GZZ chatbot - a transformer-based chatbot

Limian Guo, Xiangran Zhao, Fengzhencheng Zeng

Introduction

With the rapidly growing population of messaging app users, the demand for imitating various kinds of human conversations also increases. Being part of the progress of artificial intelligence evolution, chatbots arose as a solution to meet the demand described. A chatbot is an automated system of communication with human users via auditory or textual methods. It allows humans to interact with digital devices as if they were communicating with a real person.

Most of the invented chatbots fall into two main types: task-oriented (declarative) chatbots, and data-driven and predictive (conversational) chatbots. The task-oriented chatbots aim to achieve the single function of generating automated but conversational responses to highly specific and structured user queries, using rules, NLP and ML knowledge. The data-driven and predictive chatbots, however, can deal with way more sophisticated conversations based on analytics of user-profiles and past behaviors. Our chatbot is much closer to the first type.

To build our chatbot, we adopted a newly proposed network architecture, the Transformer, which has been demonstrated to have the best performance. We trained the transformer model based on attention mechanisms and benefited from its parallelization speeding-up characteristic during training. Besides the transformer, we also used the most advanced NLP techniques, word2vec, and the most advanced WEB tech stack to equip our chatbot with excellent user experiences.

Implementation

Data

Dataset

To prepare for the corpus of dialogs as training data for our model, we used the Taskmaster-1 English dialog datasets released by Google in September 2019. The dataset of Taskmaster-1 was created to serve the goal that high quality, goal-oriented conversational data are required to build a dialog system. It includes 13,215 task-based dialogs in English, comprising 5,507 spoken and 7,708 written dialogs. There are six domains covered by those conversations: ordering pizza, creating auto repair appointments, setting up ride service, ordering movie tickets, ordering coffee drinks and making restaurant reservations. The spoken dialogs were made by pairing one crowdsourced worker with a trained call center operator via the Wizard-of-Oz

platform. For the written dialogs, full conversations were written by crowdsourced workers based on task-oriented situations. Since the workers were not restricted to detailed scripts when writing those conversations, speaker diversity and corpus size both grow up as benefits.

A closer look at the data provided by Taskmaster-1 reveals a lot of information. Following is an example of the data format in Taskmaster-1 files.


```
[
  {
    "conversation_id": "dlg-00055f4e-4a46-48bf-8d99-4e477663eb23",
    "instruction_id": "restaurant-table-2",
    "utterances": [
      {
        "index": 0,
        "speaker": "ASSISTANT",
        "text": "Ok, what area are you thinking about?"
      },
      {
        "index": 1,
        "speaker": "USER",
        "text": "Somewhere in Southern NYC, maybe the East Village?",
        "segments": [
          {
            "start_index": 13,
            "end_index": 49,
            "text": "Southern NYC, maybe the East Village",
            "annotations": [
              {
                "name": "restaurant_reservation.location.restaurant.accept"
              }
            ]
          },
          {
            "start_index": 13,
            "end_index": 25,
            "text": "Southern NYC",
            "annotations": [
              {
                "name": "restaurant_reservation.location.restaurant.accept"
              }
            ]
          }
        ]
      }
    ]
  }
]
```


To explain the structure of each conversation as shown above:

conversationId: A universally unique identifier with the prefix ‘dlg-‘.

utterances: An array of utterances that make up the conversation.

instructionId: A reference to the file(s) containing the user and instructions for the conversation.

index: A 0-based index indicating the order of the utterances in the conversation.

speaker: Either USER or ASSISTANT, indicating which role generated this utterance.

text: The raw text of the utterance.

segments: An array of various text spans with semantic annotations.

startIndex: The position of the start of the annotation in the utterance text.

endIndex: The position of the end of the annotation in the utterance text.

text: The raw text that has been annotated.

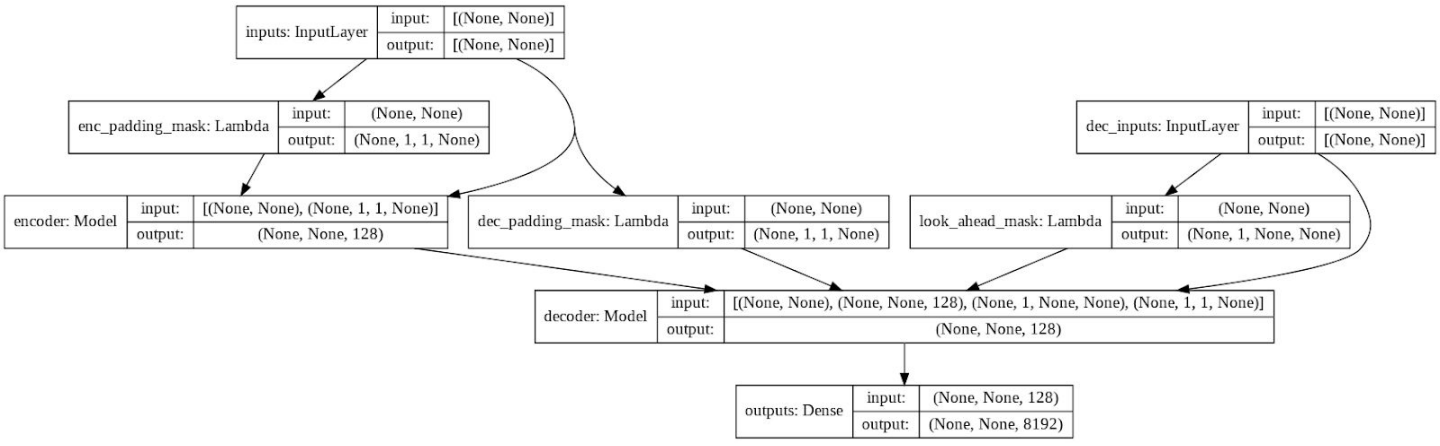
annotations: An array of annotation detailed for this segment.

name: The annotation name.

Preprocess

We made use of both the woz (spoken) and self (written) dialogs datasets. Those two datasets were stored in JSON files, which could be easily loaded with the python JSON module for later processing. Values in the ‘text’ field of ‘utterances’ were extracted for processive tokenization and filtering. Each utterance was viewed as an output response to the previous input, and input for the future output as well. Inputs and outputs were stored as separate lists of sentences. To tokenize sentences, we adapted SubwordTextEncoder from the tfds.features.text module. Together with the start and end tokens, the SubwordTextEncoder tokenizer instance saved in the ‘tm_tokenizer.pkl’ file for later use. In order to feed data to the training step in an efficient way, we stored tokenized inputs and outputs data in an instance of tf.data.Dataset class. It is ideal for representing a large set of elements from the input pipeline, and a logical plan of transformations that act on those elements.

Model



An illustration generated by keras.plot_model [3]

Word Embedding

Traditionally we might use a one-hot encoder for encoding input. Then input will be extremely huge thus hard and slow to train. And we will also abandon the relationship between words which will compromise our results. So for this project, we use embedded encoding which is not only smaller but also can maintain the relationship between words.

Positional encoding

Unlike the Seq2Seq model, the attention mechanism used in the transformer model does not keep any information on the sequential relationship among words in a sentence, which means that the output of the transformer model is out of order if we do not solve this problem. Positional coding adds positional information to the embedded representation of words so that the order of words can be kept. For this implementation, alternative sines and cosines with radians corresponding to the position are used.

Multi-head attention

Multi-head attention is the central component of the transformer model. It uses a self-attention mechanism instead of RNNs in the Seq2Seq model to predict the output. With the self-attention mechanism, the major problem of RNN, gradient vanishing, is completely avoided.

For this implementation, the scaled dot product was used as the self-attention mechanism. The embedded representation of each word in the sentence is transformed into three vectors, which are key(K), value(V), and query(Q), through three separate fully-connected layers. The dot product of K and Q is used to assess how a word is associated with another word. For example, Ki · Qj means the importance of a word i to a word j. That product, normalized by sqrt of the dimension of the word embedding is used to calculate the relative associations via softmax function. The results of softmax are multiplied with the value of words to get final attention.

Instead of using a single scaled dot product attention layer, the transformer model uses a multi-head version, which splits the K, Q, V vectors into several vectors and does the scaled dot product separately. This is said to enhance the ability of the model to focus on multiple words and “given the attention layer multiple ‘representation subspace’”.

Encoder

The encoder starts with the word embedding layer which contains a TensorFlow embedding layer and a positional encoding layer. The embedding layer is followed by customizable (default to 6) layers of multi-head attention which is followed by two dense feed-forward layers (named as encoder layer). In these 6 layers, some shortcuts are applied to avoid degradation problems. There are also several drop-out layers added to prevent overfitting.

Decoder

The decoder also starts with the word embedding layer which contains a TensorFlow embedding layer and a positional encoding layer. Followed by that are customizable (default to 6) layers of multi-head attentions and dense feed-forward layers (named as decoder layer in the code). The difference between the Edecoder layer and the encoder layer is that the decoder layer has two multi-head attention layers. One is for the decoder input and one is for the encoder output. Dropout layers and shortcuts are also added as the encoder.

Transformer model

The transformer model implementation we use is from the reference's notebook with the correction in positional encoding and adapted to support our WEB system.

Application

Server

This chatbot is implemented in WEB stacks. The back-end is using a lightweight server Flask for easy integration with Tensorflow trained model.

One advantage of using a web structure is that training thread will never block user thread which will have a better user experience.

UI

UI was built using React and integrated well with Flask so all you need to see the UI is to either just use what is already offered or build your own by following the instructions.

React offers a better response time and a data-binding system for strong architecture. We experienced React Hooks which is a promising feature for React. This accelerated our developing process.

Result

The model was trained for 20 epochs and the process lasts about 8 hours. The final loss is 0.2637 and the final accuracy is 0.056. An example of conversation is shown below:

References

[1] https://jalammar.github.io/illustrated-transformer/

[2] Attention Is All You Need, Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin, Computation and Language (cs.CL); Machine Learning (cs.LG)

[3] A Transformer Chatbot Tutorial with TensorFlow 2.0, *Bryan M. Li, FOR.ai*

[4] https://www.oracle.com/solutions/chatbots/what-is-a-chatbot/

[5] Mikolov, Tomas; et al. (2013). "Efficient Estimation of Word Representations in Vector Space". arXiv:1301.3781(cs.CL).

Assignment

Fengzhencheng Zeng was in charge of model adaptation, training, and testing. Xiangran Zhao was in charge of data processing and model adaptation. Limian Guo was in charge of testing and WEB development. We all spent lots of time researching on how to implement a chatbot and performed a thorough code review on others' code.