expectation would require summing over all possible histories. Instead, policy gradient algorithms employ stochastic gradient ascent by computing a noisy estimate of the expectation using just a subset of the histories. Specifically, we draw samples from $p(h|\theta)$ by acting in the target environment, and use these samples to approximate the expectation in equation 4. In practice, it is often sufficient to sample a single history h for this approximation.

Input: A document set D,
Feature representation ϕ,
Reward function $r(h)$,
Number of iterations T

Initialization: Set θ to small random values.

1 **for** $i = 1 \dots T$ **do**
2 **foreach** $d \in D$ **do**
3 Sample history $h \sim p(h|\theta)$ where $h = (s_0, a_0, \dots, a_{n-1}, s_n)$ as follows:
3a **for** $t = 0 \dots n-1$ **do**
3b Sample action $a_t \sim p(a|s_t;\theta)$
3c Execute a_t on state s_t: $s_{t+1} \sim p(s|s_t,a_t)$
  end
4 $\Delta \leftarrow \sum_t \left(\phi(s_t,a_t) - \sum_{a'}\phi(s_t,a')p(a'|s_t;\theta)\right)$
5 $\theta \leftarrow \theta + r(h)\Delta$
  end
end

Output: Estimate of parameters θ

Algorithm 1: A policy gradient algorithm.

Algorithm 1 details the complete policy gradient algorithm. It performs T iterations over the set of documents D. Step 3 samples a history that maps each document to actions. This is done by repeatedly selecting actions according to the current policy, and updating the state by executing the selected actions. Steps 4 and 5 compute the empirical gradient and update the parameters θ.

In many domains, interacting with the environment is expensive. Therefore, we use two techniques that allow us to take maximum advantage of each environment interaction. First, a history $h = (s_0, a_0, \dots, s_n)$ contains subsequences $(s_i, a_i, \dots s_n)$ for $i = 1$ to $n-1$, each with its own reward value given by the environment as a side effect of executing h. We apply the update from equation 5 for each subsequence. Second, for a sampled history h, we can propose alternative histories h' that result in the same commands and parameters with different word spans. We can again apply equation 5 for each h', weighted by its probability under the current policy, $\frac{p(h'|\theta)}{p(h|\theta)}$.

The algorithm we have presented belongs to a family of policy gradient algorithms that have been successfully used for complex tasks such as robot control (Ng et al., 2003). Our formulation is unique in how it represents natural language in the reinforcement learning framework.

5.2 Reward Functions and ML Estimation

We can design a range of reward functions to guide learning, depending on the availability of annotated data and environment feedback. Consider the case when every training document $d \in D$ is annotated with its correct sequence of actions, and state transitions are deterministic. Given these examples, it is straightforward to construct a reward function that connects policy gradient to maximum likelihood. Specifically, define a reward function $r(h)$ that returns one when h matches the annotation for the document being analyzed, and zero otherwise. Policy gradient performs stochastic gradient ascent on the objective from equation 2, performing one update per document. For document d, this objective becomes:

$$E_{p(h|\theta)}[r(h)] = \sum_h r(h)p(h|\theta) = p(h_d|\theta),$$

where h_d is the history corresponding to the annotated action sequence. Thus, with this reward policy gradient is equivalent to stochastic gradient ascent with a maximum likelihood objective.

At the other extreme, when annotations are completely unavailable, learning is still possible given informative feedback from the environment. Crucially, this feedback only needs to correlate with action sequence quality. We detail environment-based reward functions in the next section. As our results will show, reward functions built using this kind of feedback can provide strong guidance for learning. We will also consider reward functions that combine annotated supervision with environment feedback.

6 Applying the Model

We study two applications of our model: following instructions to perform software tasks, and solving a puzzle game using tutorial guides.

6.1 Microsoft Windows Help and Support

On its Help and Support website,[5] Microsoft publishes a number of articles describing how to perform tasks and troubleshoot problems in the Windows operating systems. Examples of such tasks include installing patches and changing security settings. Figure 1 shows one such article.

[5] support.microsoft.com

Notation	
o	Parameter referring to an environment object
L	Set of object class names (e.g. "button")
V	Vocabulary
Features on W and object o	
Test if o is visible in s	
Test if o has input focus	
Test if o is in the foreground	
Test if o was previously interacted with	
Test if o came into existence since last action	
Min. edit distance between $w \in W$ and object labels in s	
Features on words in W, command c, and object o	
$\forall c' \in C, w \in V$: test if $c' = c$ and $w \in W$	
$\forall c' \in C, l \in L$: test if $c' = c$ and l is the class of o	

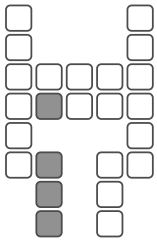
Table 1: Example features in the Windows domain. All features are binary, except for the normalized edit distance which is real-valued.

Our goal is to automatically execute these support articles in the Windows 2000 environment. Here, the environment state is the set of visible user interface (UI) objects, and object properties such as label, location, and parent window. Possible commands include *left-click*, *right-click*, *double-click*, and *type-into*, all of which take a UI object as a parameter; *type-into* additionally requires a parameter for the input text.

Table 1 lists some of the features we use for this domain. These features capture various aspects of the action under consideration, the current Windows UI state, and the input instructions. For example, one lexical feature measures the similarity of a word in the sentence to the UI labels of objects in the environment. Environment-specific features, such as whether an object is currently in focus, are useful when selecting the object to manipulate. In total, there are 4,438 features.

Reward Function Environment feedback can be used as a reward function in this domain. An obvious reward would be task completion (e.g., whether the stated computer problem was fixed). Unfortunately, verifying task completion is a challenging system issue in its own right.

Instead, we rely on a noisy method of checking whether execution can proceed from one sentence to the next: at least one word in each sentence has to correspond to an object in the envi-



Clear the bottom four from the second column from the left, the row of four, the second column from the right, the row of four, then the two columns.

Figure 3: Crossblock puzzle with tutorial. For this level, four squares in a row or column must be removed at once. The first move specified by the tutorial is greyed in the puzzle.

ronment.[6] For instance, in the sentence from Figure 2 the word "Run" matches the *Run...* menu item. If no words in a sentence match a current environment object, then one of the previous sentences was analyzed incorrectly. In this case, we assign the history a reward of -1. This reward is not guaranteed to penalize all incorrect histories, because there may be false positive matches between the sentence and the environment. When at least one word matches, we assign a positive reward that linearly increases with the percentage of words assigned to non-null commands, and linearly decreases with the number of output actions. This reward signal encourages analyses that interpret all of the words without producing spurious actions.

6.2 Crossblock: A Puzzle Game

Our second application is to a puzzle game called *Crossblock*, available online as a Flash game.[7] Each of 50 puzzles is played on a grid, where some grid positions are filled with squares. The object of the game is to clear the grid by drawing vertical or horizontal line segments that remove groups of squares. Each segment must exactly cross a specific number of squares, ranging from two to seven depending on the puzzle. Humans players have found this game challenging and engaging enough to warrant posting textual tutorials.[8] A sample puzzle and tutorial are shown in Figure 3.

The environment is defined by the state of the grid. The only command is *clear*, which takes a parameter specifying the orientation (*row* or *column*) and grid location of the line segment to be removed. The challenge in this domain is to segment the text into the phrases describing each action, and then correctly identify the line segments from references such as "the bottom four from the second column from the left."

For this domain, we use two sets of binary features on state-action pairs (s, a). First, for each vocabulary word w, we define a feature that is one if w is the last word of a's consumed words W'. These features help identify the proper text segmentation points between actions. Second, we introduce features for pairs of vocabulary word w and attributes of action a, e.g., the line orientation and grid locations of the squares that a would remove. This set of features enables us to match words (e.g., "row") with objects in the environment (e.g., a move that removes a horizontal series of squares). In total, there are 8,094 features.

Reward Function For Crossblock it is easy to directly verify task completion, which we use as the basis of our reward function. The reward $r(h)$ is -1 if h ends in a state where the puzzle cannot be completed. For solved puzzles, the reward is a positive value proportional to the percentage of words assigned to non-null commands.

7 Experimental Setup

Datasets For the Windows domain, our dataset consists of 128 documents, divided into 70 for training, 18 for development, and 40 for test. In the puzzle game domain, we use 50 tutorials, divided into 40 for training and 10 for test.[9] Statistics for the datasets are shown below.

	Windows	Puzzle
Total # of documents	128	50
Total # of words	5562	994
Vocabulary size	610	46
Avg. words per sentence	9.93	19.88
Avg. sentences per document	4.38	1.00
Avg. actions per document	10.37	5.86

The data exhibits certain qualities that make for a challenging learning problem. For instance, there are a surprising variety of linguistic constructs — as Figure 4 shows, in the Windows domain even a simple command is expressed in at least six different ways.

[6] We assume that a word maps to an environment object if the edit distance between the word and the object's name is below a threshold value.

[7] hexaditidom.deviantart.com/art/Crossblock-108669149

[8] www.jayisgames.com/archives/2009/01/crossblock.php

[9] For Crossblock, because the number of puzzles is limited, we did not hold out a separate development set, and report averaged results over five training/test splits.

```
On the tools menu, click internet options
Click tools, and then click internet options
Click tools, and then choose internet options
Click internet options on the tools menu
In internet explorer, click internet options on the tools menu
On the tools menu in internet explorer, click internet options
```

Figure 4: Variations of "click internet options on the tools menu" present in the Windows corpus.

Experimental Framework To apply our algorithm to the Windows domain, we use the Win32 application programming interface to simulate human interactions with the user interface, and to gather environment state information. The operating system environment is hosted within a virtual machine,[10] allowing us to rapidly save and reset system state snapshots. For the puzzle game domain, we replicated the game with an implementation that facilitates automatic play.

As is commonly done in reinforcement learning, we use a softmax temperature parameter to smooth the policy distribution (Sutton and Barto, 1998), set to 0.1 in our experiments. For Windows, the development set is used to select the best parameters. For Crossblock, we choose the parameters that produce the highest reward during training. During evaluation, we use these parameters to predict mappings for the test documents.

Evaluation Metrics For evaluation, we compare the results to manually constructed sequences of actions. We measure the number of correct actions, sentences, and documents. An action is correct if it matches the annotations in terms of command and parameters. A sentence is correct if all of its actions are correctly identified, and analogously for documents.[11] Statistical significance is measured with the sign test.

Additionally, we compute a word alignment score to investigate the extent to which the input text is used to construct correct analyses. This score measures the percentage of words that are aligned to the corresponding annotated actions in correctly analyzed documents.

Baselines We consider the following baselines to characterize the performance of our approach.

- **Full Supervision** Sequence prediction problems like ours are typically addressed using supervised techniques. We measure how a standard supervised approach would perform on this task by using a reward signal based on manual annotations of output action sequences, as defined in Section 5.2. As shown there, policy gradient with this reward is equivalent to stochastic gradient ascent with a maximum likelihood objective.

- **Partial Supervision** We consider the case when only a subset of training documents is annotated, and environment reward is used for the remainder. Our method seamlessly combines these two kinds of rewards.

- **Random and Majority (Windows)** We consider two naïve baselines. Both scan through each sentence from left to right. A command c is executed on the object whose name is encountered first in the sentence. This command c is either selected *randomly*, or set to the *majority* command, which is *left-click*. This procedure is repeated until no more words match environment objects.

- **Random (Puzzle)** We consider a baseline that *randomly* selects among the actions that are valid in the current game state.[12]

8 Results

Table 2 presents evaluation results on the test sets. There are several indicators of the difficulty of this task. The random and majority baselines' poor performance in both domains indicates that naïve approaches are inadequate for these tasks. The performance of the fully supervised approach provides further evidence that the task is challenging. This difficulty can be attributed in part to the large branching factor of possible actions at each step — on average, there are 27.14 choices per action in the Windows domain, and 9.78 in the Crossblock domain.

In both domains, the learners relying only on environment reward perform well. Although the fully supervised approach performs the best, adding just a few annotated training examples to the environment-based learner significantly reduces the performance gap.

[10] VMware Workstation, available at www.vmware.com

[11] In these tasks, each action depends on the correct execution of all previous actions, so a single error can render the remainder of that document's mapping incorrect. In addition, due to variability in document lengths, overall action accuracy is not guaranteed to be higher than document accuracy.

[12] Since action selection is among objects, there is no natural majority baseline for the puzzle.

	Windows				Puzzle		
	Action	Sent.	Doc.	Word	Action	Doc.	Word
Random baseline	0.128	0.101	0.000	—	0.081	0.111	—
Majority baseline	0.287	0.197	0.100	—	—	—	—
Environment reward	$*$ 0.647	$*$ 0.590	$*$ 0.375	0.819	$*$ 0.428	$*$ 0.453	0.686
Partial supervision	$\diamond$ 0.723	$*$ 0.702	0.475	0.989	0.575	$*$ 0.523	0.850
Full supervision	$\diamond$ 0.756	0.714	0.525	0.991	0.632	0.630	0.869

Table 2: Performance on the test set with different reward signals and baselines. Our evaluation measures the proportion of correct actions, sentences, and documents. We also report the percentage of correct word alignments for the successfully completed documents. Note the puzzle domain has only single-sentence documents, so its sentence and document scores are identical. The partial supervision line refers to 20 out of 70 annotated training documents for Windows, and 10 out of 40 for the puzzle. Each result marked with $*$ or $\diamond$ is a statistically significant improvement over the result immediately above it; $*$ indicates $p < 0.01$ and $\diamond$ indicates $p < 0.05$.

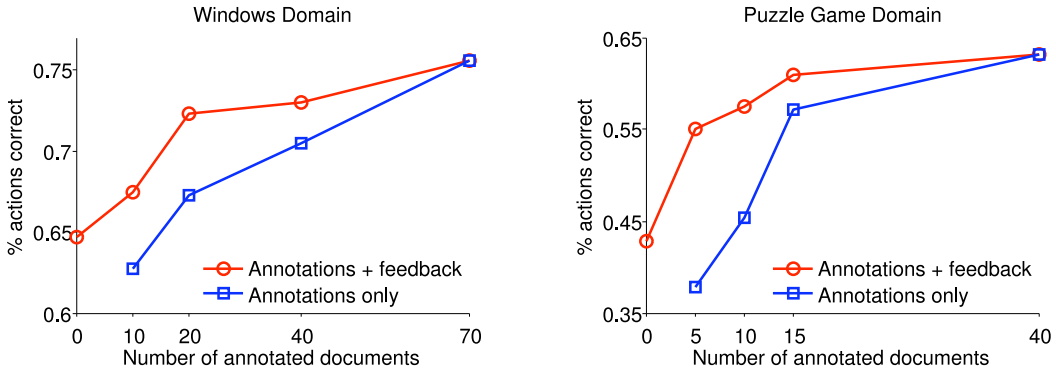


Figure 5: Comparison of two training scenarios where training is done using a subset of annotated documents, with and without environment reward for the remaining unannotated documents.

Figure 5 shows the overall tradeoff between annotation effort and system performance for the two domains. The ability to make this tradeoff is one of the advantages of our approach. The figure also shows that augmenting annotated documents with additional environment-reward documents invariably improves performance.

The word alignment results from Table 2 indicate that the learners are mapping the correct words to actions for documents that are successfully completed. For example, the models that perform best in the Windows domain achieve nearly perfect word alignment scores.

To further assess the contribution of the instruction text, we train a variant of our model without access to text features. This is possible in the game domain, where all of the puzzles share a single goal state that is independent of the instructions. This variant solves 34% of the puzzles, suggesting that access to the instructions significantly improves performance.

9 Conclusions

In this paper, we presented a reinforcement learning approach for inducing a mapping between instructions and actions. This approach is able to use environment-based rewards, such as task completion, to learn to analyze text. We showed that having access to a suitable reward function can significantly reduce the need for annotations.

Acknowledgments

The authors acknowledge the support of the NSF (CAREER grant IIS-0448168, grant IIS-0835445, grant IIS-0835652, and a Graduate Research Fellowship) and the ONR. Thanks to Michael Collins, Amir Globerson, Tommi Jaakkola, Leslie Pack Kaelbling, Dina Katabi, Martin Rinard, and members of the MIT NLP group 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

Kobus Barnard and David A. Forsyth. 2001. Learning the semantics of words and pictures. In *Proceedings of ICCV*.

David L. Chen and Raymond J. Mooney. 2008. Learning to sportscast: a test of grounded language acquisition. In *Proceedings of ICML*.

Stephen Della Pietra, Vincent J. Della Pietra, and John D. Lafferty. 1997. Inducing features of random fields. *IEEE Trans. Pattern Anal. Mach. Intell.*, 19(4):380–393.

Barbara Di Eugenio. 1992. Understanding natural language instructions: the case of purpose clauses. In *Proceedings of ACL*.

Michael Fleischman and Deb Roy. 2005. Intentional context in situated language learning. In *Proceedings of CoNLL*.

John Lafferty, Andrew McCallum, and Fernando Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of ICML*.

Diane J. Litman, Michael S. Kearns, Satinder Singh, and Marilyn A. Walker. 2000. Automatic optimization of dialogue management. In *Proceedings of COLING*.

Raymond J. Mooney. 2008a. Learning language from its perceptual context. In *Proceedings of ECML/PKDD*.

Raymond J. Mooney. 2008b. Learning to connect language and perception. In *Proceedings of AAAI*.

Andrew Y. Ng, H. Jin Kim, Michael I. Jordan, and Shankar Sastry. 2003. Autonomous helicopter flight via reinforcement learning. In *Advances in NIPS*.

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

Deb K. Roy and Alex P. Pentland. 2002. Learning words from sights and sounds: a computational model. *Cognitive Science 26*, pages 113–146.

Nicholas Roy, Joelle Pineau, and Sebastian Thrun. 2000. Spoken dialogue management using probabilistic reasoning. In *Proceedings of ACL*.

Konrad Scheffler and Steve Young. 2002. Automatic learning of dialogue strategy using dialogue simulation and reinforcement learning. In *Proceedings of HLT*.

Satinder P. Singh, Michael J. Kearns, Diane J. Litman, and Marilyn A. Walker. 1999. Reinforcement learning for spoken dialogue systems. In *Advances in NIPS*.

Jeffrey Mark Siskind. 2001. Grounding the lexical semantics of verbs in visual perception using force dynamics and event logic. *J. Artif. Intell. Res. (JAIR)*, 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*.

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*.