CS5740: Assignment 4

https://github.com/cornell-cs5740-21sp/a4--a4-group-6/tree/master

Nishant Nayak nn343 | Richard Yu ry275 | Yuxin Zhang yz2726

1 Introduction (5pt)

We implemented a sequence to sequence model to map instructions to actions in the ALCHEMY environment, which includes seven beakers that can hold up to four liquids of different colors (the beakers are treated as stacks). There are six different colors. The environment system can move colored liquid content from one beaker to another, mix content, or remove content from the set of beakers. We used data in JSON format, where each example is a dictionary containing three keys: 'identifier', 'initial_env', and 'utterances'.

Using a training set of 18,285 utterances, development set of 1,225 utterances, and test set of 2,245 utterances, We experimented with four different model architectures: (1) a basic sequence-to-sequence model, (2) a world state encoded model, (3) a world state encoded model with an attention mechanism, and (4) a world state encoded model with an attention mechanism and access to interaction history. Our final model uses model architecture (4), and it achieved 25.6% development instruction level accuracy, 0.8% development interaction level accuracy, and 0.7% test interaction level accuracy.

2 Model (35pt)

2.1 Base Sequence-to-Sequence Model

The basic sequence-to-sequence model uses an encoder RNN to process input instruction sequence i as $\overline{x}_i = \langle x_{i,1}, x_{i,2}, ..., x_{i,|\overline{x}_i|} \rangle$, with $|\overline{x}_i|$ being the input dimension (with padding), and a decoder RNN to generate the output action sequence $\overline{y}_i$. For both RNNs, we use a single-layer non-bidirectional LSTM. At each time step j, the encoder computes an (output, hidden state) tuple O_j^E, h_j^E, where $h_j^E = [\overrightarrow{h_j^E}]$ (or $[\overrightarrow{h_j^E}, \overleftarrow{h_j^E}]$ if bidirectional), for each input instruction token $x_{i,j}$ using the encoder RNN. The forward RNN calculation takes in the embedded current input token and the last step's hidden state and computes the (current output, hidden state) defined by: $O_j^E, h_j^E = LSTM^E(\phi^x(x_{i,j}); h_{j-1}^E)$. The decoder RNN's initial hidden state is the zero tensor. At each step k, the decoder takes as input the embedding of the previously predicted action token $y_{i,k-1}$ (which is computed by taking the argmax over O_{k-1}^D), the previous decoder hidden state h_{k-1}^D, and the last hidden state of the encoder. The decoder RNN then computes the current (output, hidden state) as defined by: $O_k^D, h_k^D = LSTM^D([\phi^y(y_{i,k-1})]; h_{k-1}^D; h_{|\overline{x}_i|}^E)$.

2.2 Adding State Information

To incorporate world state information into the decoder, we generated a 196 dimension state context vector sc_i, representing the initial environment of the interaction corresponding to instruction sequence i, by using 1-hot vectors to represent each "liquid item" of each beaker. The 1-hot vectors have 7 dimensions, since a beaker item can be one of 6 different colored liquids, or be nothing. Since each beaker can hold a maximum of four liquid items and there are 7 beakers, sc_i has $7 \cdot 4 \cdot 7 = 196$ dimensions. We simply concatenate sc_i with the other inputs to the decoder RNN, at every time step. $O_k^D, h_k^D = LSTM^D([\phi^y(y_{i,k-1})]; h_{k-1}^D; h_{|\overline{x}_i|}^E; sc_i)$

2.3 Adding Attention

We incorporated a Bahdanau attention mechanism over the instruction encoder outputs into our model. At each time step of the decoding process, we compute alignment scores over all the encoder outputs, do a softmax over the scores to produce a distribution, and use the distribution to compute a weighted average across all the outputs to produce a new instruction context vector to be concatenated with the other decoder RNN inputs. Using the decoder's previous hidden state, and all of encoder's outputs O^E, we generate the instruction context ic_k for time step k using various learned weights W, and the Tanh function.

$$\text{score} = W_{\text{combined}} \cdot \tanh(W_{\text{dec}} \cdot h_{k-1}^D + W_{\text{enc}} \cdot O^E)$$

$$ic_k = O^E \cdot \text{softmax}(\text{score})$$

At each time step, our decoder generates a new output and hidden state as follows: $O_k^D, h_k^D = LSTM^D([\phi^y(y_{i,k-1})]; h_{k-1}^D; ic_k; sc_i)$

2.4 Adding Interaction Information

We incorporate interaction information into our model by using a separate encoder RNN (but identical to the encoder we used to encode instruction

sequence inputs in sections 2.1, 2.2, and 2.3) to process the instruction sequence that directly came before the current one in the same interaction. We also apply a separate attention mechanism (but identical to the Bahdanau mechanism described in section 2.3) over the previous instruction encoder's outputs, to compute a previous instruction context vector pic_k for each time step k during the decoding process. We then concatenate pic_k with the other decoder inputs. At each time step, our decoder generates a new output and hidden state as follows: $O_k^D, h_k^D = LSTM^D([\phi^y(y_{i,k-1})]; h_{k-1}^D; pic_k; ic_k; sc_i)$.

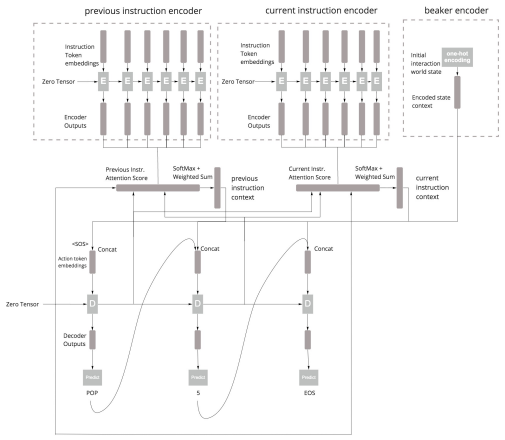

Figure 1: Diagram of our final model architecture.

3 Learning (8pt)

We first preprocessed the data by tokenizing the instructions by splitting the utterances by spaces. We used the resulting word tokens from the instructions of the training, development, and test data sets to form the input instruction vocabulary $\mathcal{V}^x$. We apply the same process to generate the ALCHEMY action vocabulary $\mathcal{V}^y$, by splitting actions from the training data on spaces to produce the tokens. After processing the training, development, and test data JSON files, we construct examples which can be represented as $\bar{X}_i := [\bar{x}_i; \bar{x}_{i-1}; sc_i]$, which have corresponding ground truth action vectors $\bar{y^*}_i$. Before training the model, we split the training data set into 85% training examples and 15% validation examples. We train our final model using the Adam optimizer for 50 epochs maximum, and we use early stopping (training stop if the validation set loss does not improve for 3 epochs). We calculate loss, with the Cross-Entropy loss function, for every pair of generated output O_k^D and true action token $y_{i,k}^*$. The loss is defined as follows:

$$\text{loss}(O_k^D, \phi^y(y_{i,k}^*)) = -\log\left(\frac{\exp(O_k^D[\phi^y(y_{i,k}^*)])}{\sum_j \exp(O_k^D[j])}\right)$$

In the training process, we use teacher-forcing by using the ground-truth ALCHEMY action $y_{i,k-1}^*$ instead of the predicted action $y_{i,k-1}$, at each time step during the decoding process, i.e. $O_k^D, h_k^D = LSTM^D([\phi^y(y_{i,k-1}^*)]; h_{k-1}^D; pic_k; ic_k; sc_i)$. However, during testing/prediction time, we do not have access to the ground-truth actions, so we cannot use teacher-forcing. As a result, if a single predicted action token is incorrect during the generation process, the entire predicted action for that specific input instruction will be wrong. This makes our model prone to one-off errors and low test interaction accuracy, since the model basically needs to correctly predict the ALCHEMY actions for all instructions in an interaction, for it to count as an interaction that the model predicted correctly. For example, if the model incorrectly predicts an action token for the first input instruction of an interaction, then it is most likely that the entire interaction will be thrown off course (unless the model somehow incorrectly predicts future action tokens that miraculous brings the interaction back on track).

4 Data (4pt)

After processing the raw JSON data as described in section 3, each example in the resulting training, development and test data sets is a $(\bar{X}_i, \bar{y}_i^*)$ pair. Table 1 contains shallow statistics of our data. The vocabulary for instruction tokens are calculated by splitting all the training, development, and test utterances on spaces (after lowercasing everything), so that there's no need to deal with unknown words. We do the same procedure to generate the vocabulary of ALCHEMY action tokens using the actions from the training data set. In addition to these tokens from the data, we also include three other "indicator" tokens in each vocabulary – the padding token,the start token, and the end token, which are respectively encoded as 0,1,2.

Training interactions	3657
Training utterances/(instr., action) pairs	18285
Dev. interactions	245
Dev utterances/(instr., action)	1225
Test interactions	449
Test utterances/(instr., action)	2245
Instruction vocabulary size	716
ALCHEMY action vocabulary size	51
Maximum instruction length	34

Table 1: Dataset Summary Statistics

5 Implementation Details (2pt)

We used an instruction token embedding size of 64, and ALCHEMY action token embedding size of 64. Our encoder LSTMs had a hidden size of 128, and our decoder LSTM also had a hidden size of 128. In order to make our model train relatively quickly, we used a batch size of 20. To batch, for every example $\bar{X}_i$, we padded $\bar{x}_i$ and $\bar{x}_{i-1}$ with zeros so that they are of the maximum instruction length. Because the

ground truth action vectors $\hat{y^*}_i$ are different lengths across different examples, we also padded them with zeros until they were all of the maximum action sequence length observed from the training data set. We used masking when computing the Cross-Entropy losses across examples in a batch.

6 Experiments and Results

Test Results (3pt) We only submitted once to the leaderboard. Our final model is the one described in section 2.4, which leverages interaction information, attention, and state information. It achieves 0.7% test interaction level accuracy. The instruction-level and interaction-level development set results of all of our model architectures/configurations are included in Table 2.

Model	Features	Instr Acc	Inter Acc
Basic	nothing special	17.63%	0.4%
State-info	State-encoding method: RNN	24.65%	0.4%
	State-encoding method: one-hot encoding	24.82%	0.4%
Attn	State-encoding method: one-hot encoding Attention: basic dot product	24.1%	0.4%
	State-encoding method: one-hot encoding Attention: Bahdanau Attention	25.5%	0.4%
Interact-info	State-encoding method: one-hot encoding Attention: Bahdanau Attention	25.6%	0.8%

Table 2: Model Performance Summary

Ablations (13pt) We discuss ablations only for the models with the best instruction-level performance for each of the four model architectures, since the models interaction interaction-level accuracies are all basically zero.

1. **Effect of state information**. Compared to the basic sequence-to-sequence model, the state-info model encodes the state and concatenates with the input token at every step of the decoder to generate output, which enables the model to understand what phrases would refer to, such as 'the red beaker'. We tried two different state-encoding mechanisms: one-hot vector and RNN beaker encodings. We tried using a single-layer forward-only LSTM to encode each of the seven beakers, concatenating each of seven final RNN output vectors to form the state context. From our experiments, the one-hot vector encoding performed the best, boosting instruction level accuracy from 17.55% to 24.82%, a marginal increase of 41.42%.
2. **Effect of Attention**. Attention gives our model the ability to learn to focus on the most appropriate parts of the encoder outputs, for action generation, by computing a different weighted combination of the encoder outputs at each time step to form the instruction context vectors. We tried two types of attention mechanisms: basic dot product attention (the exact version described in the lecture slides for April 21) and Bahdanau attention. From our experiments, Bahdanau attention yielded better performance, boosting instruction level accuracy from 24.82% to 25.5%, a marginal increase of 2.74%, compared to not using attention.
3. **Effect of interaction-embedding**. The inclusion of previous instruction encoding on top of the attention model does not show significant improvement in boosting the instruction-level accuracy performance, unfortunately, even though it theoretically should let our model understand references such as *it*.

7 Error Analysis (7pt)

Off by one error: Many times, our model would get many instructions incorrectly because of a minor error in the interpretation of the first instruction. This mistake would persist through the remaining instructions and cause them to error. For example our dev-1836-1 prediction was: "1:p 2:p 3:o 4:_ 5:r 6:yp 7:o", while the dev-1836-1 actual was: "1:_ 2:p 3:o 4:_ 5:r 6:yp 7:o". All the predictions after were also incorrect.

Misinterpreted world state: Our model had difficulty interpreting words that represented the world state such as "it", "that", and "one". With the dev-1830-4 instruction: "Mix that too", our model added blue to beaker 7 instead mixing the green in with the blue in beaker 7. With the dev-1840-2 instruction: "It turns brown", instead of turning beaker 7 from orange and yellow to brown, our model placed brown in beaker 4.

Action-to-state conversion failures: In our development data, 5% of decoder output actions were converted to a None state. In our test data, 61.8% of decoder output actions were converted to a None state. If there is an invalid action at one time step and our model emits None for that utterance, it results in a cascade of errors that results in even more None states in future actions.

8 Conclusion (3pt)

We experimented with four different model structures for the task of mapping instructions to actions in the ALCHEMY environment. We tried a basic sequence-to-sequence model, a state encoded model, a state encoded model with an attention mechanism, and finally, a state encoded model with an attention mechanism and access to interaction history (our full model). Our full model achieved a dev instruction level accuracy of 25.6%, a 0.8% dev interaction level accuracy, and a 0.7% test interaction level accuracy. Sadly, our model did not perform as well as we wanted. We spent dozens of hours re-implementing our models and looking for bugs, but we no matter what we tried, we never saw better performance.