Spring 2018 CS 5740: Assignment 4

GITHUB GROUP: MASTER-ALCHEMIST https://github.com/cornell-cs5740-18sp/assignment4-master-alchemist

Yubin Xie
yx443

Disheng Zheng
dz336

Xiang Niu
xn22

1 Introduction

Task: In this assignment, we implemented several different sequence to sequence models to map natural language instructions to actions in a small environment of Alchemy where there are seven beaks, each beaker contains one or more colors or empty. Our task is to predict actions from language instructions and execute the actions to construct state of seven beakers.

Data: The data contain three information: identifier, a unique identifier (e.g. *train-A9164*); initial_env, the representation of the initial environment (e.g. *1ggg 2_ 3_ 4_ 5o 6ooo 7gggg* represents the initial state of the seven beakers such that beaker 2 to 4 are empty, beaker 1 is filled with three green color, beaker 5 is filled with one orange color and beaker6 is filled with three orange color and beaker 7 is full with all four portions filled with orange); and utterances, a list of utterances that take place. Utterance is a dictionary with keys (e.g. *"throw out two units of first beaker"*); actions, (e.g. action of *"pop 1, pop 1"* corresponds to the instruction of *"throw out two units of first beaker"*); and after_env, a representation of the environment after taking the actions, which is in same format as the initial_env.

Evaluation: We evaluated the system performance using two metrics: single-instruction task completion and interaction task completion. For single sentence, the goal state is the annotated final state of executing this instruction. For interaction performance, the goal state is the annotated state following the execution of all the instructions.

2 Approach

2.1 Sequence-to-Sequence Model

In this task, we implemented a sequence to sequence model (seq2seq) and aimed to learn actions corresponds to each language instruction and the current states. The seq2seq model contains two neural networks: an encoder and a decoder. The encoder is capable of encoding all the instructions into a context vector which serves as the initial starting material for the decoder. The decoder takes the context vector learned by encoder and integrates the world state of the beakers to predict the actions. We further enhanced this simple seq2seq model by using attention system that creates a weighted sum of both encoder and previous decoder outputs as the input for the next decoder. The Seq2Seq model in this task translates the instruction space to the action/state space. The basic structure of encoder and decoder described earlier is shown here in the figure. The encoder first takes the each word sequentially in the instruction space, and passed through its LSTM network and save the hidden state from each step until it meets a end of sequence tag. Then the last encoded hidden state vector will be used to represent the entire sequence. This very last state vector also serves as the first input for the decoder. The decoder is another LSTM network that initialized by the last encoder state. It starts to generate and output the most likely action labels at each step.

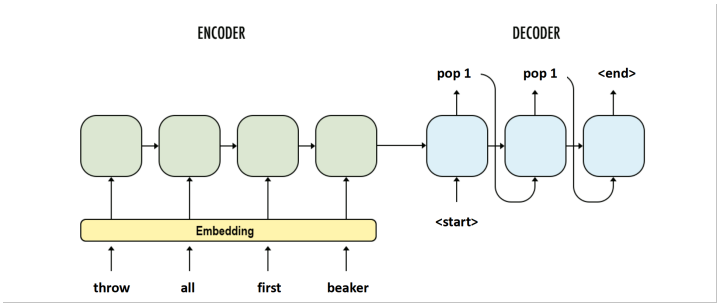
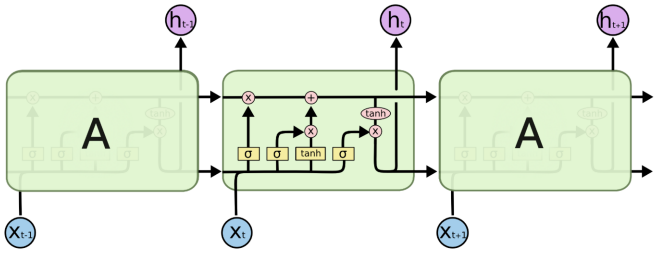


Figure 1: Seq2Seq Model Representation



The repeating module in an LSTM contains four interacting layers.

$$\mathbf{f}_t = \sigma(\mathbf{W}^f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}^f)$$
$$\mathbf{i}_t = \sigma(\mathbf{W}^i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}^f)$$
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tanh(\mathbf{W}^c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}^c)$$
$$\mathbf{o}_t = \sigma(\mathbf{W}^o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}^o)$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$

Figure 2: LSTM Networks

LSTM stands for Long Short Term Memory networks. A LSTM networks is a variation of RNN which is capable of learning long distance relationship by significantly reducing the vanishing gradient of the previous long distant passes. A LSTM network incorporate current state with the memory information by learning through combination of update gate and forget gates to create a new hidden state vector. LSTM networks are suitable for our task since the instructions and actions can be influenced by long term memories. Here x_t is the input. h_t is the output of the current hidden state. h_{t-1} is the output of the last hidden state. c_t is the cell state. Information is flowing from the input, last cell state, and last hidden state. W are the weight vectors and b are the bias vectors. i_t is the input gate; f_t is the forget gate; o_t is the output gate. The three gates are to limit and to filter information flow. Therefore, they use the σ function which is in range of [0, 1]. Input gate filters new information from input and last hidden state. Forget gate filters information from the last cell state. Together, the form the current cell state. The output gate filters information flowing from current cell state to the

current hidden state (output).

2.2 Finite State Machine

Finite state machine is a classical way of representing state that consists of states, vocabulary, start token, and transition function. Typical tradition functions are generator, acceptor, transducer, and encoder. We used finite state automata (FSA) for transitions of our decoder's output (actions) given initial world state to after instruction world state. These codes are given by the instructor.

3 Experimental Setup

3.1 Data preprocessing

We first take a glance at the data and cleaned up the data by removing punctuations in the data. There are 18285 instruction and action pairs in training. We obtained 612 unique words in instruction and 51 unique actions. There 1225 instruction and action pairs in the development set. We noticed that there are rare words that corresponds to typos, for example there is one copy of *bown* and *brow* which should be correctly spelled as *brown*. We set the rare words that appeared less than ten times as unknown words and given them *unknown* tags. In total, there are 933 unknown words in the training data.

3.2 Adding State Information

State information, especially the initial state, is important to predict action. The understanding of some instruction, e.g. *throw out one unit of green*, requires the knowledge of which beaker has green chemicals (beaker position and contents). In order to represent the environmental states as vectors, we implemented the following two embedding schemes.

Embedding with One hot encoding: In this method, we do one hot encoding for each beaker and then concatenate 7 beakers information. For each beaker there is a corresponding *4*6* matrix where the four columns are all possible positions in a beaker and the six rows are all possible colors. If there is a specific color j in a position i, then [i,j] will be one, otherwise will be zero. A total of *7*4*6* length vector is used to represent each state. This 168-length vector is then concatenated to each input of the decoder LSTM networks.

Embedding with LSTM encoder: The other method is to use LSTM to encode the state information of each beaker. The input of network is the content of beaker, e.g. for environment *1:gbg*, the input is *g ,b, g*. Then we concatenate each beaker's position embedding to the output of this state encoder. Now that we have 7 beakers' embedding vector, we apply attention model function to allows the decoder to pick the encoded inputs that are important for each step. The output is then concatenated to each input of the decoder LSTM networks.

3.3 Adding Attention

Instead of incorporating only the output of encoder final state when generating the decoder hidden state, a weighted vector representing each stage of the encoder should be considered to help the decoder to know which part of the encoding is relevant at each step of generation. We have encoded two attention into our model, one attending on the output of instruction encoder, the other on position-concatenated output of environment encoder. This allows our model to train itself to weight features that best minimizes the losses from prediction generated by each decoder state. Also, we attempted two different attention function, dot product and MLP.

For dot product:

$$importance_{ij} = decoderstate_{j-1} * c_i$$

For MLP:

$$importance_{ij} = V * tanh(c_i * W_1 + decoderstate_{j-1} * W_2)$$

Here $importance_{ij}$ is the importance of encoded vector i at decoding step j. W_1, W_2 and V are learned weight parameters. c_j is the output attending vector for each decoding step j, c_i is the softmax optimized input attending vector for encoded vector i.

For both models, attentionWeignts$_j$ =

$$softmax(importance_{1j}, ..., importance_{nj})$$

$$c_j = \sum_{i=1}^{n} attentionWeignts_{ij} * c_i$$

$attentionWeights_{ij}$ is the attention weight of encoded vector i at decoding step j.

3.4 Adding Interaction History

The interpretation of current utterance sometimes relies on previous utterance in the same experiment. E.g., in train-A9165, instruction 2:*throw out three units of third one*, instruction 3: *pour fourth beaker into it*. The history utterance is necessary to interpret the word 'it' as 'third one'. To add interaction history, we concatenate the previous utterance to current utterance (we do not add anything to the first utterance) and then use the concatenated utterance as the input of encoder.E.g., together with stop token, the input instruction of 3^{rd} utterance for train-A9165 is *throw out three units of third one < end > pour fourth beaker into it < end >* . The stop token gives separation information and the attention function will focus on relevant token for each decoding step. What's more, after each utterance within an experiment and each action within an utterance, we update the previous environment to current environment.

3.5 Rule-based greedy algorithm

In the decoder network, we generate the probability distribution of all actions given the input based on the current state. During training, we picks the maximum likelihood action as the candidate output (greedy algorithm), and then use this output as part of the next state's input. As for prediction part (post-training), we add a rule on the greedy algorithm: **operability**. Given an initial environment, not all the actions are operable. As an instance, you can't operate *pop 2* on environment *1ggg 2_ 3_ 4_ 5o 6ooo 7gggg*. An inoperable action is for sure a wrong prediction. During prediction, we will evaluate the operability of the maximum likelihood action. If it is not operable, we will do evaluation on next most likely action. We keep evaluating until we found an operable action, which is the maximum likelihood operable action. This helped our best model performance on dev. data instruction/interaction increases from 0.579/0.322 to 0.583/0.339.

3.6 Final Model

Figure 3 presents our final model architecture. The input of encoder 1 is the instruction and the input of encoder 2 is each

beaker's state. After attention model, the output of encoder 1 and the output of encoder 2 that concatenated with position information, together will join the previous decoder's output to become the input of current decoder state.

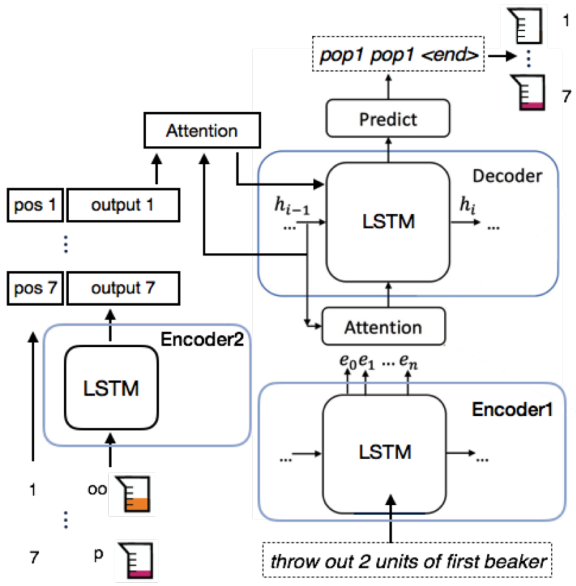


Figure 3: Representation of final model architecture

Hyper-parameter: *Builder*: all LSTM. *Layer dimension*: all 1. *Embedding dimension of instruction, action, beaker's state*: 50. *Embedding dimension of position*: 25. *Hidden dimension of encoder 1/encoder 2/decoder*:100/75/100. *Attention dimension*: 100. *Optimizer*: SimpleSGD. Batch size: 5. Dropout rate of input/output: 0/0.05.

Results: Accuracy on instruction/interaction level in Dev. set is 58.2%/33.9% and accuracy on interaction in test set 55.23%

4 Results and Analysis

In this section, we present our best results (best instruction accuracy within 100 epoch) on different experimental setting and hyper-parameter tunning. **All the model architecture and parameter are same as final model unless specified the type of ablation.**

4.1 Tasks

Table 1 Accuracy with different Task Completion

Parameter	Instruction	Interaction
seq2seq w\out state	18.1%	2.04%
w\ state w\out att.	50.6%	29.4%
w\ state, att. w\out his.	53.9%	29.0%
w\ state, att. hist.	58.3%	33.9%

We first tested the performance of each task, in the *"w\out his"* setup, there is no history information being used but only one instruction sentence being inputed to the model; in the *"w\out state"* setup, there is no state embedding being inputed to the decoder; in the *"w\out att"* setup, there is no attention being added to both state embedding and instruction encoders, however we still keep the state embeddings using LSTM network. Our result showed that state information is the most important component needed for good performance. If initial state information is not correctly incorporated, the decoder only produces an accuracy of 18.1% and 2.04% on the instruction and interaction evaluation. After including the initial state embedding information, we saw a great boost in performance to 50.6% and 29.4% respectively. We also saw a slight improvement when after adding the attention. We also see an improvement if we added the history information from 53.9% to 58.3% in the instruction and 29.0% to 33.9% in the interaction evaluation. This improvement can be explained as that the past information from the last instruction is also important for predicting the current execution. Intuitively, when the instructions contain ambiguous and changing expressions like *"the orange beaker"*, more information from the past instruction might help to define what the *"orange beaker"* referred to at current environment.

4.2 State embedding and Attention

Table 2 Accuracy of state embedding and attention

Embedding	Instruction	Interaction
one hot / dot	41.6%	15.5%
mlp / dot	58.3%	33.9%
mlp / mlp	54.8%	32.3%

We also explored different state embedding and attention mechanisms. The *"one hot"* stands for the one hot encoding of states information as described in previous section. The *"dot"* and *"mlp"* corresponds to dot product and multilayer perceptron (MLP) methods for calculating attention. We found that the greatest improvement comes from the state embedding and attention method. The simple one hot encoding of the environmental states doesn't work as well as the LSTM with attention approach. In general attention methods of both dot product and multilayer perception performed well in both instruction and state embedding encoders. The combination of MLP and dot product attention on state embedding and instruction encoder gives the best performance of 58.3% and 33.9% accuracy in development instruction and interaction accuracy.

4.3 Builder

Table 3 Accuracy of different builders

Builder	Instruction	Interaction
LSTM	58.2%	33.9%
GRU	48.7%	22.4%

To our surprise different builders also performs quite different. Both GRU and LSTM are closely related to RNN. They both use several gating conditions to incorporate information flows from long term memory and thus preventing vanishing gradient. The difference is subtle that recent developed GRU has less parameters compared to LSTM and is efficient to compute. However we observed in this task the LSTM networks outperforms the GRU in the development set accuracy.

4.4 Optimizers

Table 4 Accuracy of different optimizers

Optimizer	Instruction	Interaction
RMS	49.7%	24.4%
Adam	55.8%	32.2%
SimpleSGD	58.3%	33.9%

We have observed that both Adam optimizer and simple gradient decent optimizer perform pretty well. And simple gradient descent optimizer was selected for our final model. We also noticed that Adam optimizer is more efficient compared to others and learned faster in the early epochs, but after 100 epoch the simple gradient descent optimizer achieved the best development set accuracy.

4.5 Batch size

Table 5 Accuracy of different batch sizes

Batch Size	Instruction	Interaction
1	50.7%	25.9%
5	58.3%	33.9%
10	37.2%	13.1%
50	48.6%	22.3%
100	39.9%	13.4%

We also tested the performance of different batch sizes. We kept all other parameters the same as the best model, which are seq2seq with state embedding and history interaction, mlp and dot product attention on state embedding and encoder, LSTM networks for states embedding, encoder and decoder, Adam optimizer. Therefore we tested only for batch size changes. We observed that empirically batch size 5 performs the best. Small batch size of 1 also gives good results, but the bigger batch size of 10, 50 and 100 give worse performance which may suggest that small batch is needed for this task.

4.6 Error analysis

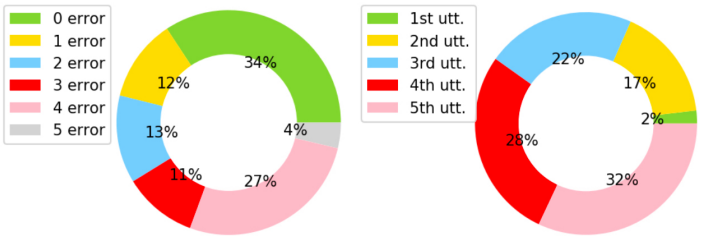


Figure 4: Distribution of error number (left) & position (right)

We observed that 34.3% of predictions in the five utterances are correct while there are also some cases (26.9% + 3.67%) where four or all five utterance were predicted wrong. This bimodal distribution of errors showed that our model performs well on majority of the instructions, but also creates frequent mistakes. We also looked in to the relative positions when the mistakes where made in the five utterance. We found that our model is good at predicting the initial state, only 2% error were made, but does not perform well in the fourth and fifth instructions, 26% and 32% errors respectively. The mistakes showed accumulating patter in general. This may be due to our history setup that we only include the one previous history. The model may not be enough to learn the previous memory at the fourth and fifth instruction. We also saw a 67% error if the previous state is predicted wrong and we get subsequent wrong as well based on the previous state environment information. This also explained the pattern of accumulation of errors.

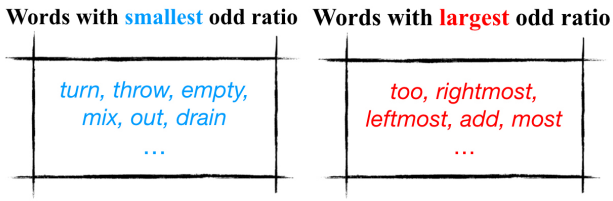


Figure 5: Selected words from largest and smallest odd ratio

Then, we also looked into the specific instructions that lead to error in predicting the action states. Among the 1245 environment states in the dev. set, there 161 environments were firstly predicted wrong in their own experiment (first error in 5 utterance tends to be an error from model while the rest can be induced by previous error). We defined odds ratios for each instruction as $odds(word_i)$:

$$\frac{error\ number(word_i)}{first\ error\ number} \times \frac{total\ error\ number(word_i)}{total\ word\ number}$$

The statistical analysis of all the instruction words showed that *turn, throw, empty, mix, out, drain* are in the top 10 smallest odds ratio list where odd ratio smaller than 0.4 and *too, rightmost, leftmost, add, most* are in the top 10 largest odds ratios list where odds ratio is larger than 2. We found that instructions with *empty, out, turn, throw, mix, drain* encode one simple action or action within a beaker, they are more likely to be predicted correctly. However, instructions with *too, rightmost, leftmost, add, most* require model to 1) infer and compare different beaker condition and 2) require history information. These are hard to be predicted correctly by our model. What's more, we also observe that if the instructions contain two or more actions like *pour, add, to, into* which needs interaction between two beakers, it is also more often to see errors (odds ratio above 1.5).

5 Conclusion

To summarize our implementation, the final sequence to sequence model presented above was able to achieve a instruction/interaction accuracy of 58.2%/55.23% in the development data. The greatest improvements in our implementation come from adding state information, attending on color states and instructions, and adding history interaction utterance. We have experimented on different batch size, builder, attending method, and optimizer, because after all, fine tuning is the key to exploit. neural networks.Due to the limitation of time, some of our ideas are not implemented. In the future, beam search can be applied to find the global optimal for each experiment that contains 5 utterances. Current greedy algorithm can only find local optimal at each utterances and beam search should help improve the prediction accuracy by considering more choices that might reach global optimal. At the same time, another attention mechanism can be applied on the previous utterance state output as a better method for history catching, which then become part of the input of decoder.