Customer Support Chat Bot for Twitter

Omar Walid Hamed Nofal

April 22, 2024

Abstract

1 Introduction

In the recent years, customer support has shifted from phone calls to a more digital service. Nowadays, users expect to find answers to their problems by chatting with a company representative on the internet. Almost all large and medium companies provide a way for users to chat with someone from customer support either on their website, or on social media platforms, like Twitter. Natural Language Processing (NLP) has proven to be well-suited for question answering tasks (QA), as several chat bots have been implemented that are able to take chat history into consideration. This project aims to implement a history-preserving chat bot that can be used by companies to respond to customer inquiries on Twitter.

2 Motivation

Since the rise of the digital age, and the shifting of customer support to websites and social media platforms, companies have been faced with challenges concerning responding to customer inquiries. Mainly, traffic has increased considerably and this puts tremendous load on customer support agents. Also, chatting with a customer doesn't normally happen in one-sitting, instead sometimes it takes hours for customers to respond to agents and vice-versa. This means that agents sometimes have to re-read the whole conversation to remember the context. Also, much of the inquires made by customers can be easily responded to and don't require a human agent to process it. Creating a chat bot which is able to respond to customer inquires would help decrease the load on customer support agents and improve customer satisfaction.

3 Literature Review

Since our problem concerns building a QA chat bot that remembers chat history, we will take a look at the previous research that focuses on either building QA systems, chat bots or both. We will also learn the brief history of neural network architectures in the context of NLP.

Question Answering (QA) systems, are software systems capable of answering questions posed by humans. They can respond to questions concerning facts, definitions, reasons and so on. Early QA systems started off as simple rule-based systems. These systems stored a database of facts or a knowledge base which contained structured information about a specific domain, and as such they were called closed-domain systems. Rule-based systems tried to formulate a database query from the user's input which had to be carefully crafted to make it easier for the system to find information. The information is queried from the database and then presented to the user in a human-readable format. The first rule-based QA systems in the 60s and 70s were BASEBALL and LUNAR. BASEBALL answered questions about Major League Baseball over a period of one year, while LUNAR answered questions about the geological properties of the lunar rocks returned by the Apollo missions. These QA systems were successful but the queries given to them had to be precise so that the model can formulate a database query from the input. By the end of the 90s and the beginning of the 2000s, QA systems started to shift from rule-based to text-based approaches. They leveraged the power of

NLP in order to understand, extract information and formulate answers from unstructured data like web pages, books, and documents. These systems also had the ability to understand unstructured user queries, so that users can be more flexible when asking their questions. One of the earliest most successful systems was IBM Watson which employed advanced NLP algorithms and machine learning techniques to win 1M US Dollars in the popular quiz show Jeopardy!. Since then, most of the effort in developing QA systems almost completely shifted from rule-based systems to machine learning and NLP statistical models. All of this evolution of QA systems happened in parallel or as a side effect of the advancements of chat bots which are now the go-to methods nowadays to implement QA systems. We will briefly go over the evolution of chat bots and how QA systems improved greatly as a side effect.

The first ever chat bot was ELIZA [7] which simulated a psychotherapist. It was very simple by today's standards but engaged users in text-based conversations and demonstrated the potential for machine to interact with humans through natural language. By the 2000s, chat bots started to benefit from the advancements of machine learning and computing power and started to use statistical models and NLP techniques to provide much more natural and realistic conversations than earlier rule-based systems. Recurrent neural networks take credit for much of the advancements in the processing human language and other sequential unstructured information like speech and videos. They are basically like feed forward neural networks but each node maintains a state which it feeds to itself as an input and updates it accordingly. RNNs, however, suffered from the vanishing gradient problem, as the length of the input increases, the model starts to 'forget' the earlier words which was not ideal for long-form text input. This motivated researchers Hochreiter and Schmidhuber to develop the Long Short-Term Memory (LSTM) variant of the RNN [2]. This variant contains much more complex units consisting of a cell, input gate, output gate and a forget gate. These gates regulate the flow of information into and out of the cell, allowing the LSTM to selectively remember or forget information over long sequences.

The latest step in NLP was the invention of Transformers which greatly influenced the development of QA systems and chat bots. Introduced by Vaswani et al. in the seminal paper "Attention is All You Need" in 2017, Transformers have demonstrated remarkable capabilities in capturing long-range dependencies in sequential data, making them particularly well-suited for tasks such as language modeling, translation, and question answering [6].

Unlike traditional recurrent neural networks (RNNs) and LSTMs, Transformers rely solely on self-attention mechanisms to weigh the importance of different input tokens when generating output representations. This attention mechanism allows Transformers to capture global dependencies in the input sequence, enabling more effective modeling of context and reducing the vanishing gradient problem often encountered in RNNs [6].

The effectiveness of Transformers has been demonstrated in various NLP tasks, including question answering, where models like BERT (Bidirectional Encoder Representations from Transformers) and T5 (Text-To-Text Transfer Transformer) have achieved state-of-the-art performance on QA benchmark datasets [1] [3]. In the context of chat bots and customer support, transformers have enabled the development of more sophisticated and context-aware conversational agents. By leveraging large-scale pre-training on vast amounts of text data, transformer-based chat bots can generate more coherent and relevant responses to user queries, enhancing the overall user experience. One of the best examples of a transformer-based chat bot is ChatGPT developed by OpenAI. It is able to mimic real human conversations and answer complex questions from many different domains. QA systems like these are called open-domain systems because they can be queried in a wide range of domains, but they can be fine-tuned to improve their accuracy in a more specialised domain like customer service.

To be able to provide a good user experience, chat bots must be able to remember the history of the conversation which gives the user the feeling that he is talking to a human. This is especially important on Twitter where conversations can span multiple tweets and DMs. Some bots like ChatGPT do that easily by feeding the whole conversation to the user in addition to the user's new query, this way, the bot is able to take the previous history into context. However, other methods exist that allow neural networks to maintain long-term memory. Weston et al. proposed an architecture for a neural network (MemNN) [8] which contains an additional memory component that acts as a dynamic knowledge base. This memory component is updated on every query and queried when generating answers for users. This architecture provided great results on the large QA dataset by Fader et al., however, it suffered a

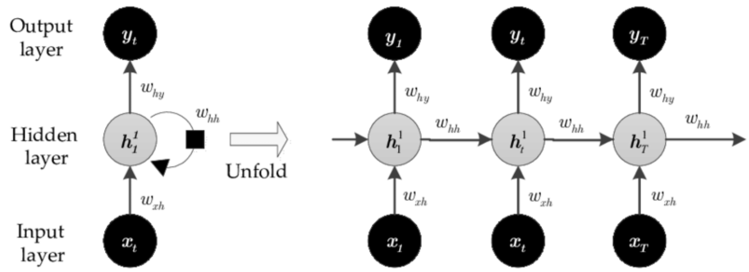


Figure 1: The standard RNN. For each time step t the output of the node at $t-1$ (H_{t-1}) is used as its input.

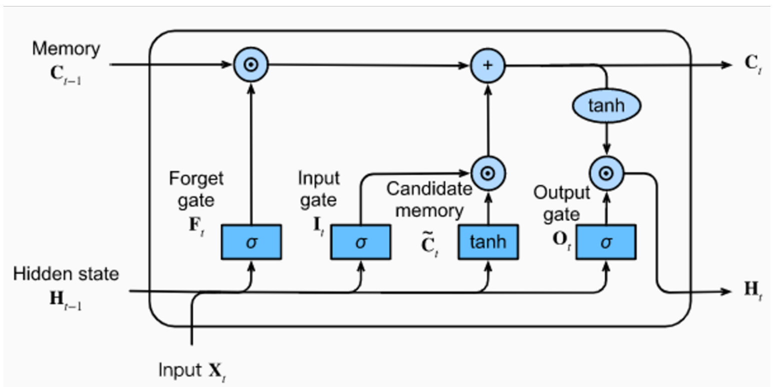


Figure 2: A single LSTM unit. The C is considered as the long term memory component while the H is the same in the case of an RNN. C is modified according to the Forget and Input gate. The H is modified from the Output gate and both C and H are reused at later time steps.

major disadvantage in that it required too much supervision, as the memory layer requires supporting facts to be trained on, but most QA datasets only contain question and answer pairs. Sukhbaatar et al. modified the previous architecture to be trained end-to-end without requiring any supervision on the memory layer, this was possible without sacrificing the performance by much [4].

To conclude, this literature review highlights the significant advancements and challenges in the development of QA systems and chat bots, driven by advancements in NLP and neural network architectures. From the early rule-based systems to modern transformer-based models, the field has seen remarkable progress in enabling machines to understand and respond to human language effectively. Moving forward, continued research and innovation in NLP and machine learning promise to further enhance the capabilities of chat bots, enabling them to provide even more context-aware and personalized interactions in various domains, especially the QA systems domain.

4 Data Analysis

In this section, we will extract some insights from our dataset that will be useful when it comes to training the model. Gaining insight of the data is crucial to be able to understand why the model is working or not. We will focus mainly on statistics useful for the training phase to detect the needed pre-processing that should be applied to our dataset.

The dataset we will be using is the Twitter customer support tweets dataset on Kaggle. It contains 2811774 tweets between customers and famous brands' support agents. Each tweet row contains the sender id, the id of the tweet it is replying to, and the id of a tweet that replies to it (if it exists). In figure 3, we present the top brands by tweet count.

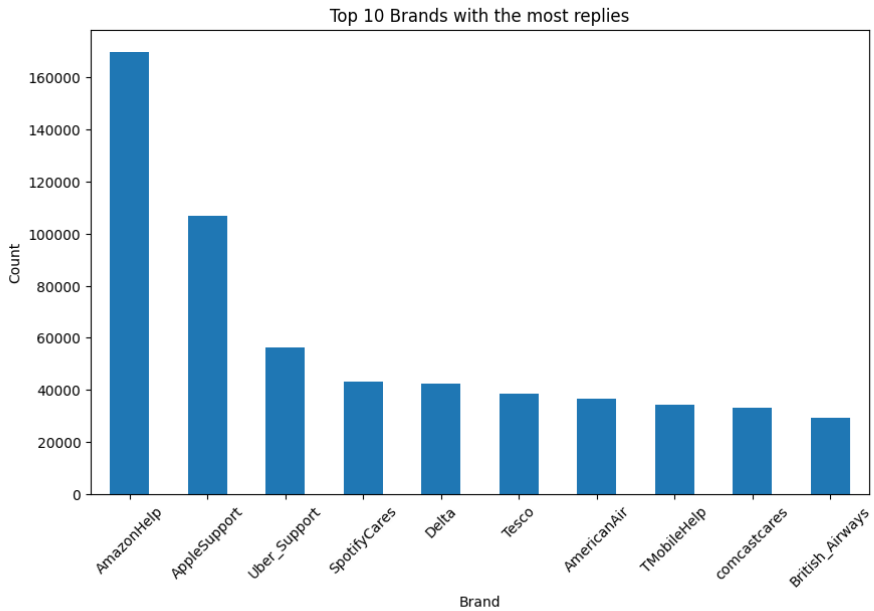


Figure 3: Top brands by tweet replies

4.1 Simple Tweet Analysis

There are 2811774 tweets, the average tweet length is 113 words. The average word length inside a tweet is 5.6 letters excluding stop-words. Almost all tweets are in English except for a few tweets in German and Chinese which will be excluded from training. Overall, the dataset structure is clean and makes it easy to find the replies to a particular tweet which will be useful when generating the training and test datasets.

4.2 Word Frequency Analysis

In figure 4, we can see a bar chart of the top common words after removing stop words. We can see the most famous word is `https`. After investigation, it turns out because a lot of the tweets sent by the companies contained links to other tweets. These links will be definitely removed before the training phase. Other famous words are also `dm` and `us`. Again, a lot of brands would just ask the customer to send them a direct message to assist them in private. These type of tweets would prove difficult to our model and would be unwanted noise and should be removed before training. Figure 5 shows a word cloud of the most common words in our dataset.

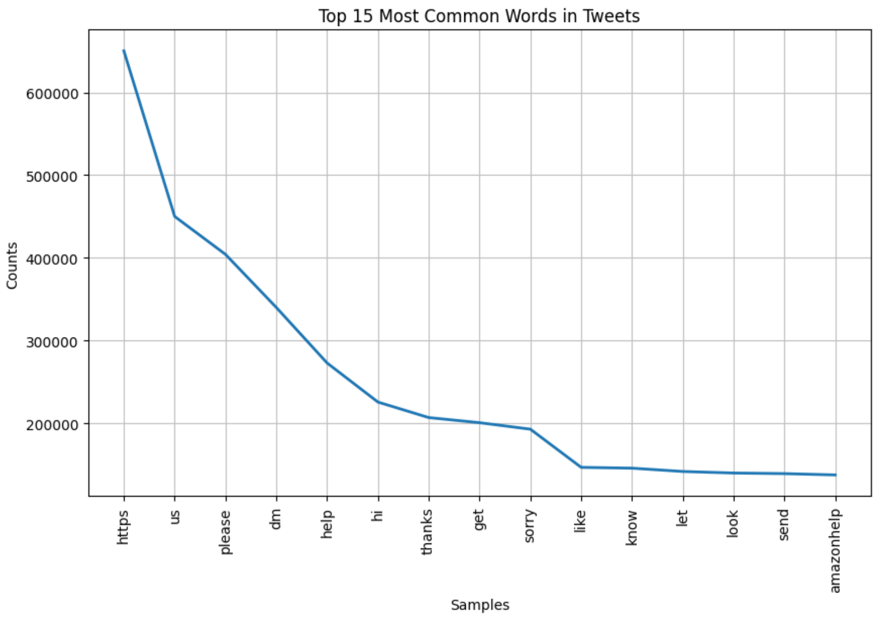


Figure 4: Most common words in the dataset

Figure 5: Word cloud of the most common words

4.3 Bi-gram Analysis

It is sometimes helpful to find the most famous bi-grams in our dataset to detect collocations and to help resolve ambiguities. Figure 6 shows the most famous bi-grams in the dataset. Again the top bi-gram is `(https, co)` which, as explained earlier, is due to the large amounts of twitter URLs sent by brands which will need to be removed. Also, we can see the famous `(dm, us)` bi-gram which occurred in more than 100,000 tweets.

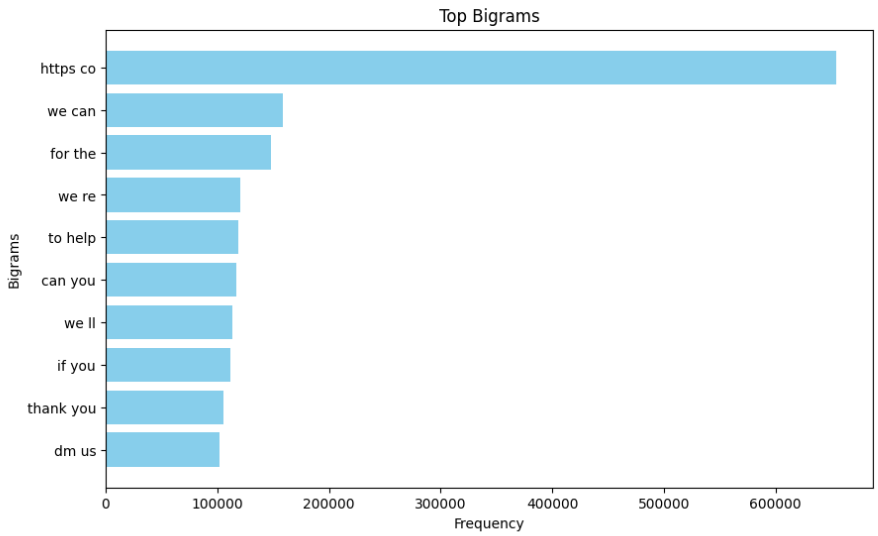


Figure 6: Top bi-grams in the dataset

4.4 Sentiment Analysis

We also applied NLTK's Vader sentiment analysis on the tweets, to try and get an idea of the overall tone of the tweets to see if there are any trends in customer satisfaction but the results were not that interesting. Almost all tweets are neutral with some positive and negative tweets, as evident in figure 7.

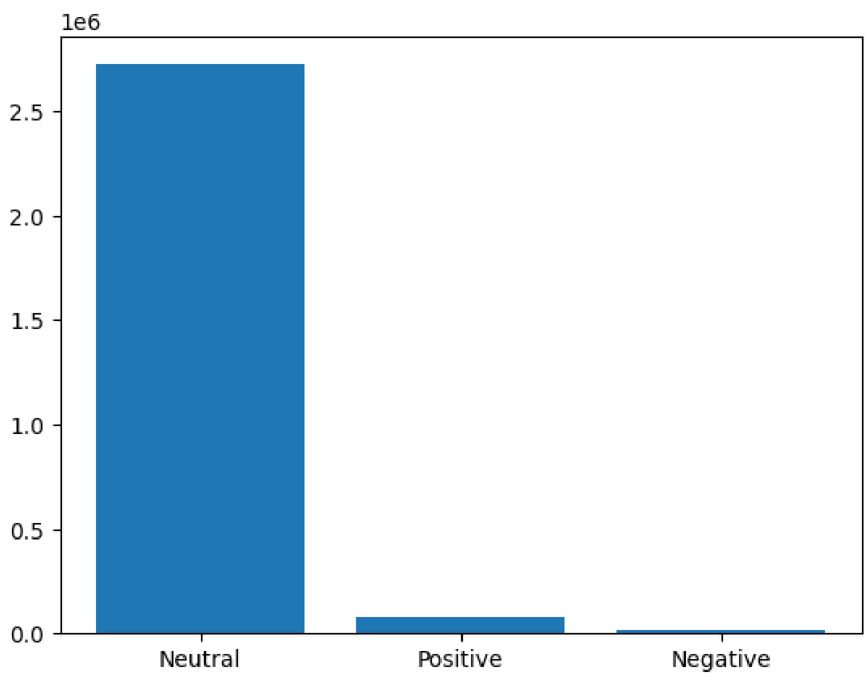


Figure 7: Sentiment Analysis of the tweets

4.5 Limitations

Overall, the dataset is quite dense, organized and clean, however, it suffers from a few limitations.

First, the tweets require a lot of cleaning before being fed into the model for training. In particular, things like other languages, emojis, URLs, id's of the senders, tags, emails, and others, will have to be removed from the dataset before training.

Second, hundreds of thousands of the responses from the brands are of the type `"Please send us a DM in order to assist you"`. These types of replies are not useful at all and provide a lot of noise for the model, and they will have to be removed, which will decrease the tweet count by a lot, since we will also have to remove the original tweet containing the question asked by the customer.

Finally, the dataset is quite large, and because the model will be created on a personal computer, this might slow down the process considerably. Therefore, depending on the amount of data left after cleaning, some of the data will have to be removed to make the training more computationally feasible

5 Methodology

This section discusses the steps taken to implement a model that is able to answer users' queries trained on the dataset mentioned in the previous section. We will discuss

1. Data Cleaning
2. Model Architecture
3. Word Embedding
4. Training method
5. Training results and discussion

5.1 Data Cleaning

The dataset came in a table format containing tweets and each tweet contained info about which tweet it was responding to. But this format is not suitable to be processed directly, so the dataset was converted to a list of conversations. Each conversation contains alternating messages between a client and a support agent of a company. The dataset contained 787346 conversation in total, with an average of 2.96 messages for each conversation. Each message contained an average of 133.9 characters.

The number of conversations is too large to be trained on a personal computer, therefore only conversations with the brands `Amazon, Apple, Uber and Spotify` were used and half of those conversations were dropped. This reduced the number of conversations to 115611, with a total of 360835 messages.

The conversations were then fed into a cleaning process. The process involved removing links, mentions, emails, emojis and non-ASCII languages. Also, all messages were lower-cased to reduce the number of distinct tokens. Stop words and punctuation symbols (!, ?, etc..) were not dropped to give the model the ability to generate well-formed sentences.

To summarize data cleaning, in total, there are 115611 conversation with an average of 3 messages per conversation. Each conversation has lower-case ASCII characters containing stop words, and special characters.

5.2 Model Architecture

For the architecture of our model, a sequence-to-sequence (Seq2Seq) architecture was used, this type of architecture is made specifically for tasks that require mapping a sequence to another sequence, like machine translation, text summarizing or in our case, a QA system. [5]

A sequence to sequence model contains two parts. An encoder and a decoder. The encoder takes the input and converts it into a vector called the context vector or the thought vector. This vector ideally should represent and capture the details of the input. The context vector is then fed into the second part, the decoder, which starts generating a response based on the query contained in the context vector. Figure 8 shows a diagram of the process.

For our encoder, it was implemented using a 2-layer stacked LSTM [2] with a hidden layer size of 30 each. After feeding the whole input, the last layer's hidden state is taken as the context vector to be fed into the decoder.

For the decoder, it was also implemented as a 2-layer stacked LSTM with the same number of hidden units for each layer. A linear layer was added to perform the classification task based on the

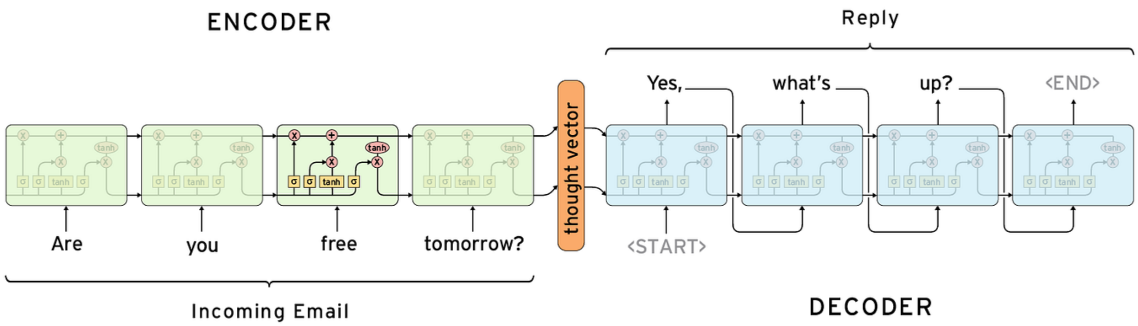


Figure 8: General structure of a Seq2Seq model. The encoder takes the input and generates a thought vector. The decoder takes the thought vector and generates the resulting sequence.

output of the LSTM layer. The linear layer has a size equal to the vocabulary size of our dataset. And so, the model generates a probability distribution of all possible tokens that the model can generate. When the encoder outputs the context vector, it is then used as the starting initial state of the decoder ($h_{decoder}(t = 0) = ContextVector$). The decoder then starts generating a sequence until a special token is generated or a maximum sequence length is reached.

5.3 Word Embedding

A machine learning model can't understand text, so we will have to generate numerical representations of each possible word in our dataset (token). These numerical representations are called embedding vectors. Each possible token in our dataset gets assigned an embedding vector of 100 dimensions. One of the benefits of embedding vectors is that, words that have similar meaning can have very similar vector representations so that the model can understand that both words can be used in a similar context.

The embeddings of all tokens in our dataset were generated using Word2Vec with the skip-gram algorithm and a context window of 5 words and for 10 epochs. A total of 21716 tokens exists in our dataset, excluding rare tokens (<5 occurrences). The output of this step was an embeddings table that contains a row for each token along with its embedding vector. Before feeding sequences into the encoder, we will have to pass the token to the embeddings table to get the vector associated with that word. This vector is then used as the input to the encoder and the model continues as described in the last section.

Five special tokens were also added to the embeddings table. Mainly `<BOC>, <EOC>, <BOA>, <EOA>`, and `<PAD>`, which stand for, beginning of client question, end of client question, beginning of agent answer, end of agent answer, and padding respectively. The first four tokens are responsible to mark the beginning and end of questions and answers before giving it to the encoder. This should give it the ability to understand which sentence is a question and which is an answer, the `<EOA>` token is also the token which signals the end of the answer generated by the decoder. The padding token is used to fill input to the model if it is below a specified sequence length (more on that later).

5.4 Training Method

Up until now, we have a clean dataset, a model and word embeddings. To train the model, we need pairs of questions and answers. However, we also want the chat bot to be able to remember the conversation context. Therefore, ideally, we should give the model pairs of contexts and answers, where a context is basically the last few messages before and including the new question. Unfortunately however, generating all possible combinations of contexts and answers was impossible to achieve given our machine's computational capabilities. Therefore, for the training, we only used pairs, where each pair contains a single question and its answer. This will seriously impact the performance of the model and its ability to recall context as we will see in the results.

Before giving our input pair to the model, the sentences are tokenized, and then for each token we use its embedding vector in the embeddings table. If the length of the tokens are less than a pre-defined sequence length (50), then we will fill up the remaining tokens with the `<PAD>` token. The embedding vector of the `<PAD>` token is all zeros, as to not affect the hidden state of the model when it passes through it. The sequence length 50 was used, because in our dataset the average message length was 25 tokens. Therefore a sequence length of 50 should enable it to take 2 messages into consideration without taking up too much computational resources.

For the training loop, we give the encoder a single question to get the context vector associated with that question. This context vector is used as the initial hidden state of the decoder as discussed in the model architecture section. At the decoder, we give it the whole answer sequence (teacher forcing), and for each token in the answer, it generates a probability distribution containing probabilities of each possible token becoming the next token.

After we get all the predictions made by the model for each token in the answer, we calculate the loss between the predicted outputs and the target outputs determined by our answer. We use the cross-entropy loss function to get the loss, and then we calculate the perplexity using $perplexity = e^{loss}$. We use the perplexity to gauge the model's improvement over the whole training cycle.

For training, we used the Adam's optimizer with 0.001 learning rate and a batch size of 128, and we trained for 15 epochs, while recording the loss and the perplexity for visualizing the results. We used 80% of our dataset as the training set and reserved the remaining 20% to be used as the testing dataset. The model was tested on the test dataset after each epoch to monitor its progression without updating the weights after testing.

5.5 Training Results and Discussion

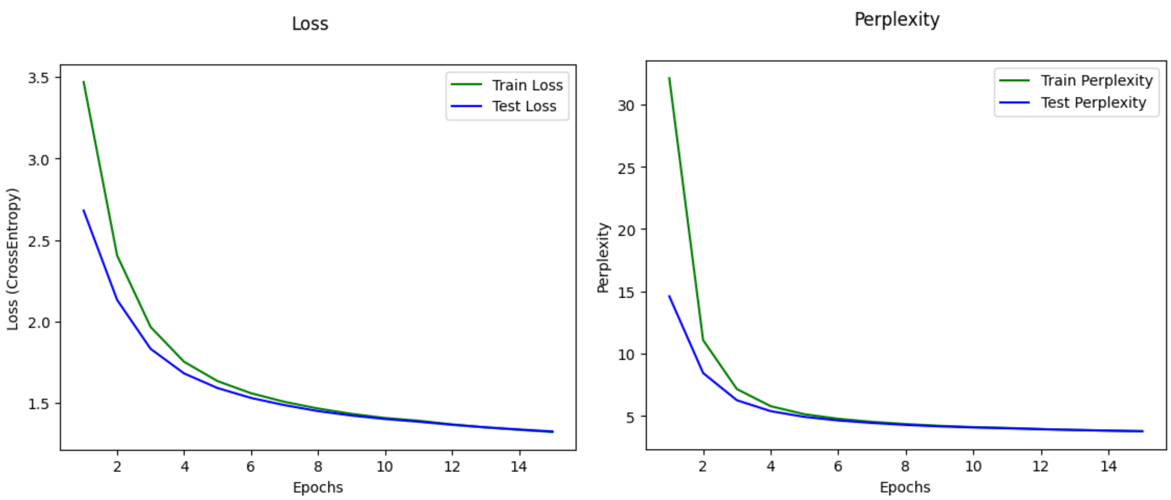
During training, the loss and perplexity metrics decreased quickly during the first few epochs. Figure 9 shows the graphs of the loss and perplexity over the training and testing datasets. We can see an overall nice decrease in the beginning which later slowed down, which indicates that the model is converging. Also, the testing loss decreased as well with the same rate and reached an almost equivalent loss as the training dataset, indicating that the model was able to generalize without overfitting on the training dataset.

However, when it came to testing the chat bot using real-world conversations, the performance turned out to be terrible. Two sample conversations have been provided in listing 1 for reference.

There are a few reasons that could explain why the performance is so terrible. First of all, because of our computational constraints, we had to do some sacrifices like truncating the dataset, decreasing the sequence length, decreasing the context length, and using a smaller model. Some of the sacrificed stuff was essential to improve the model's ability to retain context. Second of all, the cleaning process was not the best as we can see some gibberish in the sample conversations that the model picked up on like clients' names, agents' name, and we even see the bot trying to provide links but it failed because we deleted the links before. Finally, the dataset itself was not of a very high quality due to many responses not containing anything useful except links or asking the client to talk to them in private.

Although the overall model performance is not to be celebrated, there are still some nice properties which we can notice. First, the model is able (sometimes) to output nice coherent sentences like `we 'd love to help` , `we 'll reach out to you` , `sorry for the delay !` , `contact us for help`, etc.. . The reason why these sentences are good is because they are the most frequent sentences sent by the agents in our dataset, which means the model was able to memorize them. This could tell us that the model is performing well and able to pick up patterns but maybe the dataset is of low quality that the model wasn't able to perform any useful task.

To summarize, the model's training exhibited promising signs initially with decreasing loss and perplexity metrics, suggesting convergence without overfitting. However, when tested on real-world conversations, the performance was disappointing due to a few sacrifices made before training and low quality of the dataset.



(a) Loss progression

(b) Perplexity progression

Figure 9: Loss and perplexity progression as the training progresses over epochs

Listing 1: Sample testing conversations

```
Conversation #1
---------------
Client> hello there please help me
Agent> if you 've sent us the direct message , the email linked , had following a shot
    addresses , and we 'll update your device - 11.1.1 . and keep us posted if the one arrive
     ,
Client> i have a problem with my iphone
Agent> we 'd be happy to assist with you . send us dm the ios version and we 'll go from
    there . that does n't sound right here :
Client> my ios is 11.1 and the battery started to drain quickly
Agent> we try to the family steps . we 'll suggest there us here : /mu
Client> no don't do that
Agent> here to help . send us a note at so our team can connect .
Client> i already sent you but no one responded
Agent> we 'd love to help . please let us know out with your device . 2 : how here for be
    right . we are there on your order info . when we 'd love to provide that feature here
    here : \^pk
Client> bad agent
Agent> when you have never exchange level of your update and our best team . kindly fill
    forward the with available options . it will receive from our support . \^hd


Conversation #2
---------------
Client> hello there can you help me with my package
Agent> i 'm afraid we ca n't cause it wo n't help you soon now if the item , do you receive a
     demand delivery as delivery date : \^td
Client> i didn't even tell you the problem yet
Agent> kindly check with this at the account seems from the additional form ( 1/2 ) \^cr
Client> hmm, you are speaking nonsense
Agent> we 'll reach out to you with them ! if
Client> who are 'them'
Agent> upload destinations balance \& feedback ) , please reach us via twitter in english .
    contact us for help in with review these review :
Client> the package is marked as deliverd but I didn't receive it
Agent> sorry for the delay ! we 'd like to provide your information via ; email ! \^dd
Client> good agent
```

```
Agent> we 're so sorry for it ! send us a the bottom so we can further assist .
```

References

[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[2] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[3] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2023.

[4] S. Sukhbaatar, A. Szlam, J. Weston, and R. Fergus. End-to-end memory networks, 2015.

[5] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks, 2014.

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need, 2023.

[7] J. Weizenbaum. Eliza—a computer program for the study of natural language communication between man and machine. *Commun. ACM*, 9(1):36–45, jan 1966.

[8] J. Weston, S. Chopra, and A. Bordes. Memory networks, 2015.