on Chatbots

Alvin Li, Charmaine Lam

April 2020

1 Preface

Thanks for reading this! This project took about a couple of weeks during COVID-19 to finish. This ultimate goal of this project is for us to learn. Not all parts of the document would be on the analysis of the methods used. In fact, in certain parts of the document, it would just seem like I am explaining how things work, which is just us digesting and learning! I hope you don't mind. We will be revisiting this project again in a couple of months to add new models as we learn more!

2 Introduction

Chatbots are becoming more and more prevalent in modern-day society. Through the use of online services such as DialogFlow, one can create and use chatbots quite easily by using APIs (a version of this was created as a Chrome extension here). To challenge ourselves further, we decided to create a general chatbot mainly using machine learning in Python. While the aforementioned chatbots are intent-based, our goal is only to create a chatbot that responds to general questions/ statements. It should have an open domain with no memory (each input is independent of each other). Also, the bot would be generative, which means that it should not be retrieving pre-set answers from a database.

In this project, we will be using various models for seq2seq in Tensorflow 2.0.

Sidenote: The original goal was to create a therapy chatbot but those specific datasets are difficult to find.

3 Dataset

The dataset we used was found on Kaggle, collected from the r/CasualConversation subreddit through pushshift.io. It contains 56296 entries, with conversations of length 3. Only the first two parts of the conversations were used.

A secondary dataset was created to test the models to ensure the models are setup correctly to prevent the waste of time. It contains two columns: The first is a combination of a number between 1 and 20 (ex. 12), a mathematical operation (ex. subtract), and

another number between 1 and 20 (ex. 5); The second is a number denoting the result of the previous column (ex. 7).

4 Data Preprocessing

The following techniques were used to 'clean' the data:

1. Removal of Non-Alphabetical characters
2. Lowercase

   To reduce the vocabulary size, the entire dataset was set to lower case.
3. Autocorrect

   Over 10000 words only appeared once or twice in the entire dataset, most of which are incorrectly spelled words. The autocorrect package was used on all sentences in the dataset. We noted that some names were over-corrected, like Samsung became Tamlung. The caught instances were re-corrected.
4. Remove Rarely-used Words

   Around half of the remaining total words that were still only used once or twice in the dataset, which caused some minor dimensionality issues when converting the dataset into one-hot encoding. These words are thus removed.
5. Reduction of Sentence Length

   The longest sentence length in the dataset was 60 words. However, once the sentence length was plotted with their frequency (Figure 1), it can be seen that most sentences lies between 1 and 25 words. Thus, sentences with longer than 25 words are cut short.
6. Empty and Short sentences (n<2) are deleted

   Many of the remaining pairs had entries that contained only one word (or no words). Those were removed.
7. Integer Encoding

   Each unique word was turned into an integer
8. Padding

   All sentences regardless of length are padded up to the length of 27 (due to two extra tokens, start and end) by filling the rest (padding type: post) with zeros.
9. One-Hot Encoding

   One-hot encoded numpy arrays were of shape: Total Dataset Length, Sentence Length, Vocab Size. They were first declared as zeros, then ones were put at positions according to their respective integer encoded sequences.

The final cleaned dataset has an input vocabulary size of 7000 and an output vocabulary size of 6000. Note that two extra words (ie. `\t` and `\n`) were added to each as the start and end tokens.

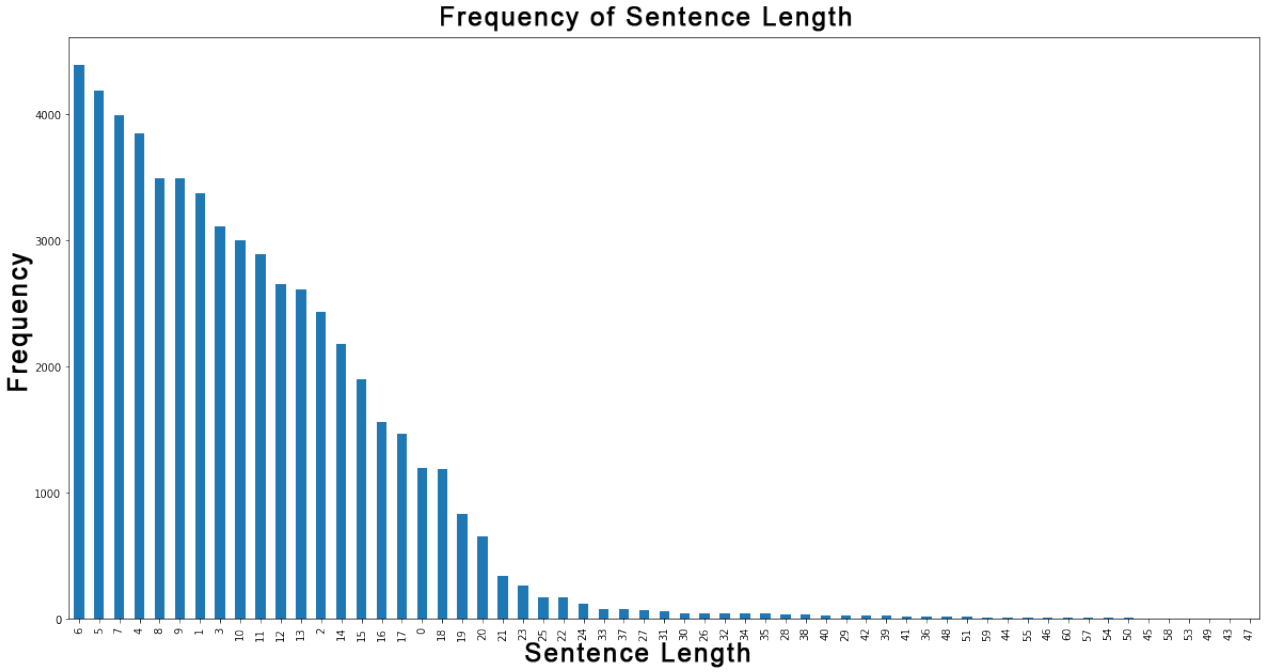

Figure 1: Frequency of Sentence Length

5 Models Overview

Almost all models used were seq2seq models, meaning it takes a sequence of inputs and outputs another sequence, which is perfect for chatbot tasks. In a chatbot, the input is the user's input, and the output is the chatbot's response. A Seq2Seq model consists of a decoder and an encoder. The encoder encodes the sequence by putting the input through a model composed mainly of LSTM or GRU layers. The encoded sequence is returned in the form of a hidden state vector. The decoder then uses the hidden state vector, along with another model composed mainly of LSTM or GRU layers to predict the output sequence.

What is a Long Short Term Memory layer (LSTM)?
Traditionally, Recurrent Neural Networks (RNN) such as LSTMs are used for sequential data (for example, stock prediction, text processing) due to their ability to back-propagate through time. However, RNNs suffer from the problem of exploding/ vanishing gradients as weights become massive or tiny when they were updated across the neurons via the chain rule [1], making them only suitable for 10 timesteps or less [2]. LSTMs partially resolve the problems by using cell gates [3]. It includes a memory gate, a forget gate, an input gate and an output state [4]. Every LSTM cell contains three types of inputs: the previous cell state c_{t-1}, the previous hidden state h_{t-1} and the current input x_t, as shown in Figure 2.
The input gate is the product of the sigmoid function of the hidden state and input ('squashed' between 0 and 1) and the tanh function of the hidden state and input ('squashed' between -1 and 1). Please note that W_x denotes the weights of each gate x, the states and

inputs are all multiplied by their respective weights prior to activation.

$$f_i = tanh(W_c \cdot [h_{t-1}, x_t]) \otimes \sigma(W_i \cdot [h_{t-1}, x_t])$$

The forget gate is the sigmoid function of the hidden state and input. It controls what gets forgetten.

$$f_f = \sigma(W_f \cdot [h_{t-1}, x_t])$$

The output gate is the sigmoid function of the hidden state and input

$$f_o = \sigma(W_o \cdot [h_{t-1}, x_t])$$

The current hidden state is calculated by the multiplication of the tanh function of the current cell state (calculated below), and the output of the output function.

$$h_t = f_o \otimes tanh(c_t)$$

The current cell state is calculated by multiplying (element-wise) the output of the forget gate and the previous cell state plus the output of the input gate. The beauty of this function is that the derivative of this function is additive, which means it would not suffer from vanishing/ exploding gradients due to multiplicative operations.

$$c_t = c_{t-1} \otimes f_f \oplus f_i$$

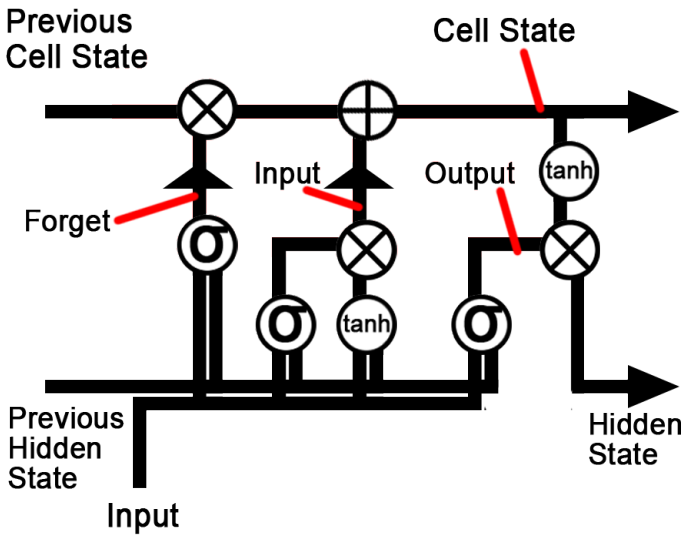

Figure 2: Representation of a LSTM cell [Li, A.]

6 Metrics

The standard form of measuring the effectiveness of the output is usually measured by BLEU scores. However, as mentioned by the paper by Liu et al., BLEU is not as effective

in measuring chatbot responses due to the diversity of the responses [6]. There is simply no ground-truth response for a certain statement/ question input. They proposed a embedding-based metric which would calculate the cosine similarity of the word embeddings, allowing for a wider variety of words [6]. They concluded that most metrics do not reflect human judgement [6]. With the consideration that our tests do not have a ground-truth or a proposed response, we decided that our metrics would be mainly based on human judgment. The metric is rather objective.
With 20 sample statements/questions, the responses will be rated as per the following, the higher the number, the better the response:

1. Completeness: Out of 2, on how complete a response is.
2. Humour: Out of 3, on how humorous a response is.
3. Grammar: Out of 4, on how much a sentence follows grammatical sense.
4. Sense: Out of 4, does it make sense?

   The metrics above are calculated and totalled per response, then averaged. The following is calculated once across the entire testing set, then added to the average of the metrics above.
5. Uniqueness: Out of 3, on whether the responses exempt uniqueness. If the responses are the same, the score would be 0.

The scores are then averaged between the co-authors.

7 Specific Models Used

1. Simple Seq2Seq

   The model is taken directly from the Keras Official Site example [5]. It uses one LSTM layer in the encoder, then one LSTM layer and a Dense layer in the decoder. The prediction is done using Greedy Search. By rearranging the model to take in hidden state, cell state and the original decoder input layer, the original Dense layer and the decoder states were outputted. For each timestep, we find the word closest to the predicted output using argmax, then feed that back into the encoder for the next timestep. This continues until the stop token is received or the max output length is reached. This method is used unless said otherwise.
   Bidirectional LSTMs: [needs fixing]
   They were all trained with 100 epochs, and the models' accuracy and loss are observed to plateau (or fluctuate) beyond 100 epochs. An increased training size is observed to lead to better results. The following variants were tested:

   (a) Base Model (RMSprop lr = 0.001, on word level, validation split = 0.05)
   (b) Base Model with RMSprop (lr = 0.01)
   (c) Base Model with Adam (lr = 0.001)
   (d) Base Model with Reversed input
   (e) Base Model with character level input and output
   (f) Base Model with Bidirectional LSTM

(g) Base Model with GRU

Note: Reversed input was tested as it was found to be more effective than the initial input [7]

2. Seq2Seq with Embeddings
Embeddings are traditionally used to improve word-based tasks. Embeddings are vectors that represent words so that related or similar words have similar vectors. The pretrained GLoVE vector was used [8]. Appoximately 90% of the vocabulary can be found within GLoVE. The rest are initialized randomly. Note the inputs are integer-encoded while the outputs are one-hot encoded. The following variants were tested:

   (a) Base Model (separate input/output embeddings, weights by GLoVE)

   (b) Base Model with separate embeddings, no initialization

3. Seq2Seq with Attention

   The model is taken from the official Tensorflow site here, which is an implementation of the attention model suggested by Bahdanau et. al. [8]. Each variant of the model was trained for 10 epochs. Subsequent changes were made to accommodate for LSTM layers as well as different attention score equations.
   Attention essentially creates a link between the input and output on a word-level. It tells the decoder which words to pay attention to. The attention layer exists between the decoder and the encoder, takes in the encoder hidden state and the decoder hidden state and calculates a score. The score depends on the type of attention we are calculating. The scores are then put through a softmax layer, and multiplied by the encoder hidden state again, which is the main reason attention works. This creates a vector that tells us which words are closely related between the input and output. The vector is then summed and used as input for the decoder.

   Bahdanau's Model [8]: The encoder is a bidirectional GRU, resulting in two hidden states, one forward and one backward. They are then concatenated. Score function is additive.

$$score = v_a^T tanh(W_1 h_t + w_2 h_s)$$

   Luong's Global Attention Model [9]: Stacked LSTMs are used, mutiplicative score function (we chose the general one). Note: The authors proposed multiple score functions.

$$score = h_t^T W_a h_s$$

   where h_t represents encoder output and h_s represents encoder output. Teacher Forcing was also used in training most of the implementations. It involves feeding the 'correct' target output as the next input rather than the predicted output.
   In the last variant, Beam Search is used for the base model to adhere with the original research paper by Bahdanau et. al. [8]. Beam Search utilizes the top n possible candidates to widen the amount of hypotheses tested. By multiplying the positive log probabilities of the steps in each hypothesis, the optimal sequence can be determined

by locating the smallest probability. Note k = 3, and max sentence length used in generation = 20 is used due to time constraints.

The following variants were tested:

(a) Base Model (GRU, Bahdanau's Score Function)

(b) Base Model with LSTM

(c) Base Model with BiGRU

(d) Base Model with 2 LSTM layers per encoder/decoder

(e) Base Model with Luong's Score and LSTM [9] (no output)

(f) Base Model with Beam Search (k=3)

4. Transformer [11]

Transformer is the only model we tried that does not involve RNNs. However, it does follow the encoder-decoder structure as in Seq2Seq models mentioned above. The inputs are first positionally encoded, then passed to the encoder. The encoder and decoder are composed of multi-head attention layers and feed-forward layers, ending with a Dense layer for outputs.

Core Concepts:

- Scaled Dot-Product Attention (SDPA):
  The input vector is multiplied to become queries, keys and values by their respective trained weights. A score is calculated by the dot product between query and key, divided by the root of dimentions of key vectors, then softmax'd. The value is multiplied by the value vector, then summed. This can be represented in the equation below, where x is the input vector, Q is query, K is key, V is value, d_k is dimension of key:

$$Attention = softmax(\frac{(W_Q(x))(W_K(x))^T}{\sqrt{d_k}})(W_V(x))$$

$$Attention = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

- Multi-Head Attention:
  This is where SDPA is used. Multi-Head represents each of the SDPA with different weights per head. For each head, SDPA is performed on the input vector, concatenated, then multiplied by another weight matrix.
  It can be represented with the equation below, where n represents the number of heads.

$$MultiHeadAttention = Concat(SDPA_1, SDPA_2...SDPA_n)W_o$$

- Positional Encoding:
  This step adds how the respective positions of each word influences the output. Thus, in the embedding layer, the words are separated based on how close they are in terms of meaning, as well as the position they are in in the sentence.

- Feed-Forward Networks

  A fully connected network with two layers and a ReLU activation in between.

- Residual Connections

  Between the feed forward and attention layers, there are residual connections. They prevent degradation problems that often occur when the network depth increases [12]. As the network gets deeper, the layers learn through the outputs of each layer, which is what is left of the input after processing (ie. residuals). By adding the original input to the processed data between the layers, residual connections work similarly to the cell states in LSTM cells to allow the flow of data from the initial layers.

8 Outtakes

Not all models result in great outcomes. Here are some of the best (worst?) performed questions and answers:

Prompt: "so i got a girlfriend today"

```
"i am not sure i am not sure i am not sure i am not sure" - 3d
"congratulations and i am so happy for you" - 3b
"that is not a productive meeting then" - 5
```

Prompt: "is the sky purple when i am not looking"

```
"oh man i have not seen it" - 3c
"you are a fucking idiot and your parents never loved you" - 5
```

Prompt: "how many days does it take to build a bear"

```
"a few weeks not sure" - 3a
"i am learning learning learning learning learning learning" -3f
```

Prompt: "what do you think about trump"

```
"it is ok to make mistakes" - 2b
```

Prompt: "i finished my homework"

```
"my sources say no" - 2b
"what is the purpose of life" - 5
```

9 Conclusion and Thoughts

None of the models are completely perfect. Many come close to answering all prompts in a correct manner.

Beam search with log multiplication seems to favour longer sentences, as for every token probability under one, the value becomes smaller, which is ultimately what beam search is striving for.

The metrics for the chatbot is undoubtedly flawed, and would not be able to accurately represent the actual performance of the models. Thus, in the end we did not calculate the score for each of the models.

10 Other Models To be Considered

1. GAN

   The GAN (Generative Adversarial Network) is often used for generating images. The classic GAN is a structure of Generator vs Discriminator, where the generator keeps trying to generate better images, and the discriminator keeps getting better at differentiating between real and generated images. The discriminator is trained to classify real vs generated data, by feeding it both. The generator is fed random noise and trained by the discriminator. As per Goodfellow, the creator of GANs, the problem with applying GANs directly to text generation is the fact that image pixel values are continuous while text values are not [13]. Although texts can be represented as a vector, a change in those values would result in completely different words, or even no words due to a limited dictionary size [14]. This creates a problem when the gradient is propagated back into the generator model.
   Introducing SeqGAN: It solves the discrete-value problem by using Monte Carlo search, which is most famously used by AlphaGo in a game that uses discrete actions [15]. As well, the sequence only matters when the sentence is complete, which is similar to Go where only the outcome is vital to learning. SeqGAN also utilizes reinforcement techniques, like rewards that need to be maximized. They are now based on how likely it will fool the discriminator [14]. The following is one of the goals of this model, to maximize the reward function:

   $$maxJ(\theta) = E[R_T|s_0, \theta] = \Sigma_{y1 \in y} G_\theta(y_1|s_0) \cdot Q^{G_\theta}_{D_\phi}(s_0, y_1)$$

   Reward Function: The reward given a parameter θ (used in the generator) is the expected return of the reward function for a complete sentence given state s_0 and parameter θ. The parameter must be optimized to maximize the reward J. The right hand side of the equation denotes the sum of the policy (probability of the token) weighted by the value (think of it as how good the token is), for all possible tokens. The parameter is updated through:

   $$\theta \leftarrow \theta + \alpha_h \nabla_\theta J(\theta)$$

   Note the plus sign instead of the traditional minus sign in gradient descent, which is because we are performing gradient ascent!
   The action-value function:

   $$Q^{G_\theta}_{D_\phi}(a = y_T, s = Y_{1:T-1}) = D_\phi(Y_{1:T})$$

The action-value function follows the policy defined by G_θ (the probability distribution of the next token), where the action is defined by y_T, which is the next token, and the state is defined by the previous generated sequences. We can see that this equals $D_\phi(Y_{1:T})$, which is the likelihood of the sequence being real.
Let's go back to the gradient, how do we calculate the gradient of the reward function (in order to maximize it)? Well, it's:

$$\Sigma_T Q \nabla log(G)$$

which means the gradient ascent algorithm is now

$$\theta \leftarrow \theta + \alpha_h \Sigma_T Q \nabla log(G)$$

[Will be revisited, unfinished]

2. BERT

   Normally, a NLP model would have to look at a given sentence during training from only one direction (ex. from left to right) and will keep predicting words until the sentence is complete.

   However, the main idea behind the BERT (Bidirectional Encoder Representations from Transformers) model is to predict a word in a sentence based on context, or in other words, "fill in the blank". For example:

   "I went to the (blank) to buy groceries."

   BERT is an unsupervised pre-trained model, that can be fine-tuned to specific data [17]. BERT looks at a sentence from both directions (left to right and right to left) [16]. This bi-directional attribute allows BERT to gain better context and understanding of flow by looking at the words on both sides of 'blank' at the same time to make a prediction [16][17]. This concept through masking, where words in the middle of sentence are randomly 'masked', in which BERT then tries to predict them.

   An important thing to note about BERT is that it is based on the Transformer model, as stated in the name. As aforementioned, this means that recurrent neural networks are not used, but rather a series of small steps. However, BERT only uses the encoder part of the transformer. Thus, the input has three components: token embeddings, segment embeddings, and positional embeddings [16].

   As mentioned in the original article, for pre-training, inputs are handled in pairs of sentences (such as question/answer pairs). These pairs can be denoted as A and B. A 'CLS' token is added to the beginning of the input, and a 'SEP' token is used to separate sequences and also to mark the end of the input [18]. Furthermore, segment embeddings are used to distinguish words between sentence A and B [18]. Lastly, positional embeddings represent a tokens relative position in an input [16].

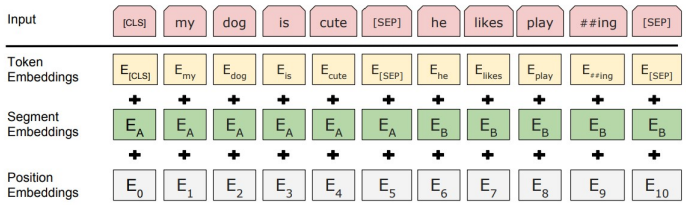

Figure 3: Representation of the BERT inputs. Source: [18]

Another task the BERT model performs during pre-training is 'next sentence prediction' to understand how the two sentences in the input are associated with each other. Thus, a label is associated with the pair, indicating whether or not the second sentence is related to the first [16]. This idea is especially useful for question and answer.

The pre-trained model of BERT can then be fine-tuned for example, by giving sentiment labels for sentence classification (eg. identifying whether a review is good or bad). One thing to consider however is that there is varying discussion around whether or not BERT is able to be used for sentence generation due to its bidirectional nature [19]. On the other hand, it is possible to start with a sentence of completely masked values, and have BERT predict the words in an arbitrary order [19].

11 References

[1] https://arxiv.org/pdf/1211.5063.pdf
[2] https://papers.nips.cc/paper/522-induction-of-multiscale-temporal-structure.pdf
[3] https://medium.com/datadriveninvestor/how-do-lstm-networks-solve-the-problem-of-vanishing-gradients-a6784971a577
[4] https://arxiv.org/pdf/1909.09586.pdf
[5] https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html
[6] https://arxiv.org/pdf/1603.08023.pdf
[7] https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf
[8] https://arxiv.org/pdf/1409.0473.pdf
[9] https://arxiv.org/pdf/1508.04025.pdf
[10] https://towardsdatascience.com/attn-illustrated-attention-5ec4ad276ee37eef
[11] https://arxiv.org/pdf/1706.03762.pdf
[12] https://arxiv.org/pdf/1512.03385.pdf
[13] https://www.reddit.com/r/MachineLearning/comments/40ldq6/generative_adversarial_networks_for_text/cyyp0nl?utm_source=shareutm_medium=web2x
[14] https://arxiv.org/pdf/1609.05473.pdf
[15] https://deepmind.com/research/case-studies/alphago-the-story-so-far
[16] https://towardsml.com/2019/09/17/bert-explained-a-complete-guide-with-theory-and-tutorial/
[17] https://moz.com/blog/what-is-bert
[18] https://arxiv.org/pdf/1810.04805.pdf
[19] https://ai.stackexchange.com/questions/9141/can-bert-be-used-for-sentence-generating-tasks