Supplementary Information

Here we provide details of the various language modeling approaches used for task grounding including a breakdown of the specific details of the IBM Model 2 Language Model, as well as each of the separate deep neural network models.

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

$$\hat{l}, \hat{m} = \arg\max_{l,m} Pr(l, m \mid c) \tag{1}$$

IBM Model 2 - Statistical Language Model:

IBM2 is a generative model that solves the following objective, which is equivalent to Eqn. 1 by Bayes' rule:

$$\hat{l}, \hat{m} = \arg\max_{l,m} Pr(l, m) \cdot Pr(c \mid l, m) \tag{2}$$

In this equation, the first term, $Pr(l, m)$ can be treated as a distribution over the reward function space. We make the assumption that each (l, m) tuple is distributed uniformly at random. Thus, the IBM2 learning objective simplifies to the following:

$$\hat{l}, \hat{m} = \arg\max_{l,m} Pr(c \mid l, m) \tag{3}$$

This probability of a natural language command (c) given the reward (m) and level of abstraction (l) is then given by the following IBM2 equation:

$$Pr(c|m, l) = \eta(n_c|n_m, l) \sum_{a} \prod_{j}^{n_c} \delta(a_j|j, n_c, n_m, l)\tau(c_j|m_{a_j}, l) \tag{4}$$

where η, δ, and τ are IBM2 specific parameters that are learned via the EM algorithm. $\eta(n_c \mid n_m, l)$ denotes the probability of generating a natural language command of length n_c from a reward function of length n_m, and level l. The sum is defined over all possible alignments of natural language words to reward function tokens. For computational efficiency, we approximate the sum by sampling from the set of possible alignments, following standard practice.

We take a standard approach to training our IBM2 using the EM algorithm with a "bake-in" period where the EM algorithm is run for a set number of iterations only for

translation parameter (τ) updates. We then learn follow with regular iterations of the EM algorithm where both the translation parameters (τ) and the alignment parameters (δ) are updated. We estimate the length parameters (η) using Maximum-Likelihood estimation.

At inference time, to pick the (l, m) tuple that maximizes the objective from Equation 3 we calculate the IBM2 probability for every possible (l, m) combination, using the IBM2 as a reranker over the possible reward function translations. We find this gives significantly better results than beam-search decoding due to the relatively small size of the reward function space, as well the formulaic nature of each reward function string.

Multi-NN - Multiple Output Feed-Forward Network:

A breakdown of the exact network transformations is as follows:

$$\vec{e} = \text{Lookup}(\mathbf{E}, \vec{c})$$
$$\vec{s} = \text{ReLU}(\vec{e} \cdot \mathbf{W_s} + \mathbf{b}_s)$$
$$\vec{t} = \text{ReLU}(\vec{s} \cdot \mathbf{W_t^k} + \mathbf{b}_t^k)$$
$$\vec{o} = \text{Softmax}(\vec{t} \cdot \mathbf{W_t^k} + \mathbf{b}_t^k)$$

Here, the layer specific weight and bias parameters are given by $\mathbf{W}, \mathbf{b}$ respectively. Superscripts denote output-specific parameters and the $(\cdot)$ operation denotes matrix-vector product. In order to produce high-dimensional, fixed-size representations of each word in the finite natural language vocabulary, the initial embedding layer contains a lookup matrix $\mathbf{E}$, trained via backpropagation with the rest of the model, where each row denotes a single word embedding. The embedding for all words in $\vec{c}$ are summed together according to their respective frequencies to produce an embedding for the full natural language command. All hidden layers employ the rectifier activation function (ReLU) whereas the final output layer produces a Softmax distribution over the output categories.

The fixed-size embedding is then passed through a neural network layer (shared across all outputs), with a ReLU non-linear activation, generating a hidden state vector $\vec{h}$. This hidden state vector is passed through an output-specific hidden layer, also with a ReLU activation, and finally an output-specific read-out layer, with a Softmax activation, to generate a probability distribution over the output categories. The loss is computed as the sum of the cross-entropy loss over the different outputs - namely, the computed loss for the level selection distribution, and each of the three different reward function distributions.

Gated Recurrent Units

Both the Multi-RNN and Single-RNN models leverage Gated Recurrent Unit (GRU) cells, a specific type of Recurrent Neural Network cell. GRU Cells only maintain a

single hidden state h, and the update rules are as follows:

$$\begin{aligned}
\vec{z}_t &= \sigma(\mathbf{W_z} \cdot \vec{x}_t + \mathbf{U_z} \cdot \vec{h}_{t-1} + \mathbf{b}_z) \\
\vec{r}_t &= \sigma(\mathbf{W_r} \cdot \vec{x}_t + \mathbf{U_r} \cdot \vec{h}_{t-1} + \mathbf{b}_r) \\
\vec{n}_t &= \text{Tanh}(\mathbf{W_h} \cdot \vec{x}_t + \mathbf{U_h} \cdot (\vec{r}_t \odot \vec{h}_{t-1}) + \mathbf{b}_z) \\
\vec{h}_t &= (\vec{1} - \vec{z}_t) \odot \vec{h}_{t-1} + \vec{z}_t \odot \vec{n}_t
\end{aligned}$$

Here, the $(\cdot)$ operation denotes matrix-vector product, while the $(\odot)$ operation denotes element-wise product. The intermediate vectors $\vec{z}, \vec{r}$ act as update and reset "gates" dictating how much of the hidden state should be overwritten with the new information in x_t. The parameters $\mathbf{W}, \mathbf{U}, \mathbf{b}$ are specific to the GRU cell, and are trainable via backpropagation along with the rest of the model. The hidden state h is initialized as the zero vector at $t = 0$.

Multi-RNN - Multiple Output Recurrent Network:

We now give a detailed breakdown of the exact network transformations that make up the Multi-RNN:

$$\begin{aligned}
\vec{e}_1, \vec{e}_2 \dots \vec{e}_n &= \text{Lookup}(\mathbf{E}, c_1, c_2 \dots c_n) \\
\vec{h} &= \text{GRU}(\vec{e}_1, \vec{e}_2, \dots \vec{e}_n) \\
\vec{s} &= \text{ReLU}(\vec{h} \cdot \mathbf{W_s} + \mathbf{b}_s) \\
\vec{t} &= \text{ReLU}(\vec{s} \cdot \mathbf{W_t^k} + \mathbf{b}_t^k) \\
\vec{o} &= \text{Softmax}(\vec{t} \cdot \mathbf{W_t^k} + \mathbf{b}_t^k)
\end{aligned}$$

Again, layer parameters are given by $\mathbf{W}, \mathbf{b}$, with superscripts denoting output-specific parameters. The loss is the same as that used by the Multi-NN model.

Single-RNN - Single Output Recurrent Network:

A detailed breakdown of the Single-RNN transformations are as follows:

$$\begin{aligned}
\vec{e}_1, \vec{e}_2 \dots \vec{e}_n &= \text{Lookup}(\mathbf{E}, c_1, c_2 \dots c_n) \\
\vec{h} &= \text{GRU}(\vec{e}_1, \vec{e}_2, \dots \vec{e}_n) \\
\vec{s} &= \text{ReLU}(\vec{h} \cdot \mathbf{W_s} + \mathbf{b}_s) \\
\vec{t} &= \text{ReLU}(\vec{s} \cdot \mathbf{W_t} + \mathbf{b}_t) \\
\vec{o} &= \text{Softmax}(\vec{t} \cdot \mathbf{W_t} + \mathbf{b}_t)
\end{aligned}$$

Note that these transformations are exactly the same as the Multi-RNN, with the sole exception that there is only a single output, rather than multiple. As there is only a single output, the new loss is just the cross-entropy loss of the predicted joint level-reward function distribution.