Practical 5: Task-Oriented Dialogue System

Deep Natural Language Processing Class, 2023

Report due date - 11.05.2023

The practical report (pdf + code) should be send via Moodle/Pegaz with a standard due dates policy.

1 Neural Task-oriented Dialogue System

Task-oriented dialogue systems are a part of conversational systems and their goal is to assist user at achieving daily goals. In this practical, we will implement a sequence-to-sequence (seq2seq) network, to build a neural task-oriented dialogue system system. It will be a greatly simplified proxy of current production-oriented models. First, let's describe the general data pipeline that a model process in order to output a text answer.

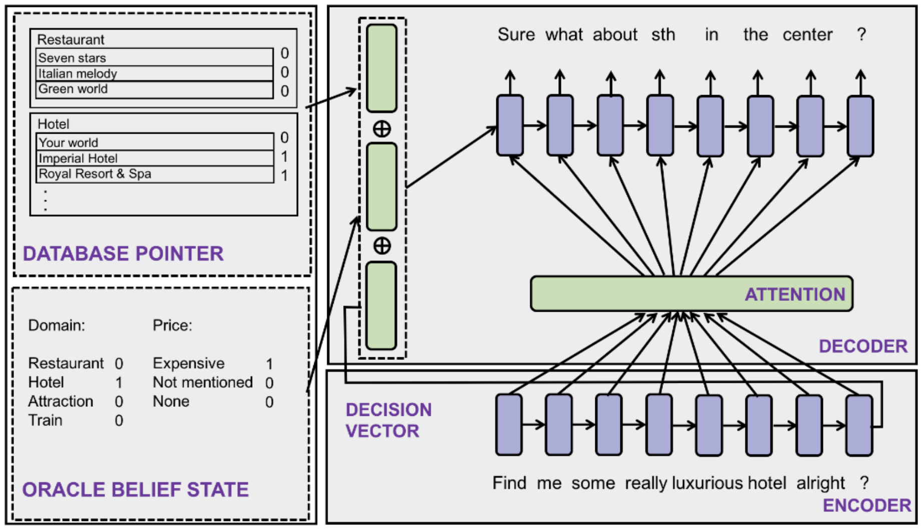

Figure 1: An architecture of the multi-domain dialogue system composed of three sources of input: 1) the oracle belief state, 2) the database pointer and 3) the dialogue context.

1.1 Encoding Data

At each turn, the encoder takes a sequence of input tokens $u_t = (w_0^t, w_1^t, ..., w_L^t)$ and uses a (bi-directional) recurrent neural network to output a distributed user utterance representation

$\mathbf{u}_t$ which is the final hidden state:

$$\mathbf{u}_t = \mathbf{h}_L^u = \text{RNN}_\theta(u_t).$$

The encoder is conditioned on two additional models which are essential for long-term dialogue modeling: 1) dialogue state tracking (DST) and 2) the knowledge base querying component. The summarized dialogue state is formed by concatenating three probability values: 1) a probability that the slot has not been mentioned until this turn; 2) a probability that the user does not care about this constraint; and 3) a sum of all other values for the given slot. This yields a 3-bin one-hot encoding $\mathbf{b}_{s_d,t}$ for each slot, where d signifies the domain (for example `Hotels` or `Attractions`), s_d a slot from the domain (for example `cuisine type`, `time` or `place`) and t is a turn indicator. The global dialogue/belief state vector is then formed by concatenating all slot-dependent vectors over all domains:

$$\mathbf{b}_t = \bigoplus_{d \in \mathcal{D}} \bigoplus_{s_d \in \mathcal{S}_d} \mathbf{b}_{s_d,t}, \tag{1}$$

where $\mathcal{S}$ is a set of of all slots in a domain.

The current belief state $\mathbf{b}_t$ can be used to query the knowledge base. The knowledge base consists of the list of entities (for example restaurants) with characteristics like address, price-range or cusine type. Based on the numbers of entities in the database that satisfy the current belief state, we form n-bin one-hot encodings for each domain $\mathbf{k}_d$. In all experiments, we use 6-bin encodings for 0, 2, 5, 10, 40, or more than 40 matches. All domain-dependent vectors are then concatenated into a global domain vector:

$$\mathbf{k}_t = \bigoplus_{d \in \mathcal{D}} \mathbf{k}_{d,t}. \tag{2}$$

Additionally, once the sought entity is provided by the system, the user might *request* additional information like `phone number` or `address`. This information is stored in 1-bin one-hot encoding

$$\mathbf{r}_t = \bigoplus_{d \in \mathcal{D}} \mathbf{r}_{d,t}. \tag{3}$$

which is added to the original belief state $\mathbf{b}_{s_d,t}$.

1.2 Decision Making

In the next step, a policy vector is created to mimic the dialogue manager in the traditional modular approach. The intent vector $\mathbf{u}_t$, the belief state vector $\mathbf{b}_t$ and the knowledge database vector $\mathbf{k}_t$ are combined together and processed through a nonlinear layer:

$$\mathbf{a}_t = \tanh(\mathbf{W}_u \mathbf{u}_t + \mathbf{W}_k \mathbf{k}_t + \mathbf{W}_b \mathbf{b}_t). \tag{4}$$

This vector can be seen as a continuous version of a system act in the traditional modular approach summarizing the current state and action in a high-dimensional space.

1.3 Generation

The generation module uses a language model in the form of the recurrent neural network that outputs probabilities over the vocabulary set at each time step:

$$P(w_{j+1}|w_j, \mathbf{h}^m_{j-1}) = \text{softmax}(\text{RNN}(w_j, \mathbf{h}^m_{j-1})),$$

where w_j is the last output token and $\mathbf{h}^m_{t-1}$ is the hidden vector from the previous step. We condition a language generator through the action vector $\mathbf{a}_t$ by using it as the first hidden vector, i.e:

$$\mathbf{h}^m_0 = \mathbf{a}_t.$$

To begin the generation process we use a special token (signifying the beginning of a sentence, SOS) as w_0. The generation process stops when the network outputs a special token informing about the end of the sentence (EOS). The standard cross entropy is adopted as our objective function to train a language model:

$$L(\theta) = \sum_t \sum_j y^t_j \log p^t_j,$$

where y^t_j and p^t_j are output token targets and predictions respectively, at turn t of output step j. In our case every token is treated equally to make the model as simple and as general as possible.

2 Neural implementation

In this practical, we will only work with the `RESTAURANT` domain from the WOZ2 dataset. The system is designed to assist users to find a restaurant in the Cambridge, UK area. There are three informable slots (food, pricerange, area) that users can use to constrain the search and six requestable slots (address, phone, postcode plus the three informable slots) that the user can ask a value for once a restaurant has been offered. There are 99 restaurants in the database.

a) (1 point)

Create a one-hot encoding of the belief state as described in Equation 1 - fill the `belief_state` function in the `create_delex_data.py`. The belief state should consists of 3 numbers per slot. Each 0-1 number represents whether the slot was 'not mentioned' or the user 'dont care' or the slot has specific value from the ontology. The whole belief state should have 12 slots. The implementation should not take more than 10 lines.

You can test your implementation by running a non-exhaustive test:

```
python test_exercises.py --task a
```

b) (1 point)

Create a one-hot encoding informing how many entities are available to the user in the `one_hot_vector` function from `utils/db_pointer.py`. The output list should consists of 6 buckets following the ideas from Equation 2. The implementation should not take more than 15 lines.

You can test your implementation by running a non-exhaustive test:
`python test_exercises.py --task b`

c) (1 point)

Create a one-hot encoding of requested slots defined in `requested_state` in `create_delex_data.py`. If the slot was requested, you should mark it as a 1 or otherwise 0. The implementation should not take more than 5 lines.

You can test your implementation by running a non-exhaustive test:
`python test_exercises.py --task c`

d) (2 points)

When working with large vocabularies of specific name entities, it's often useful to apply *delexicalization* - a process of replacing slots and values by generic tokens (e.g. keywords like Chinese or Indian are replaced by `SLOT_FOOD` to allow generalization during generation).

Finish the `prepare_slot_values_independent` function in the `utils/delexicalize.py`. Add tuples to the delexicalization list consisting of normalized value and the delexicalized equivalent. For example when the key is `AREA` and the value is `center` you should add a tuple `(center, [value_AREA])`. The implementation should not take more than 7 lines.

You can test your implementation by running a non-exhaustive test:
`python test_exercises.py --task d`

Create delexicalized data

Once you finish all above exercises you can should run: `python create_delex_data.py` to create delexicalized data.

e) (2 points)

Finish the forward pass of the `forward` function in the `model/model.py`. You should use the Encoder, Policy, and Decoder networks. Check what these classes need as input. All necessary variables are prepared for you already.
After that, you can train the new policy head by running: `python train.py`. There are

three metrics we look at: whether the system has provided an appropriate entity (Inform rate) and then answered all the requested attributes (Success rate) while fluency is approximated via BLEU score.

Report the performance (BLEU, Inform and Success rates) by running: `python test.py`.

f) (2 points)

One can also create a policy head to choose the best action through a softmax layer to force the model to define a latent action space (of the size `act_num`):

$$
\begin{aligned}
\mathbf{a}_t =& \mathbf{W}_u\mathbf{u}_t + \mathbf{W}_k\mathbf{k}_t + \mathbf{W}_b\mathbf{b}_t \\
\mathbf{a}_t =& \mathbf{W}_1(\mathbf{a}_t), \mathbf{W}_1 \in \mathbb{R}^{h\times h} \\
\mathbf{a}_t =& \mathbf{W}_2(\mathbf{a}_t), \mathbf{W}_2 \in \mathbb{R}^{h\times act\_num} \\
\mathbf{o}_t =& \text{softmax}(\mathbf{a}_t) \in \mathbb{R}^{1\times act\_num} \\
\mathbf{a}_t =& \mathbf{W}_e(\mathbf{o}_t), \mathbf{W}_e \in \mathbb{R}^{act\_num\times h}
\end{aligned}
$$

Define the policy with the latent action space head from above equations in `model/policy` in the `SoftmaxPolicy` class.

You can test your implementation by running a non-exhaustive test:
`python test_exercises.py --task f`

You can now train the new policy head by running: `python train.py --policy softmax`. Report the performance by running (BLEU, Inform and Success rates): `python test.py`. Can you see any differences in results compared to the point e)?

g) (2 points)

What do you think of using a BLEU score as a way to evaluate dialogue systems? Can you evaluate the model only relying on this metric?

3 Prompt-based Conversational AI

The power of large language models allows to start building much more powerful systems. In this section, we will try to build a simple system using only the power of in-context learning.
The structure of the prompt should have a following logic:

```
{PROMPT DESCRIPTION}
{CONVERSATION_HISTORY}
{LAST_TOURIST_TURN}
{LLM ANSWER}
```

For example:

```
You are a travel information agent with an access to Cambridge databases about
restaurants, taxis and hotels. The entities in the database are:
Pizza Hut City Centre, italian, cheap
...
A and B guest house, moderate, 4 stars.

The example dialogues are:

Tourist: "i want to find a restaurant in the center part of town
and serves italian food ."
Tourist_delex: "i want to find a restaurant in the [value_area] part of town
and serves [value_food] food ."
System: there are several Italian restaurant -s .
do you have a price range in mind ?
System_delex: there are several [value_food] restaurant -s .
do you have a price range in mind ?
...

Here is the current dialogue you should serve:
System: Hello
System_delex: Hello
Tourist: Hi,  I need ..
Tourist_delex: Hi, I need ..
... (all turns that have happened so far)
Tourist: Ok, I would choose candian then
Tourist_delex: I would choose [value_food] then
System_delex:
```

You have 5k token limit for the whole prompt (without the dialogue that will be provided at the test run). You can test the length of the prompt here (https://platform.openai.com/tokenizer)

a) (9 points)

Create a prompt that will answer questions about the restaurants. Try to provide database information, some examples of the conversations (however, do not utilize testing dialogues). For testing purposes you can use free access to the ChatGPT.

Enclose in submission a text file `gpt_system.txt` with a text prompt. We will run submissions with GPT4 and provide you with the results while scoring the practical. The testing protocol will follow the logic from Section 2.