# Pre-Learning Environment Representations for Data-Efficient Neural Instruction Following

**David Gaddy** and **Dan Klein**
Computer Science Division
University of California, Berkeley
{dgaddy,klein}@berkeley.edu

## Abstract

We consider the problem of learning to map from natural language instructions to state transitions (actions) in a data-efficient manner. Our method takes inspiration from the idea that it should be easier to ground language to concepts that have already been formed through pre-linguistic observation. We augment a baseline instruction-following learner with an initial environment-learning phase that uses observations of language-free state transitions to induce a suitable latent representation of actions before processing the instruction-following training data. We show that mapping to pre-learned representations substantially improves performance over systems whose representations are learned from limited instructional data alone.

## 1 Introduction

In the past several years, neural approaches have become increasingly central to the instruction following literature (e.g. Misra et al., 2018; Chaplot et al., 2018; Mei et al., 2016). However, neural networks' powerful abilities to induce complex representations have come at the cost of data efficiency. Indeed, compared to earlier logical form-based methods, neural networks can sometimes require orders of magnitude more data. The data-hungriness of neural approaches is not surprising – starting with classic logical forms improves data efficiency by presenting a system with pre-made abstractions, where end-to-end neural approaches must do the hard work of inducing abstractions on their own. In this paper, we aim to combine the power of neural networks with the data-efficiency of logical forms by pre-learning abstractions in a semi-supervised way, satiating part of the network's data hunger on cheaper unlabeled data from the environment.

When neural nets have only limited data that pairs language with actions, they suffer from a lack of inductive bias, fitting the training data but generalizing in ways that seem nonsensical to humans. For example, a neural network given the transition shown in Figure 1 might map the corresponding instruction to an adequate but unlikely meaning that red blocks should be stacked to the right of blue blocks except for on brown blocks. The inspiration for this work comes from the idea that humans avoid spurious hypotheses like this example partly because they have already formed a set of useful concepts about their environment before learning language (Bloom, 2000; Hespos and Spelke, 2004). These pre-linguistic abstractions then constrain language learning and help generalization.

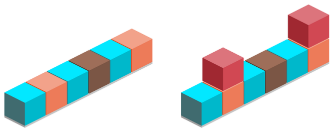
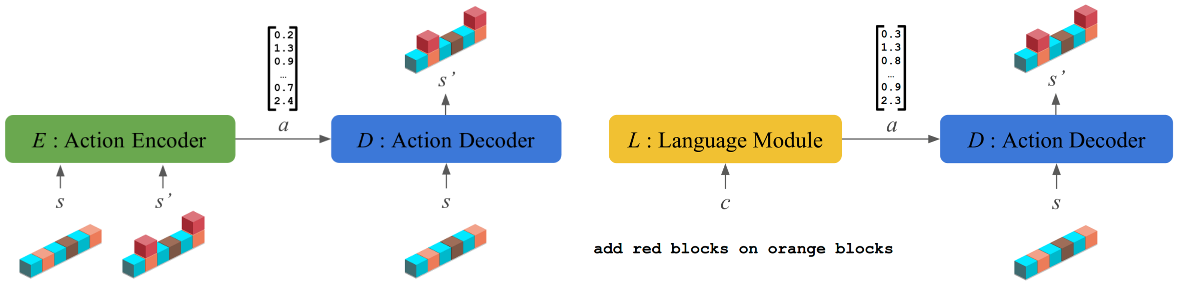


Figure 1: After seeing this transition, a neural net might generalize this action as *stack red blocks to the right of blue blocks except for on brown blocks*, but a generalization like *stack red blocks on orange blocks* is more plausible and generally applicable. We aim to guide our model towards more plausible generalizations by pre-learning inductive biases from observations of the environment.

With this view in mind, we allow our instruction following agent to observe the environment and build a representation of it prior to seeing any linguistic instructions. In particular, we adopt a semi-supervised setup with two phases, as shown in Figure 2: an *environment learning* phase where the system sees samples of language-free state transitions from actions in the environment, and a *language learning* phase where instructions are given along with their corresponding effects on the en-

*Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1946–1956
Florence, Italy, July 28 - August 2, 2019.



(i) Environment learning

(ii) Language learning

Figure 2: Diagram of the network modules during the environment learning and language learning phases. $s$ and $s'$ represent states before and after an action, $c$ represents a natural language command, and $a$ represents a latent action representation. The environment learning phase (i) uses a conditional autoencoder to pre-train the decoder $D$ toward a good representation space for $a$, so that fewer linguistic examples are needed during language learning (ii).

vironment. This setup applies when interactions with the environment are plentiful but only a few are labeled with language commands. For example, a robotic agent could passively observe a human performing a task, without requiring the human to perform any work they would not normally do, so that later the agent would need less direct instruction from a human in the form of language. We present an environment learning method that uses observations of state transitions to build a representation that aligns well with the transitions that tend to occur. The method takes advantage of the fact that in complex environments (or even relatively simple ones), not every state transition is equally likely, but the patterns of actions that do occur hint at an underlying structure that we can try to capture.

We demonstrate the effectiveness of our pretrained representations by using them to increase data efficiency on two instruction-following tasks (Section 4). We show that when given few instruction examples, a network using our pre-learned representations performs substantially better than an otherwise identical network without these representations, increasing performance by over ten absolute percentage points on small datasets and increasing data-efficiency by more than an order of magnitude. We find that while performance with a typical neural representation trained end-to-end lags considerably behind performance with human-designed representations, our unsupervised representations are able to help cross a substantial portion of this gap. In addition, we perform analysis of the meaning captured by our representations during the unsupervised environment learning phase, demonstrating that the semantics captured has noteworthy similarity to a hand-defined system of logical forms (Section 7).

## 2 Problem Setup

This work applies to the class of problems where instructions are mapped to actions conditioned on an environment state. These tasks can be formalized as learning a mapping $M(s, c) \mapsto s'$, where $c$ is a command in natural language, $s$ is an environment state, and $s'$ is the desired environment state after following the command. Classically, these problems are approached by introducing a logical form $l$ that does not depend on the state $s$, and learning a mapping from language to logical forms $P(c) \mapsto l$ (Artzi and Zettlemoyer, 2013; Zettlemoyer and Collins, 2005). A hand-defined execution function $Q(s, l) \mapsto s'$ is then used to generalize the action across all possible states. When $P$ and $Q$ are composed, $Q$ constrains the overall function to generalize in a semantically coherent way across different states.

In contrast to the logical form-based method, our work builds off of an end-to-end neural approach, which is applicable in settings where a system of logical forms is not provided. We structure our network as a language module $L$ and action decoder $D$ in an encoder-decoder style architecture (Figure 2(ii)), similar to previous neural instruction following work (e.g. Mei et al., 2016). $L$ and $D$ are analogous to the $P$ and $Q$ functions in the logical form approach, however, unlike before, the interface between the two modules is a vector $a$ and the function $D$ is learned. This gives the neural network greater flexibility, but also creates the problem that the decoder $D$ is no longer constrained to generalize across different states in natural ways.

## 3 Method

### 3.1 Learning Action Representations from the Environment

The goal of this paper is improve data efficiency by pre-training the decoder $D$ to use a better-generalizing representation for the vector $a$. We do this in an unsupervised way by allowing our system to see examples of state transitions (actions) in the environment before seeing any language. We suppose the existence of a large number of language-free state transitions $s, s'$ and introduce an environment learning phase to learn representations of these transitions before language learning starts. During this environment learning phase, we train a *conditional autoencoder* of $s'$ given $s$ by introducing an additional encoder $E(s, s') \mapsto a$ to go along with decoder $D(s, a) \mapsto s'$, as shown in Figure 2(i). Both $E$ and $D$ are given the initial state $s$, and $E$ must create a representation of the final state $s'$ so that $D$ can reproduce it from $s$. The parameters of $E$ and $D$ are trained to maximize log likelihood of $s'$ under the output distribution of $D$.

$$\arg\max_{\theta_E, \theta_D} \left[\log P_D(s'|s, E(s, s'))\right] \qquad (1)$$

If given enough capacity, the representation $a$ might encode all the information necessary to produce $s'$, allowing the decoder to ignore $s$. However, with a limited representation space, the decoder must learn to integrate information from $a$ and $s$, leading $a$ to capture an abstract representation of the transformation between $s$ and $s'$. To be effective, the representation $a$ needs to be widely applicable in the environment and align well with the types of state transitions that typically occur. These pressures cause the representation to avoid meanings like *to the right of blue except for on brown* that rarely apply. Note that during pre-training, we do not add any extra information to indicate that different transitions might be best represented with the same abstract action, but the procedure described here ends up discovering this structure on its own.

Later, after demonstrating the effectiveness of this environment learning procedure in Section 4, we introduce two additional improvements to the procedure in sections 5 and 6. In Section 7, we show that our pre-training discovers representations that align well with logical forms when they are provided.

### 3.2 Language Learning

After environment learning pre-training, we move to the language learning phase. In the language learning phase, we are given state transitions paired with commands $(s, s', c)$ and learn to map language to the appropriate result state $s'$ for a given state $s$. As discussed above and shown in Figure 2(ii), we form an encoder-decoder using a language encoder $L$ and action decoder $D$. To improve generalization, we use the decoder $D$ that was pre-trained during environment learning. If $D$ generalizes representations across different states in a coherent way as we hope, then the composed function $D(s, L(c))$ will also generalize well. We can either fix the parameters of $D$ after environment learning or simply use the pre-learned parameters as initialization, which will be discussed more in the experiments section below. The language module $L$ is trained by differentiating through the decoder $D$ to maximize the log probability that $D$ outputs the correct state $s'$.

$$\arg\max_{\theta_L} \left[\log P_D(s'|s, L(c))\right] \qquad (2)$$

### 3.3 Comparison with Action Priors

One of the roles of environment learning pre-training is to learn something like a prior over state transitions, ensuring that we select a reasonable action based on the types of transitions that we have seen. However, the method described here has advantages over a method that just learns a transition prior. In addition to representing which transitions are likely, our pre-training method also induces structure within the space of transitions. A single action representation $a$ can be applied to many different states to create different transitions, effectively creating a group of transitions. After training, this grouping might come to represent a semantically coherent category (see analysis in Section 7). This type of grouping information may not be easily extractable from a prior. For example, a prior can tell you that stacking red blocks on orange blocks is likely across a range of initial configurations, but our pre-training method may also choose to represent all of these transitions with the same vector $a$. Finding this underlying structure is key to the generalization improvements seen with our procedure.

## 4 Experiments

We evaluate our method in two different environments, as described below in sections 4.1 and 4.2.[1]

### 4.1 Block Stacking

For our first test environment, we use the block stacking task introduced by Wang et al. (2016) and depicted in Figure 1. This environment consists of a series of levels (tasks), where each level requires adding or removing blocks to get from a start configuration to a goal configuration. Human annotators were told to give the computer step by step instructions on how to move blocks from one configuration to the other. After each instruction, the annotator selected the desired resulting state from a list.

Following the original work for this dataset (Wang et al., 2016), we adopt an online learning setup and metric. The data is broken up into a number of sessions, one for each human annotator, where each session contains a stream of commands $c$ paired with block configuration states $s$. The stream is processed sequentially, and for each instruction the system predicts the result of applying command $c$ to state $s$, based on a model learned from previous examples in the stream. After making a prediction, the system is shown the correct result $s'$ and is allowed to make updates to its model before moving on to the next item in the stream. The evaluation metric, online accuracy, is then the percentage of examples for which the network predicted the correct resulting state $s'$ when given only previous items in the stream as training. Under this metric, getting predictions correct at the beginning of the stream, when given few to no examples, is just as important as getting predictions correct with the full set of data, making it as much a measure of data-efficiency as of final accuracy. The longest sessions only contain on the order of 100 training examples, so the bulk of predictions are made with *only tens of examples*.

To train a neural model in this framework, the model is updated by remembering all previous examples seen in the stream so far and training the neural network to convergence on the full set of prior examples. While training the network to convergence after every example is not very computationally efficient, the question of making efficient online updates to neural networks is orthogonal to the current work, and we wish to avoid any confounds introduced by methods that make fewer network updates.

Since the original dataset does not contain a large number of language-free state transitions as we need for environment learning, we generate synthetic transitions. To generate state transitions $s, s'$, we generate new levels using the random procedure used in the original work and programmatically determine a sequence of actions that solve them. The levels of the game are generated by a procedure which selects random states and then samples a series of transformations to apply to generate a goal state. We create a function that generates a sequence of states from the start to the goal state based on the transformations used during goal generation. Most of the levels require one or two actions with simple descriptions to reach the goal. Following the assumption that state transitions in the environment are plentiful, we generate new transitions for every batch during environment learning. We leave an analysis of the effect of environment learning data size to future work.

#### 4.1.1 State Representation and Network Architecture

We represent a state as a two dimensional grid, where each grid cell represents a possible location (stack index and height) of a block. The state inputs to the encoder and decoder networks use a one-hot encoding of the block color in each cell or an empty cell indicator if no block is present. The output of the decoder module is over the same grid, and a softmax over colors (or empty) is used to select the block at each position. Note that the original work in this environment restricted outputs to states reachable from the initial state by a logical form, but here we allow any arbitrary state to be output and the model must learn to select from a much larger hypothesis space.

The encoder module $E$ consists of convolutions over the states $s$ and $s'$, subtraction of the two representations, pooling over locations, and finally a fully connected network which outputs the representation $a$. The decoder module $D$ consists of convolution layers where the input is state $s$ and where $a$ is broadcast across all positions to an intermediate layer. The language module $L$ runs an LSTM over the words, then uses a fully connected

[1] Code for all experiments can be found at `github.com/dgaddy/environment-learning`.

network to convert the final state to the representation $a$. Details of the architecture and hyperparameters can be found in Appendix A.1.

### 4.1.2 Results

Our primary comparison is between a neural network with pre-trained action representations and an otherwise identical neural model with no pre-trained representations. The neural modules are identical, but in the full model we have fixed the parameters of the decoder $D$ after learning good representations with the environment learning procedure. We tune the baseline representation size independently since it may perform best under different conditions, choosing among a large range of comparable sizes (details in Appendix A.1). To evaluate the quality of our representations, we also compare with a system using hand-designed logical representations (Wang et al., 2016). While not strictly an upper bound, the human-designed representations were designed with intimate knowledge of the data environment and so provide a very good representation of actions people might take. This makes them a strong point of comparison for our unsupervised action representations.

Table 1 shows the results on this task. We find that training the action representation with environment learning provides a very large gain in performance over an identical network with no prelinguistic training, from 17.9% to 25.9%. In sections 5 and 6 below, we'll add discrete representations and an additional loss term which together bring the accuracy to 28.5%, an absolute increase of more than 10% over the baseline. Comparing against the system with human-designed representations shows that the environment learning pre-training substantially narrows the performance gap between hand designed representations and representations learned as part of an end-to-end neural system.

| Learned Representations (this work) | |
|---|---|
| Baseline | 17.9 |
| Environment Learning | 25.9 |
| + Discrete $a$ (Section 5) | 27.6 |
| + Encoder matching (Section 6) | **28.5** |
| Human-Designed Representations | |
| Wang et al. (2016) | 33.8 |

Table 1: Online accuracy for the block stacking task.[2] Pre-learning action representations with environment learning greatly improves performance over the baseline model, substantially narrowing the gap between hand designed representations and representations learned as part of an end-to-end neural system. Note that these numbers represent accuracy after learning from only tens of examples.

## 4.2 String Manipulation

The second task we use to test our method is string manipulation. In this task a state $s$ is a string of characters and actions correspond to applying a transformation that inserts or replaces characters in the string, as demonstrated in Figure 3. We use the human annotations gathered by Andreas et al. (2018), but adapt the setup to better measure data-efficiency.

The baseline neural model was unable to learn useful models for this task using data sizes appropriate for the online learning setup we used in the previous task, so we instead adopt a slightly different evaluation where accuracy at different data sizes is compared. We structure the data for evaluation as follows: First, we group the data so that each group contains only a small number of instructions (10). In the original data, each instruction comes with multiple example strings, so we create distinct datapoints $s, s', c$ for each example with the instruction string repeated. Our goal is to see how many examples are needed for a model to learn to apply a set of 10 instructions. We train a model on training sets of different sizes and evaluate accuracy on a held-out set of 200 examples. We are primarily interested in generalization across new environment states, so the held-out set consists of examples with the same instructions but new initial states $s$. Due to high data requirements of the baseline neural system, we found it necessary to augment the set of examples for each instruction with additional generated examples according to the regular expressions included with the dataset. Our final metric is the average accuracy across 5 instruction groups, and we plot this accuracy for different training set sizes.

State transitions for environment learning are generated synthetically by selecting words from a dictionary and applying regular expressions, where the regular expressions to apply were sampled from a regular-expression generation procedure written by the creators of the original dataset. The environment learning procedure is exposed to

[2] Although the variance between runs was small relative to the gaps in performance, we report an average over three random initializations to ensure a fair comparison.

| | |
|---|---|
| $c$ | replace consonants with p x |
| $s$ | `fines` |
| $s'$ | `pxipxepx` |

| | |
|---|---|
| $c$ | add a letter k before every b |
| $s$ | `rabbles` |
| $s'$ | `rakbkbles` |

| | |
|---|---|
| $c$ | replace vowel consonant pairing with v g |
| $s$ | `thatched` |
| $s'$ | `thvgchvg` |

| | |
|---|---|
| $c$ | add b for the third letter |
| $s$ | `thanks` |
| $s'$ | `thbanks` |

Figure 3: Examples from the string manipulation task along with desired outputs.

transitions from thousands of unique regular expressions that it must make sense of and learn to represent.

#### 4.2.1 Network Architecture

For this task, the state inputs and outputs are represented as sequences of characters. The encoder $E$ runs a LSTM over the character sequences for $s$ and $s'$, then combines the final states with a feed-forward network to get $a$. The decoder $D$ runs a LSTM over the characters of $s$, combines this with the representation $a$, then outputs $s'$ using another LSTM. The module architecture details and hyper-parameters can be found in Appendix A.2.

Since our evaluation for this task considers larger dataset sizes in addition to very small sizes, we do not fix the parameters of the decoder $D$ as we did in the previous task, but instead use the pre-trained decoder as initialization and train it along with the language module parameters. Allowing the parameters to train gives the decoder more power to change its representations when it has enough data to do so, while the initialization helps it generalize much better, as demonstrated by our results below.

#### 4.2.2 Results

As with the other dataset (Section 4.1), we compare the full model with a baseline that has no environment learning, but an otherwise identical architecture. To ensure a fair comparison, we tune the baseline representation size separately, choosing the best from a range of comparable sizes (see Appendix A.2).

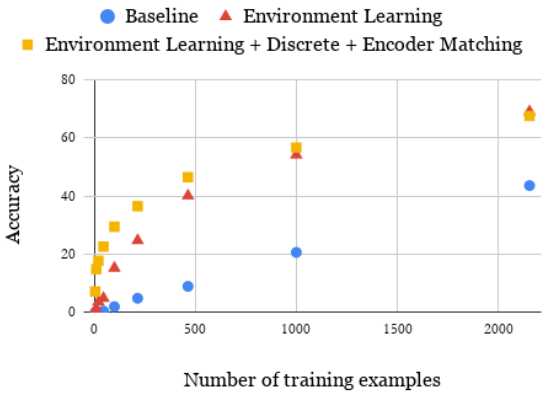


Figure 4: Accuracy for the string manipulation task as the number of examples $(s, s', c)$ is increased. Environment learning pre-training increases data efficiency by an order of magnitude or more. The results in yellow include additional improvements described in sections 5 and 6 below.

Figure 4 plots the accuracy across different data sizes of the baseline neural model and the model with environment learning pre-training. Note that models are trained to convergence, so this plot is intended to indicate data efficiency, not training speed (though training speed is also likely to increase at similar rates). As seen in the figure, environment learning substantially increases data efficiency on this task. At small data sizes, the baseline model struggles to generalize across different states $s$, often choosing to output one of the training outputs $s'$ rather than learning a rule and applying it to $s$. Environment learning greatly increases the ability of the model to find the correct generalization.

## 5 Discrete Action Representations

In this section, we describe a variant of our model where we use a discrete representation $a$ instead of a continuous one and evaluate this variant on our two tasks. Semantics is often defined in terms of discrete logical structures. Even in continuous environments, it is often natural to describe objects and relations in discrete ways. Using a discrete space for our learned action representations can provide useful inductive bias for capturing this discrete structure. In addition, a discrete representation has the potential advantages of increased robustness and increased control of information flow during environment learning.

When using discrete representations, we divide our $a$ into $n$ different discrete random variables where each variable selects its value from one of $k$

categories. We train the discrete representation using the Gumbel-Softmax (Jang et al., 2017; Maddison et al., 2017), which gives us a continuous relaxation of the discrete variables that we can backpropagate through. The Gumbel-Softmax operation transforms an $n \times k$ vector into $n$ discrete random variables, which we represent as one-hot vectors and feed through the rest of the network just as we would a continuous representation. The Gumbel-Softmax is calculated as

$$G(x_i) = \frac{\exp(x_i + \epsilon_i)}{\sum_{j=0}^{k} \exp(x_j + \epsilon_j)}$$

where $\epsilon$ are i.i.d. samples from the Gumbel(0,1) distribution and the vector $x$ represents unnormalized log probabilities for each of the $k$ categories. This operation is analogous to a softening of a sample from the distribution. While the original work suggested the use of an annealed temperature parameter, we did not find it necessary in our experiments. We use the straight-through variant, where the discrete mode of the softmax distribution is used in the forward pass, but the backward pass is run as if we had used the continuous value $G(x_i)$. We found that a representation with $n = 20$ variables and $k = 30$ values works well for all our experiments.

Using discrete representations instead of continuous representations further improves environment learning results on both tasks, increasing the block stacking task accuracy from 25.9% to 27.6% (Table 1) and improving string manipulation on moderate training sizes (200 examples) from 24.7% to 36.9%. We also ran the baseline neural models with discrete representations for comparison but did not observe any performance gains, indicating that the discrete representations are useful primarily when used with environment learning pre-training.

## 6 Encoder Representation Matching

One potential difficulty that may occur when moving from the environment learning to the language learning phase is that the language module $L$ could choose to use parts of the action representation space that were not used by the encoder during environment learning. Because the decoder has not seen these representations, it may not have useful meanings associated with them, causing it to generalize in a suboptimal way. In this section, we introduce a technique to alleviate this problem and show that it can lead to an additional improvement in performance.

Our fix uses an additional loss term to encourage the language module $L$ to output representations that are similar to those used by the encoder $E$. For a particular input $c, s, s'$ in the language learning phase, we run the encoder on $s, s'$ to generate a possible representation $a_E$ of this transition. We then add an objective term for the log likelihood of $a_E$ under $L$'s output distribution. The full objective during language learning is then

$$\arg\max_{\theta_L} \left[\log P_D(s'|s, L(c)) + \lambda \log P_L(a_E|c)\right] \tag{3}$$

where the encoder matching weight $\lambda$ is a tuned constant. $P_L$ is the softmax probability from the output of the language module when using discrete representations for $a$, and $a_E$ is the discrete mode of the encoder output distribution.[3]

Using this technique on the block stacking task (with $\lambda = .01$), we see a performance gain of .9% over discrete-representation environment learning to reach an accuracy of 28.5%. This number represents our full model performance and demonstrates more than 10% absolute improvement over the baseline. The additional loss also provides gains on string manipulation, especially on very small data sizes (e.g. from 3.9% to 14.8% with only 10 examples). The performance curve of our complete model is shown in Figure 4. With our full model, it takes less than 50 examples to reach the same performance as with 1000 examples using a standard neural approach.

## 7 Exploring the Learned Representation

A primary goal of the environment learning procedure is to find a representation of actions that generalizes in a semantically minimal and coherent way. In this section, we perform analysis to see what meanings the learned action representations capture in the block stacking environment. Since logical forms are engineered to capture semantics that we as humans consider natural, we compare our learned representations with a system of logical forms to see if they capture similar meanings without having been manually constrained to do

[3] When using continuous representations, a $\ell_2$ distance penalty could be used to encourage similarity between the output of $L$ and $E$, though this tended to be less effective in our experiments.

so. We compare the semantics of the learned and logical representations by comparing their effect on different states, based on the method of Andreas and Klein (2017).

We test an encoder and decoder using the following procedure: First, we generate a random transition $s_1, s'_1$ from the same distribution used for environment learning and run the encoder to generate an action representation $a_1$ for this transition. Then, we generate a new state $s_2$ from the environment and run the decoder on the new state with the representation generated for the original state: $D(a_1, s_2) \mapsto \bar{s}_2$. We are interested in whether the output $\bar{s}_2$ of this decoding operation corresponds to a generalization that would be made by a simple logical form. Using a set of logical forms that correspond to common actions in the block stacking environment, we find all simple logical forms that apply to the original transition $s_1, s'_1$ and all forms that apply to the predicted transition $s_2, \bar{s}_2$. If the intersection of these two sets of logical forms is non-empty, then the decoder's interpretation of the representation $a_1$ is consistent with some simple logical form. We repeat this procedure on 10,000 state transitions to form a logical form consistency metric.

Running this test on our best-performance model, we find that 84% of the generalizations are consistent with one of the simple logical forms we defined. This result indicates that while the generalization doesn't perfectly match our logical form system, it does have a noteworthy similarity. An inspection of the cases that did not align with the logical forms found that the majority of the "errors" could in fact be represented by logical forms, but ones that were not minimal. In these cases, the generalization isn't unreasonable, but has slightly more complexity than is necessary. For example, from a transition that could be described either as *stack a blue block on the leftmost block* or separately as *stack blue blocks on red blocks* (where red only appears in the leftmost position), the representation $a$ that is generated generalizes across different states as the conjunction of these two meanings (*stack blue blocks on the leftmost block AND on red blocks*), even though no transitions observed during environment learning would need this extra complexity to be accurately described.

## 8 Related Work

Many other works use autoencoders to form representations in an unsupervised or semi-supervised way. Variants such as denoising autoencoders (Vincent et al., 2008) and variational autoencoders (Kingma and Welling, 2013) have been used for various vision and language tasks. In the area of semantic grounding, Kočiský et al. (2016) perform semi-supervised semantic parsing using an autoencoder where the latent state takes the form of language.

Our approach also relates to recent work on learning artificial languages by simulating agents interacting in an environment (Mordatch and Abbeel, 2018; Das et al., 2017; Kottur et al., 2017, i.a.). Our environment learning procedure could be viewed as a language learning game where the encoder is a speaker and the decoder is a listener. The speaker must create a "language" $a$ that allows the decoder to complete a task. Many of these papers have found that it is possible to induce representations that align semantically with language humans use, as explored in detail in Andreas and Klein (2017). Our analysis in Section 7 is based on the method from this work.

Model-based reinforcement learning is another area of work that improves data-efficiency by learning from observations of an environment (Wang et al., 2018; Deisenroth et al., 2013; Kaiser et al., 2019). It differs from the current work in which aspect of the environment it seeks to capture: in model-based RL the goal is to model which states will result from taking a particular action, but in this work we aim to learn patterns in what actions tend to be chosen by a knowledgeable actor.

Another related line of research uses language to guide learning about an environment (Branavan et al., 2012; Srivastava et al., 2017; Andreas et al., 2018; Hancock et al., 2018). These papers use language to learn about an environment more efficiently, which can be seen as a kind of inverse to our work, where we use environment knowledge to learn language more efficiently.

Finally, recent work by Leonandya et al. (2018) also explores neural architectures for the block stacking task we used in section 4.1. The authors recognize the need for additional inductive bias, and introduce this bias by creating additional synthetic *supervised* data with artificial language, creating a transfer learning-style setup. This is in contrast to our unsupervised pre-training method that does not need language for the additional data. Even with their stronger data assumptions, their online accuracy evaluation reaches just 23%, compared to our result of 28.5%, providing independent verification of the difficulty of this task for neural networks.

## 9 Conclusion

It is well known that neural methods do best when given extremely large amounts of data. As a result, much of AI and NLP community has focused on making larger and larger datasets, but we believe it is equally important to go the other direction and explore methods that help performance with little data. This work introduces one such method. Inspired by the idea that it is easier to map language to pre-linguistic concepts, we show that when grounding language to actions in an environment, pre-learning representations of actions can help us learn language from fewer language-action pairings.

## Acknowledgments

This work is supported by the DARPA Explainable Artificial Intelligence (XAI) program. We would like to thank the members of the Berkeley NLP group and the anonymous reviewers for their helpful feedback.

## References

Jacob Andreas and Dan Klein. 2017. Analogs of linguistic structure in deep representations. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2893–2897. Association for Computational Linguistics.

Jacob Andreas, Dan Klein, and Sergey Levine. 2018. Learning with latent language. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2166–2179. Association for Computational Linguistics.

Yoav Artzi and Luke Zettlemoyer. 2013. Weakly supervised learning of semantic parsers for mapping instructions to actions. *Transactions of the Association for Computational Linguistics*, 1:49–62.

Paul Bloom. 2000. *How children learn the meanings of words*. MIT Press.

SRK Branavan, David Silver, and Regina Barzilay. 2012. Learning to win by reading manuals in a monte-carlo framework. *Journal of Artificial Intelligence Research*, 43:661–704.

Devendra Singh Chaplot, Kanthashree Mysore Sathyendra, Rama Kumar Pasumarthi, Dheeraj Rajagopal, and Ruslan Salakhutdinov. 2018. Gated-attention architectures for task-oriented language grounding. In *AAAI*.

Abhishek Das, Satwik Kottur, José MF Moura, Stefan Lee, and Dhruv Batra. 2017. Learning cooperative visual dialog agents with deep reinforcement learning. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2951–2960.

Marc Peter Deisenroth, Gerhard Neumann, Jan Peters, et al. 2013. A survey on policy search for robotics. *Foundations and Trends in Robotics*, 2(1–2):1–142.

Braden Hancock, Paroma Varma, Stephanie Wang, Martin Bringmann, Percy Liang, and Christopher Ré. 2018. Training classifiers with natural language explanations. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1884–1895. Association for Computational Linguistics.

Susan J Hespos and Elizabeth S Spelke. 2004. Conceptual precursors to language. *Nature*, 430(6998):453.

Eric Jang, Shixiang Gu, and Ben Poole. 2017. Categorical reparameterization with gumbel-softmax. In *International Conference on Learning Representations*.

Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. 2019. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*.

Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

Tomáš Kočiský, Gábor Melis, Edward Grefenstette, Chris Dyer, Wang Ling, Phil Blunsom, and Karl Moritz Hermann. 2016. Semantic parsing with semi-supervised sequential autoencoders. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1078–1087. Association for Computational Linguistics.

Satwik Kottur, José Moura, Stefan Lee, and Dhruv Batra. 2017. Natural language does not emerge 'naturally' in multi-agent dialog. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 2962–2967. Association for Computational Linguistics.

Rezka Leonandya, Elia Bruni, Dieuwke Hupkes, and Germán Kruszewski. 2018. The fast and the flexible: training neural networks to learn to follow instructions from small data. *arXiv preprint arXiv:1809.06194*.

Chris J Maddison, Andriy Mnih, and Yee Whye Teh. 2017. The concrete distribution: A continuous relaxation of discrete random variables. In *International Conference on Learning Representations*.

Hongyuan Mei, Mohit Bansal, and Matthew R Walter. 2016. Listen, attend, and walk: Neural mapping of navigational instructions to action sequences. In *AAAI*, volume 1, page 2.

Dipendra Misra, Andrew Bennett, Valts Blukis, Eyvind Niklasson, Max Shatkhin, and Yoav Artzi. 2018. Mapping instructions to actions in 3d environments with visual goal prediction. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2667–2678. Association for Computational Linguistics.

Igor Mordatch and Pieter Abbeel. 2018. Emergence of grounded compositional language in multi-agent populations. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Shashank Srivastava, Igor Labutov, and Tom Mitchell. 2017. Joint concept learning and semantic parsing from natural language explanations. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1527–1536. Association for Computational Linguistics.

Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. 2008. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM.

Sida I. Wang, Percy Liang, and Christopher D. Manning. 2016. Learning language games through interaction. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2368–2378. Association for Computational Linguistics.

Xin Wang, Wenhan Xiong, Hongmin Wang, and William Yang Wang. 2018. Look before you leap: Bridging model-free and model-based reinforcement learning for planned-ahead vision-and-language navigation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 37–53.

Luke Zettlemoyer and Mike Collins. 2005. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Uncertainty in Artificial Intelligence (UAI)*, pages 658–666.

# A Neural Architectures and Hyperparameters

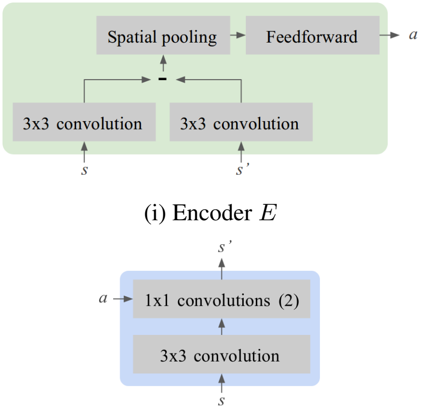
## A.1 Block Stacking

The encoder and decoder module architectures for the block stacking task are shown in Figure 5. The encoder module $E$ consists of convolutions over the states $s$ and $s'$, subtraction of the two representations, pooling over locations, and finally a fully connected network which outputs the representation $a$. The fully connected network has a single hidden layer. The decoder module $D$ consists of two convolution layers where the input is state $s$ and where $a$ is broadcast across all positions and concatenated with the input to the second layer. All convolutions and feedforward layers for $E$ and $D$ have dimension 200 and all intermediate layers are followed by ReLU non-linearities. Dropout with probability 0.5 was used on the encoder feedforward hidden layer and before the last convolution layer in the decoder.

The language module $L$ uses a LSTM encoder (Figure 6). It takes a command $c$ as a sequence of learned word embeddings, runs an LSTM over them, then projects from the final cell state to get the output vector $a$. The word embeddings have dimension 100 and the LSTM has hidden size 200.

When using a continuous action representation, $a$ has dimension 600. When using a discrete representation, we use $n = 20$ discrete variables where each takes one of $k = 30$ values. Environment learning is run on 500,000 batches of size 20, after which we fix the parameters of $D$. During language learning, we optimize $L$ for 50 epochs after each new example is presented, using a batch size of 1. All optimization is done using Adam with learning rate 0.001.

To ensure a fair comparison with the baseline, we ran the baseline system with both continuous and discrete representations and took the best. Generally, the baseline performed slightly better



(i) Encoder $E$

(ii) Decoder $D$

Figure 5: Architecture for block stacking task modules.

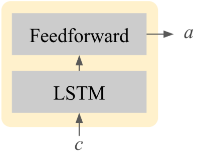


Figure 6: The language module $L$ used for both the block stacking and string manipulation tasks uses a LSTM over the words of the command $c$.

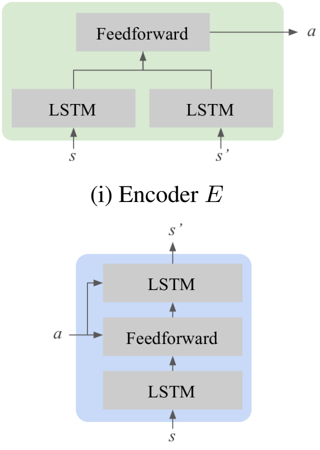
with continuous representations. We ran with continuous sizes 20, 50, 100, 300, and 600; selecting the best result. This range was chosen to be between the number of discrete variables $n$ and the total number of inputs to the discretization $n \times k$.

### A.2 String Manipulation

Figure 7 shows the encoder and decoder module architectures for the string manipulation task. The encoder $E$ runs a LSTM over the character sequences for $s$ and $s'$, using separate LSTMs for the two sequences, but tying their parameters. The final states of the two LSTMs are then concatenated and fed into a feedforward network with one hidden layer that outputs the action representation $a$. The decoder $D$ consists of a LSTM over the sequence $s$, a feedforward network of a single linear layer combining $a$ with the LSTM final state, and a LSTM that outputs the sequence $s'$, where the output LSTM's initial state comes from the output of the feedforward network. $a$ is also concatenated with the previous output embedding that is fed into the input of the LSTM at each timestep. The character embeddings input to the LSTM have dimension 50, and all LSTM and feedforward layers have dimension 500. When using a continuous representation $a$, we use a representation dimension of 20, though the results were not overly sensitive to this value. When using a discrete representation, we use $n = 20$ variables where each takes one of $k = 30$ values.

The language module for this task is identical to the module used for the block stacking task, as shown in Figure 6. The training and optimizer hyperparameters are the same as in the block stacking task.

As in the block stacking task, we tune the baseline representation hyperparameters over continuous sizes 20, 50, 100, 300, and 600, as well as an identical-sized discrete representation.



(i) Encoder $E$

(ii) Decoder $D$

Figure 7: Architecture for string manipulation task modules.