Tribhuvan University

Institute of Science and Technology

Kirtipur, Kathmandu

A PROJECT REPORT

ON

Chatbot Using Sequence to Sequence Model

In Partial Fulfillment of the Requirement for a Bachelor's Degree In

Computer Science and Information Technology

Submitted by

Dinesh Kumar Thapa (20235/075)

Sijan Adhikari (20278/075)

Siraj Bhattarai (20279/075)

Under the Supervision of

Asst. Prof. Surya Bam

Institute of Science and Technology

Bhaktapur Multiple Campus

Dudhpati, Bhaktapur

SUPERVISOR'S RECOMMENDATION

I hereby recommend that this report has been prepared under my supervision by **Dinesh Kumar Thapa (20235/075), Sijan Adhikari (20278/075)Siraj Bhattarai (20279/075)** entitled **"Chatbot Using Sequence to Sequence Model"** in partial fulfillment of this requirement for the degree of B.Sc. In Computer Science and Information Technology (B. Sc. CSIT) be processed for evaluation.

.....

Asst.Prof Surya Bam

Supervisor

Department of Computer Science and Information Technology, Bhaktapur Multiple Campus, Dudhpati, Bhaktapur

LETTER OF APPROVAL

This is to certify that this project **Dinesh Kumar Thapa (20235/075) Sijan Adhikari (20278/075) Siraj Bhattarai (20279/075)** entitled **"Chatbot Using Sequence to Sequence Model"** in partial fulfillment of the requirement for the degree of Bachelor of Science in Computer Science and Information Technology (B.Sc. CSIT) has been well studied. In our opinion, it is satisfactory in the scope and quality for the required degree.

Supervisor: **Asst.Prof. Surya Bam**,
Department of Computer Science and Information Technology,
Bhaktapur Multiple Campus

HOD: **Mr.Sushant Poudel**
Bhaktapur Multiple Campus

Internal Examiner: **Bhaktapur Multiple Campus**

ACKNOWLEDGEMENT

The successful completion of this project work would not have been possible without the support and assistance of many individuals. We feel immensely blessed to have gotten this during our study. We want to take this opportunity to offer earnest admiration to everyone of them.

First and foremost, we would like to acknowledge our mentor and our project supervisor, **Asst. Prof Surya Bam**, whose guidance helped us with different views of development and research in the project. His guidance has been a cornerstone in developing the Citizenship Automation module in our project. We were able to accomplish this project with the help of his helpful directions and suggestions.

We want to express our special thanks to the Department of Computer Science and Information Technology for providing us with an environment to explore the latest technology, investigate it, and research those areas as a major project. We are grateful to **Mr. Sushant Poudel,** coordinator of the Department of Computer Science and Information Technology, for presenting valuable suggestions regarding the queries put forward regarding the project. Last but not least, we would appreciate all our teachers, seniors, and friends for their support, irrespective of the situation, and constant inspiration, which helped us achieve our dreams and make our project come to an end.

ABSTRACT

Chatbots are made up of special software that can comprehend natural language and have a conversation just like a human would. Chatbots are replacing conventional human assistance agents everywhere. Instead of just answering questions, this project aims to develop a model that can make a conversation just like a human would. This project uses this Generative model and is able to keep track of what was said in previous messages as well, thereby improving the responsiveness of the answer and its quality. The generative chatbot is built with sequence-to-sequence models using Recurrent Neural Networks (RNN). Seq2seq turns one sequence into another sequence. The context for each item is the output from the previous step. The primary components are one encoder and one decoder network. The encoder turns each item into a corresponding hidden vector containing the item and its context. The decoder reverses the process, turning the vector into an output item, using the previous output as the input context. Attention mechanism can be used with the RNN, allowing it to focus on certain parts of the input sequence when predicting a certain part of the output sequence, enabling easier learning and higher-quality output.

Table of Contents

LIST OF FIGURES

ABBREVIATIONS

AI	Artificial Intelligence
GRU	Gated Recurrent Unit
NLP	Natural Language Processing
RNN	Recurrent Neural Network
Seq2Seq	Sequence to Sequence

Chapter One: Introduction

1.1. Introduction

Modern society is not imaginable without personal virtual assistants and conversational chatbots, such as e.g. Siri, Alexa, Google Assistant, Cortana, and ChatGPT; they are completely changing our communication habits and interactions with technology. Intelligent chatbots are fast and can substitute for some human functions; however, artificial intelligence (AI) and natural language processing (NLP) technologies are still limited in replacing humans completely.

Our Simple generative chatbot is made using the seq2seq model with a maximum reply length of 10 letters. The model is built using GRU (Gated Recurrent Units) and trained on the Cornell Movie Dialogs Corpus, a popular dataset used for natural language processing tasks. Our chatbot uses an encoder-decoder architecture to generate responses based on the input it receives. With a limited reply length of 10 letters, the chatbot is designed to provide quick and snappy responses, making it an ideal tool for engaging in short conversations.

In this Project, we explore seq2seq-based deep learning (using RNNs) architecture solutions by training generative chatbots on small dataset. The generative chatbot creation task for the final project is more challenging due to some of the many following conditions: computing power, trust deficit, bias problem and so on.

1.2. Problem Statement

The aim of this project is to develop a generative chatbot using sequence-to-sequence (seq2seq) models. The chatbot will be designed to understand natural language inputs from the user and generate as much relevant response as possible in natural language.

The challenge with building a generative chatbot is to ensure that it produces responses that are not only grammatically correct but also contextually appropriate. Seq2seq models are effective in generating coherent responses in natural language, but they require a large amount of training data and careful tuning of hyperparameters to achieve optimal performance. The chatbot will be evaluated on its ability to generate responses that are relevant and appropriate to the given context, as well as its ability to handle a wide range of inputs and respond promptly. The success of this project will be measured by the performance of the chatbot in terms of its accuracy, fluency, and coherence, as well as its ability to engage in a meaningful conversation with the user.

Overall, the development of a contextually-aware generative chatbot using seq2seq models has the potential to revolutionize the way we interact with technology, and this project.

1.3. Objectives

The main objective of this project is to build a chatbot that can take user input and give a reply accordingly. However, the thorough set of objectives can be listed below:

- To design and develop a chatbot that can handle a limited set of predefined user inputs and provide a fixed set of responses.
- To provide suitable responses to user requests.

1.4. Scope and Limitations

The underlying goal of building any chatbot is to support and scale the work of people in an organization. Some of the scopes are:

- Chatbots can provide personalized recommendations based on user behavior and preferences.
- Chatbots can be used to handle customer service inquiries and support.
- Chatbots can be used in healthcare to provide medical advice and support.
- Chatbots can be used in the automotive industry to provide vehicle-related support and assistance.

Some of the limitations are:

- Accuracy and relevancy: as the accuracy of output is related to training, computer power, and data bias affect the accuracy and relevancy of output from the chatbot
- Limited Range of User Inputs: A simple chatbot with less accuracy may not be able to handle a wide range of user inputs or understand complex queries, which can limit its usefulness and effectiveness.
- Fixed Set of Responses: A simple chatbot with less accuracy may only be able to provide a fixed set of responses to users, which can lead to repetitive and unhelpful interactions.
- Limited Ability to Learn and Adapt: A simple chatbot with less accuracy may not have the ability to learn and adapt to new situations or user feedback, which can limit its ability to improve over time.
- Inaccurate Responses: A simple chatbot with less accuracy may generate inaccurate or irrelevant responses, which can lead to frustration and a negative user experience.

1.5 Development Methodology

To begin with, the collection of back-and-forth conversational datasets is an essential first step in the process of building a generative chatbot to get human like response. The dataset must contain the data in form of a question-answer pair which will help system to learn on basis of question and answer, to ensure that the system can handle different scenarios and produce relevant response.

Once the dataset has been collected, it is important to clean up the conversation and normalize string by changing Unicode to ascii, removing punctuation and whitespaces etc. and build a vocabulary that acts as dictionary to store all question-answer pair in number form trimming question-answer pair to maximum length of 10 and removing rare word less than minimum count of 3 is also done.

Although we have put a great deal of effort into preparing and massaging our data into a nice vocabulary object and list of sentence pairs, our models will ultimately expect numerical torch tensors as inputs. So we convert words into indexes and pad our sentences to get a batch of data to train the model with.

After preparing the input for the model, the next step is to build a seq2seq model, encoder model takes input and gives output and its hidden state using a bidirectional GRU. attention model takes the encoder output and hidden state to return SoftMax normalized probability score. decoder model takes encoder outputs and attention weight and gives output and hidden state.

After preparing model, next step is to train model for single iteration by setting up device to train on cpu or gpu, using gradient clipping and teacher forcing to calculate loss and loss is backpropagated and model weight is adjusted. After training for single iteration we train for multiple iteration by passing our models, optimizers, data etc. and save the trained model as checkpoints in tar files.

After training a model, we want to be able to talk to the bot ourselves. First, we must define how we want the model to decode the encoded input. Greedy decoding is used when we are not using teacher forcing to decode our input. Evaluate function manages the low-level process of handling the input sentence, and the evaluateInput function acts as our interface to chatbot.

Finally next step is to run the model with preferred initialized parameters and run an evaluation to chat with the chatbot.

1.6. Report Organization

Chapter One is about the introduction of the project along with the problem statement, objectives, scope, and limitations. **Chapter Two** specifies the background study for the project with a literature review of the existing systems. **Chapter Three** comprises system analysis, which includes requirement analysis and feasibility analysis. **Chapter Four** specifies the system design, algorithms that have been used during the development of this application. **Chapter Five** includes the implementation and testing of the system. In the section on implementation, the tools used during this system development have been included. Similarly, algorithm procedures have been mentioned. The results with the relevant screenshots and code snippet have been included. In **Chapter Six**, the conclusion and future recommendations have been mentioned. Hence, the report consists of six chapters where all the details of the system have been explained.

Chapter Two: Background Study and Literature Review

2.1. Background Study

In the context of chatbots, NLP techniques are used to enable the chatbot to understand the user's intent and generate relevant responses. Sequence to sequence models is a type of NLP technique that is particularly useful for generating conversational responses. These models use a combination of encoder and decoder neural networks to convert an input sequence of text into an output sequence.

The first chatbot ELIZA [1]invented in 1966 was based on keyword recognition in the text and acted more as a psychotherapist. ELIZA did not answer questions; it asked questions instead, and these questions were based on the keywords in human responses.

*"*Parry" chatbot was constructed by American psychiatrist Kenneth Colby in 1972. The program imitated a patient with schizophrenia. It attempts to simulate the disease. It is a natural language program that resembles the thinking of an individual. "PARRY" works via a complicated system of assumptions, attributions, and "emotional responses" triggered by changing weights assigned to verbal inputs.

The "smarterchild" was in many ways a precursor of Siri and was developed in 2001. The chatbot was available on AOL IM and MSN Messenger with the strength to carry out fun conversations with quick data access to other services.

"ChatGPT" is a large language model trained by OpenAI. It was founded by the OpenAI team in 2021. It is designed to assist users in generating human-like text based on given input. ChatGPT can be used for a variety of tasks, including conversation generation and language translation. The model is trained on a massive amount of data, allowing it to generate text that is often difficult to distinguish from text written by a human. ChatGPT has been praised for its ability to generate natural-sounding text and its potential applications in a variety of fields.

Our project also focuses on making a generative chatbot that can chat with user without rigorously coding every response. Using seq2seq model we have built a small chatbot that can respond to our queries, responses are nowhere near as contextually-relevant as replies from "ChatGPT" but it does responds to short general queries.

2.2. Literature Review

[2] "Towards Open Domain Chatbots — A GRU Architecture for Data Driven Conversations" by Asmund Kamphaug. In this paper, we propose a novel deep learning architecture for content recognition that consists of multiple levels of gated recurrent units (GRUs). The architecture is designed to capture complex sentence structure at multiple levels of abstraction, seeking content recognition for very wide domains, through a distributed, scalable representation of content.

[3] "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation" Kyunghyun Cho. This paper proposes a novel neural network model called RNN Encoder–Decoder that consists of two recurrent neural networks (RNN). One RNN encodes a sequence of symbols into a fixed-length vector representation, and the other decodes the representation into another sequence of symbols. The encoder and decoder of the proposed model are jointly trained to maximize the conditional probability of a target sequence given a source sequence. The performance of a statistical machine translation system is empirically found to improve by using the conditional probabilities of phrase pairs computed by the RNN Encoder–Decoder as an additional feature in the existing log-linear model. Qualitatively, we show that the proposed model learns a semantically and syntactically meaningful representation of linguistic phrases.

[4] "A Neural Network based Vietnamese Chatbot" by Trang Nguyen and Maxim Shcherbakov employed the use of a Recurrent Neural Network in building a Vietnamese language-specific chatbot. It uses Sequence-to-Sequence models along with an attention mechanism. This made it able to handle new cases because they do not rely on any predefined response and creates its own response starting from the person they must respond to. These models are special, they can give the users the feeling of talking to a real human. Since there are no predefined answers, these models need to learn how to build responses using a large collection of conversations. Attention is calculated with another feed-forward layer in the decoder. The feed-forward layer uses the current input and hidden state to generate a new vector, which is the same size as the input sequence. This vector is processed through softmax to create attention weights that are multiplied by the encoders' outputs to create a new context vector. Then the new context vector is used to predict the next output. This model cannot handle long-term dependencies existing in the input data and thus fails to create a meaningful response.

In an IEEE research paper [5] published by Akhtar Rasool, Gaurav Hajela, and G Krishna Vamsi explored creating a chatbot using a deep neural learning method. In this method, a neural network with multiple layers is built to learn and process the data. In deep learning, a neural network with multiple layers is built for processing the input data or training data so that the model can extract higher-level features of the data. The conversational bot is trained on the dataset, which holds categories/classes, intents, patterns, responses, and context. To classify which category the user's message fits into, a special DNN is used, and then a random response from the list of responses is given. Natural Language Processing and Keras are used for creating a retrieval-based chatbot. The core aspects of Deep Learning are successive layers of representations, which helps in learning the features from low-level to high-level in a hierarchical manner, Deep Learning which is also known as a hierarchical representation of learning invalidated the feature engineering's higher-level dependence of shallow networks A feed-forward neural network type architecture is used in this model building in which the information moves only forward direction in the network. In each hidden layer, the input is transformed, and the output is transferred to the following layers.

Chapter Three: System Analysis

3.1. System Analysis

System analysis of a chatbot refers to the process of examining and breaking down the chatbot system into smaller parts to understand its behavior, interactions, and role in achieving the overall objectives. It involves identifying the requirements, feasibility, and creating models and diagrams to represent the chatbot system and its flow of information. It plays a crucial role in ensuring the successful development of a chatbot system by providing a clear understanding of its requirements and design.

3.1.1. Requirement Analysis

Requirement analysis of a chatbot refers to the process of identifying and documenting the functional and non-functional requirements that the chatbot should meet to successfully perform its intended tasks. By performing requirement analysis, the development team can ensure that the chatbot meets the needs of its users and performs optimally.

i. Functional Requirements

Functional requirements of a chatbot are the specific features and capabilities that it must have to meet its intended purpose. These requirements define what the chatbot should be able to do in terms of user interactions, data processing, integration with other systems, and overall functionality.

Functional requirement of our system:

- Chatbots should be able to understand and interpret user input, including natural language queries and commands.
- Chatbots should be able to provide relevant responses to user queries, based on the context of the conversation and user preferences.
- Chatbots should be able to provide personalized responses and recommendations based on user behavior, preferences, and historical data.
- The chatbot must be able to handle a large volume of text inputs and provide responses in a timely manner, even when dealing with a high volume of requests.

Use Case Diagram

Use case diagram is a diagrammatic representation that helps the user to represent the interaction of user with the system. Use Case Diagram shows the interaction between the system and the user in the particular environment. The use case model contains actors and the use cases. The actors are the external entities and the use cases are the functions of the system.

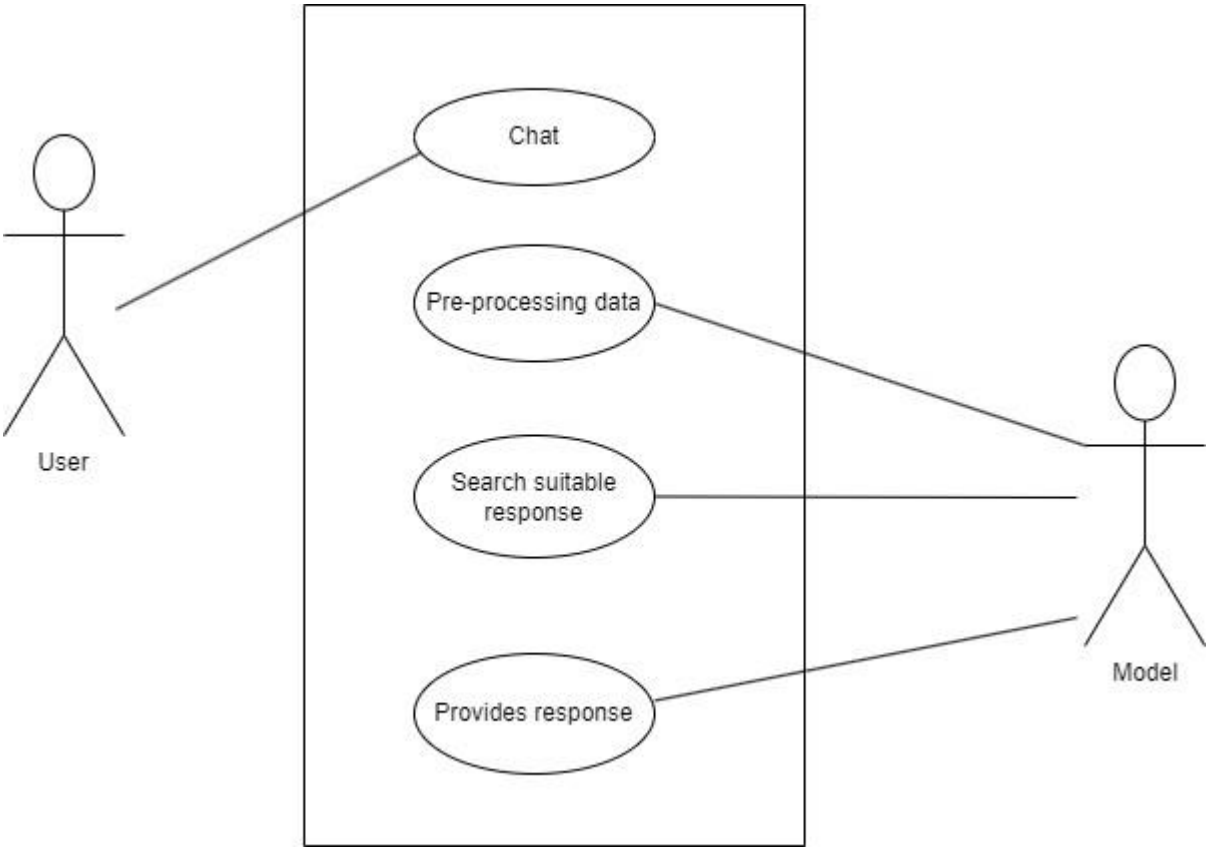


FIgure3. 1:Use Case Diagram

ii. Non-Functional Requirements

Non-functional requirements of a chatbot refer to the aspects of the system that describe how the system should perform, rather than what it should do. These requirements are typically related to the performance, usability and reliability of the chatbot system.

- Usability:

Our project is focused on the user preferences and convenient. So, it is created in such a manner that every user can easily access to every points and every data.

- Reliability**:**

For better performance, the system will be able to work for the user 24/7/365.

- Performance:

The higher the speed of the system, the higher the chance of the system's success. So, the system can be said to be faster and reliable.

3.1.2. Feasibility Study

Feasibility study of a chatbot refers to the process of assessing whether the development and implementation of the chatbot system is technically, operationally, economically, and schedule-wise feasible or not. The main objective of a feasibility study is to evaluate the potential success of a project and determine whether it should be pursued or not.

i. Technical

All the tools and software products required to construct this project is easily available in the internet. It does not require special environment to execute. Thus, it is affordable and it can be said to be technically feasible.

ii. Operational

All the functions of the system are possible to create. The calculations and database queries are possible to execute without any errors and extra requirements. The software configurations used by the system are possible to establish.

iii. Economic

This system is aimed for the equipment and system that is already in present. So, it does not require new equipment, it is economically feasible.

iv. Schedule

The analysis and coding portion takes a significant amount of time. Additionally, the research phase takes up the majority of the time. The project is finished after an operating program is created. As a result, the project was completed on time.

3.1.3. Project Process Modeling

Class And Object Diagram

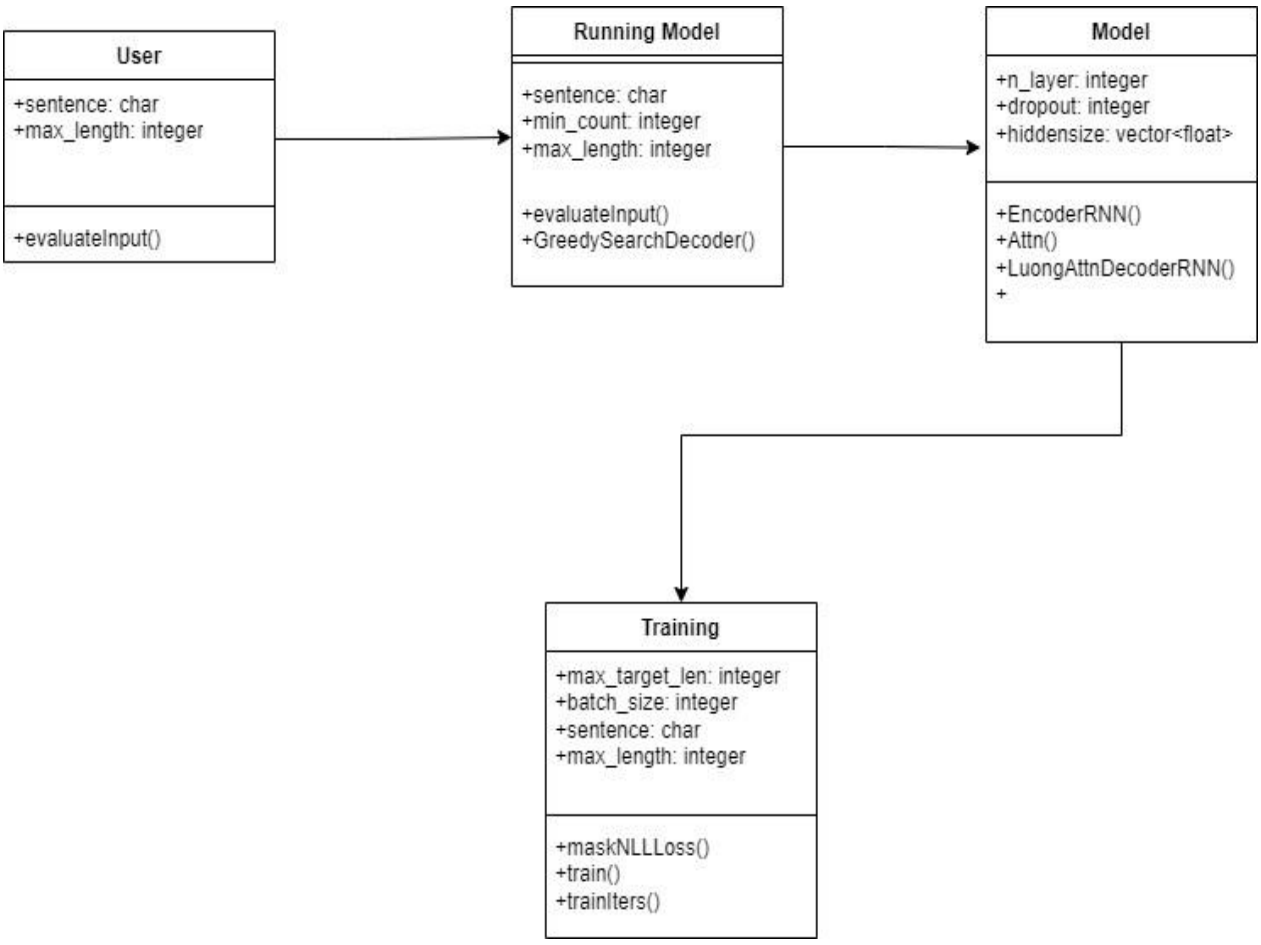


Figure3. 2:Class Diagram

Sequence Diagram

A sequence diagram is a type of diagram that illustrates the interactions and messages exchanged between objects or components in a system or process. A sequence diagram of a chatbot is a visual representation of the interactions between the user and the chatbot, showing the messages and responses exchanged between them in a sequential order. It helps to illustrate the flow of the chatbot's behavior and the steps involved in a conversation with the user.

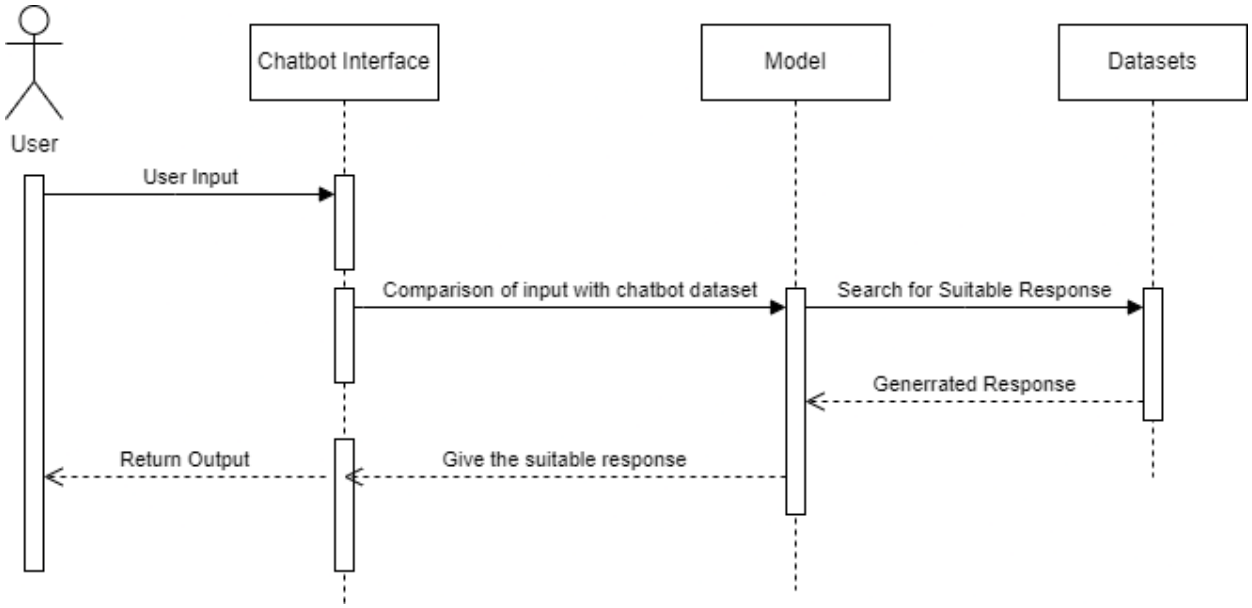


FIgure3. 3: Sequence Diagram

Context Diagram

A context diagram of a chatbot is a high-level visual representation of the chatbot system and its interactions with external entities, such as users and other systems. It helps to illustrate the overall architecture and context of the chatbot within its environment.

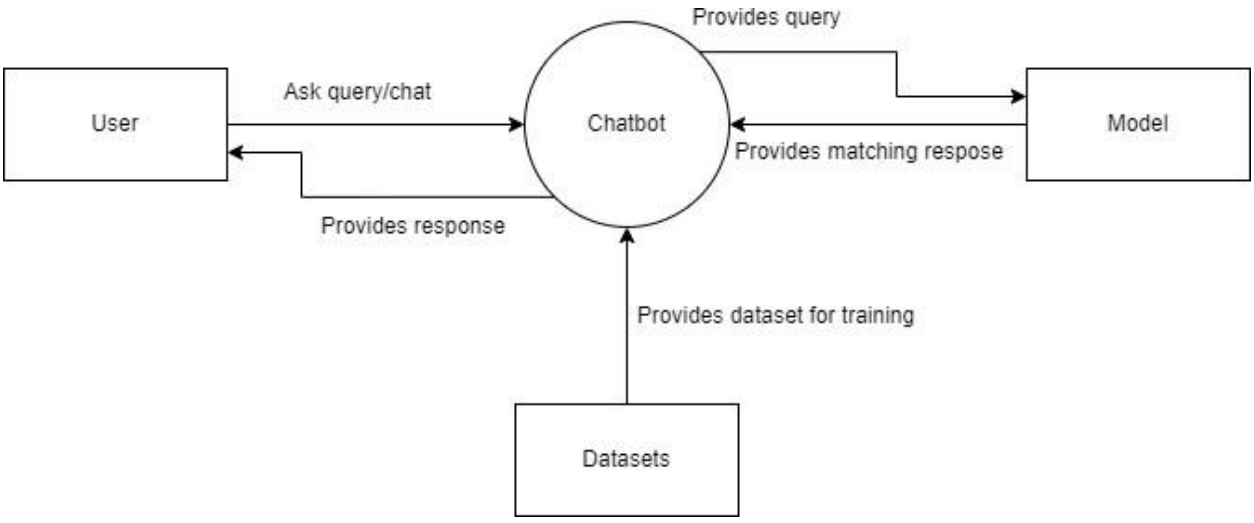


FIgure3. 4: Context Diagram

Activity diagram

An activity diagram in chatbots is a graphical representation of the sequence of activities involved in the operation of the system. It helps to visualize the logic of the chatbot and how it interacts with the user. Activity diagrams show the flow of actions and decision points of the chatbot.

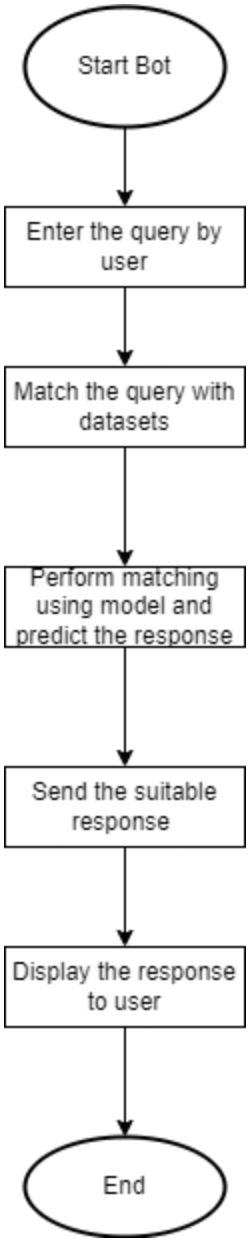


FIgure3. 5: Activity Diagram

3.2 Project Schedule

Gantt Chart

Gantt chart gives the detail information on how the project is done. It gives the visual graph of the project activities. The below figure shows the detail of activities conducted for the completion of the project.

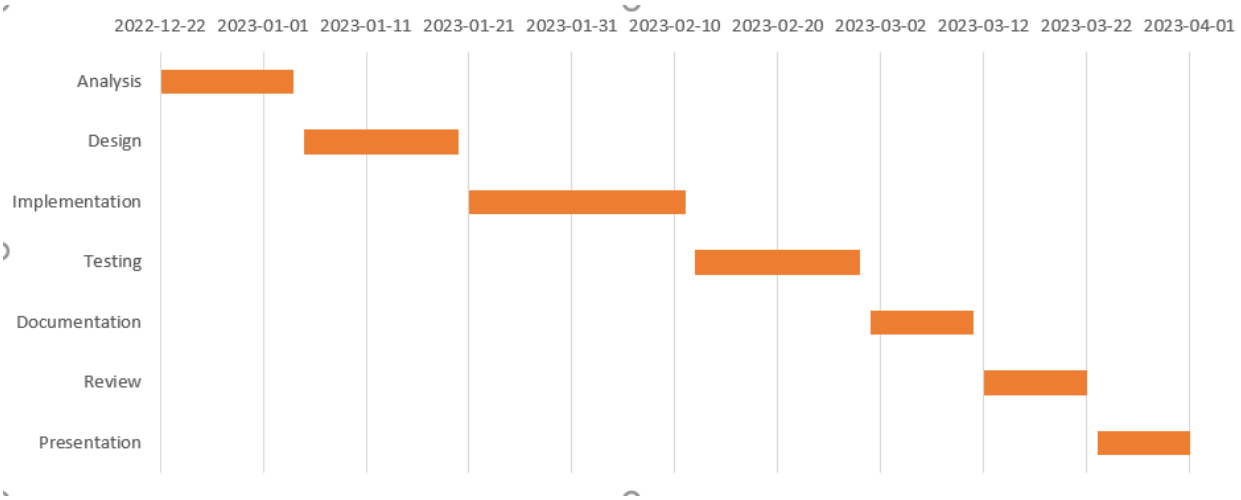


FIgure3. 6: Gnatt Chart

Chapter Four: System Design

4.1. Design

System design is the process of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements. It is a critical phase in the software development lifecycle and involves translating requirements into a detailed design that can be implemented by developers. A well-designed system is scalable, maintainable, and efficient, and meets the needs of its users.

System Architecture and Overview

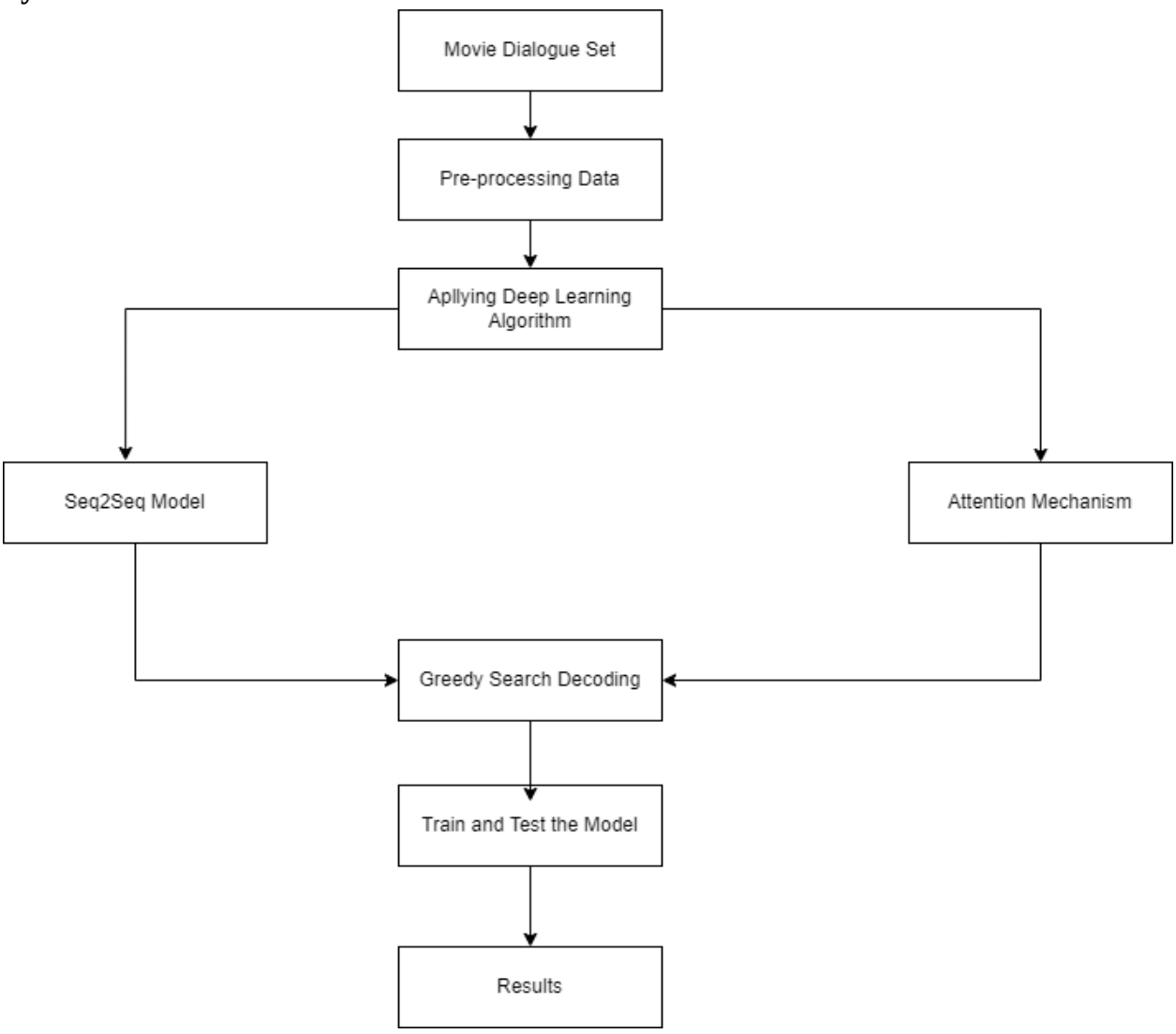


Figure4. 1: System Architecture of the system

The Cornell movie dialogue corpus is used as the dataset for our chatbot to train on. The movie corpus is then cleaned, stemmed, paired, and transformed into mini-batches for training. We apply a seq2seq model as our Deep learning algorithm. Here encoder uses GRU, and the decoder uses the Luong attention mechanism. With the help of teacher forcing and gradient clipping, we train our model in batches. Greedy search decoding is used to choose final answers when we are not using the teacher forcing Model is then evaluated and trained with parameters lastly user interacts with the bot to get a response.

4.2. Algorithm Details

Sequence-to-Sequence Model

Our approach makes use of the sequence-to-sequence (seq2seq) model. The model is based on a recurrent neural network that reads the input sequence one token at a time and predicts the output sequence, also one token at a time. During training, the true output sequence is given to the model, so learning can be done by backpropagation. The model is trained to maximize the cross-entropy of the correct sequence given its context. During inference, given that the true output sequence is not observed, we simply feed the predicted output token as input to predict the next output. This is a "greedy" inference approach. [6]

The brain of our chatbot is a sequence-to-sequence (seq2seq) model. The goal of a seq2seq model is to take a variable-length sequence as an input and return a variable-length sequence as an output using a fixed-sized model.

Sutskever discovered that by using two separate recurrent neural nets together, we can accomplish this task. One RNN acts as an **encoder**, which encodes a variable-length input sequence to a fixed-length context vector. In theory, this context vector (the final hidden layer of the RNN) will contain semantic information about the query sentence that is input to the bot. The second RNN is a **decoder**, which takes an input word and the context vector, and returns a guess for the next word in the sequence and a hidden state to use in the next iteration. [7]

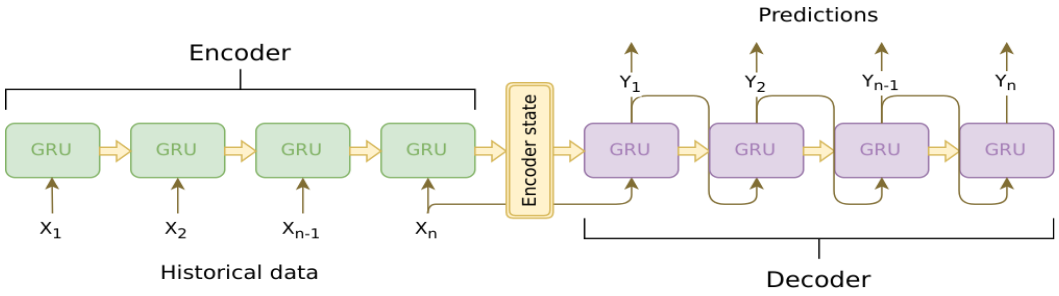


Figure4. 2: Sequence-to-Sequence Architecture

Encoder-Decoder Architecture:

The most common architecture used to build Seq2Seq models is the Encoder-Decoder architecture.

Encoder

The encoder RNN iterates through the input sentence one token (e.g., word) at a time, at each time step outputting an "output" vector and a "hidden state" vector. The hidden state vector is then passed to the next time step, while the output vector is recorded. The encoder transforms the context it saw at each point in the sequence into a set of points in a high-dimensional space, which the decoder will use to generate a meaningful output for the given task.

At the heart of our encoder is a multi-layered Gated Recurrent Unit, invented by Cho in 2014. We will use a bidirectional variant of the GRU, meaning that there are essentially two independent RNNs: one that is fed the input sequence in normal sequential order, and one that is fed the input sequence in reverse order. The outputs of each network are summed at each time step. Using a bidirectional GRU will give us the advantage of encoding both past and future contexts. [8]

Bidirectional RNN:

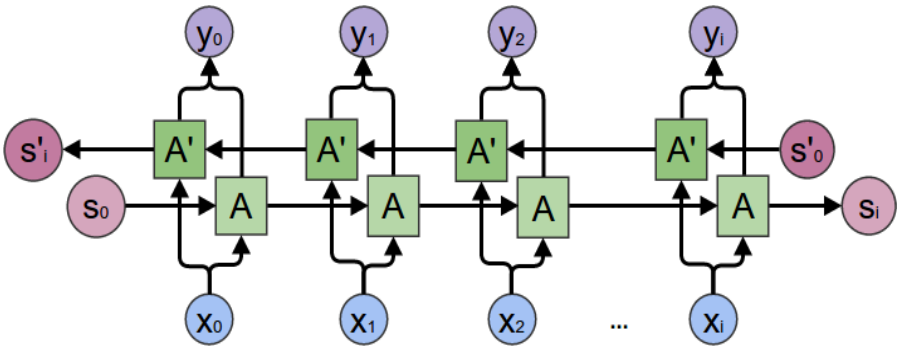


Figure4. 3: Seq2seq Encoder

Recurrent Neural Networks (RNN):

Recurrent Neural Networks (RNNs) are popular models that have shown great promise in many NLP tasks. The idea behind RNNs is to make use of sequential information. In a traditional neural network, we assume that all inputs (and outputs) are independent of each other. But for many tasks, that's a very bad idea. If you want to predict the next word in a sentence, you'd better know which words came before it. RNNs are called recurrent because they perform the same task for every element of a sequence, with the output depending on the previous computations. Another way to think about RNNs is that they have a "memory" which captures information about what has been calculated so far. In theory, RNNs can make use of information in arbitrarily long sequences, but in practice, they are limited to looking back only a few steps. [9]

Equations needed for training:

$$1)\quad h_t = f(W^{(hh)}h_{t-1} + W^{(hx)}x_t)$$

$$2)\quad y_t = softmax(W^{(S)}h_t)$$

$$3)\quad J^{(t)}(\theta) = \sum_{i=1}^{|V|} (y_{t_i}^{'} log\, y_{t_i})$$

Figure4. 4: Working of RNN

1) Holds information about the previous words in the sequence. As you can see, h_t is calculated using the previous h_(t-1) vector and current word vector x_t. We also apply a non-linear activation function f (usually tanh or sigmoid) to the final summation. It is acceptable to assume that h_0 is a vector of zeros.

2) Calculates the predicted word vector at a given time step t. We use the softmax function to produce a (V,1) vector with all elements summing up to 1. This probability distribution gives us the index of the most likely next word from the vocabulary.

3) Uses the cross-entropy loss function at each time step t to calculate the error between the predicted and actual word

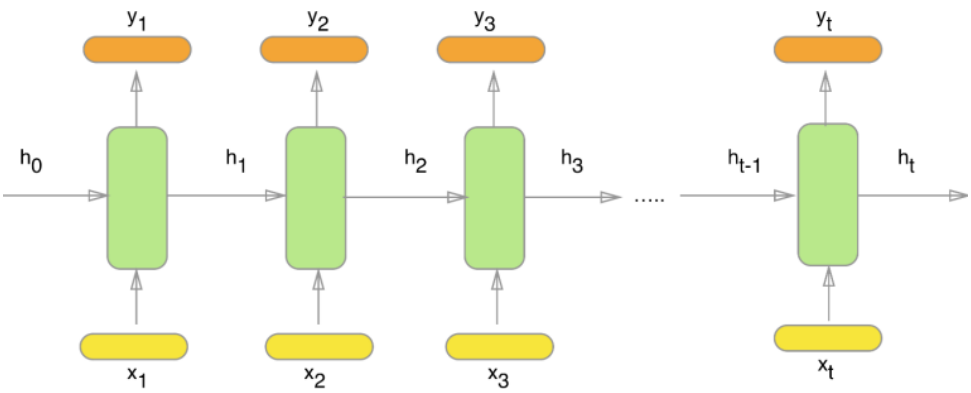


Figure4. 5: RNN architecture

Gated Recurrent Units :

Beyond the extensions discussed so far, RNNs have been found to perform better with the use of more complex units for activation. So far, we have discussed methods that transition from hidden state ht−1 to ht using an affine transformation and a point-wise nonlinearity. Here, we discuss the use of a gated activation function, thereby modifying the RNN architecture. What motivates this? Well, although RNNs can theoretically capture long-term dependencies, they are very hard to actually train to do this. Gated recurrent units are designed in a manner to have more persistent memory, thereby making it easier for RNNs to capture long-term dependencies. Let us

see

Mathematically, how a GRU uses ht−1 and xt to generate the next hidden state ht. We will then dive into the intuition of this architecture. [10]

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1}) \quad \text{(Update gate)}$$
$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1}) \quad \text{(Reset gate)}$$
$$\tilde{h}_t = \tanh(r_t \circ Uh_{t-1} + Wx_t) \quad \text{(New memory)}$$
$$h_t = (1 - z_t) \circ \tilde{h}_t + z_t \circ h_{t-1} \quad \text{(Hidden state)}$$

Figure4. 6: Working of GRU

The above equations can be thought of a GRU's four fundamental operational stages and they have intuitive interpretations that make this model much more intellectually satisfying.

1. New memory generation: A new memory ˜ht is the consolidation of a new input word xt with the past hidden state ht−1. Anthropomorphically, this stage is the one who knows the recipe of combining a newly observed word with the past hidden state ht−1 to summarize this new word in light of the contextual past as the vector ˜ht .

2. Reset Gate: The reset signal rt is responsible for determining how important ht−1 is to the summarization ˜ht . The reset gate has the ability to completely diminish past hidden state if it finds that ht−1 is irrelevant to the computation of the new memory.

3. Update Gate: The update signal zt is responsible for determining how much of ht−1 should be carried forward to the next state. For instance, if zt ≈ 1, then ht−1 is almost entirely copied out to ht. Conversely, if zt ≈ 0, then mostly the new memory ˜ht is forwarded to the next hidden state

. 4. Hidden state: The hidden state ht is finally generated using the past hidden input ht−1 and the new memory generated ˜ht with the advice of the update gate.[6]

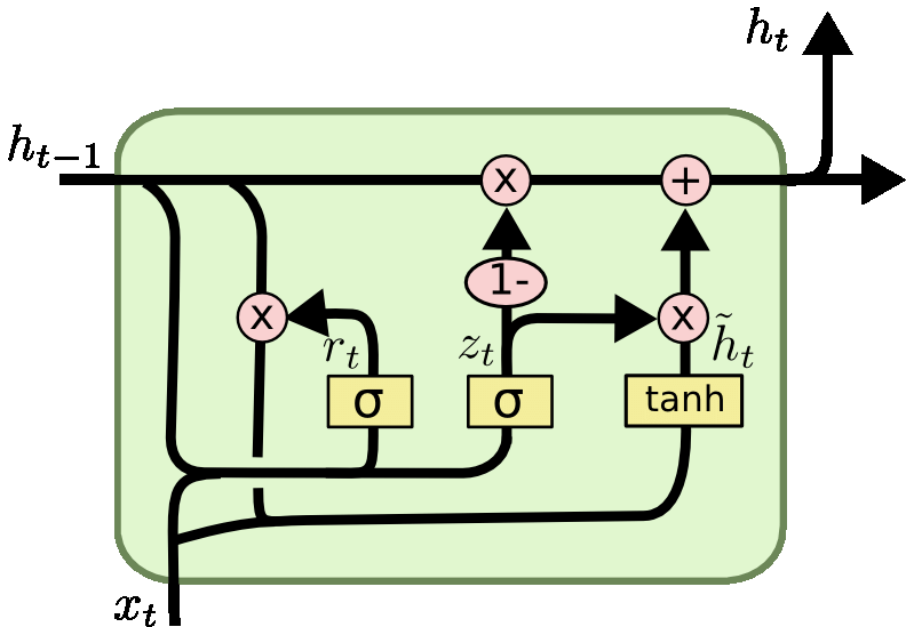


Figure4. 7: single cell GRU

Decoder:

The decoder RNN generates the response sentence in a token-by-token fashion. It uses the encoder's context vectors, and internal hidden states to generate the next word in the sequence. It continues generating words until it outputs an "*EOS_token",* representing the end of the sentence. It also uses a bidirectional GRU to decode the output generated from Encoder as its input.

A common problem with a vanilla seq2seq decoder is that if we rely solely on the context vector to encode the entire input sequence's meaning, it is likely that we will have information loss. This is especially the case when dealing with long input sequences, greatly limiting the capability of our decoder. To combat this, Bahdanau created an "attention mechanism" that allows the decoder to pay attention to certain parts of the input sequence, rather than using the entire fixed context at every step.[4]

Attention:

[8]At a high level, attention is calculated using the decoder's current hidden state and the encoder's outputs. The output attention weights have the same shape as the input sequence, allowing us to multiply them by the encoder outputs, giving us a weighted sum, which indicates the parts of the encoder output to pay attention to. Sean Robertson's figure describes this very well:

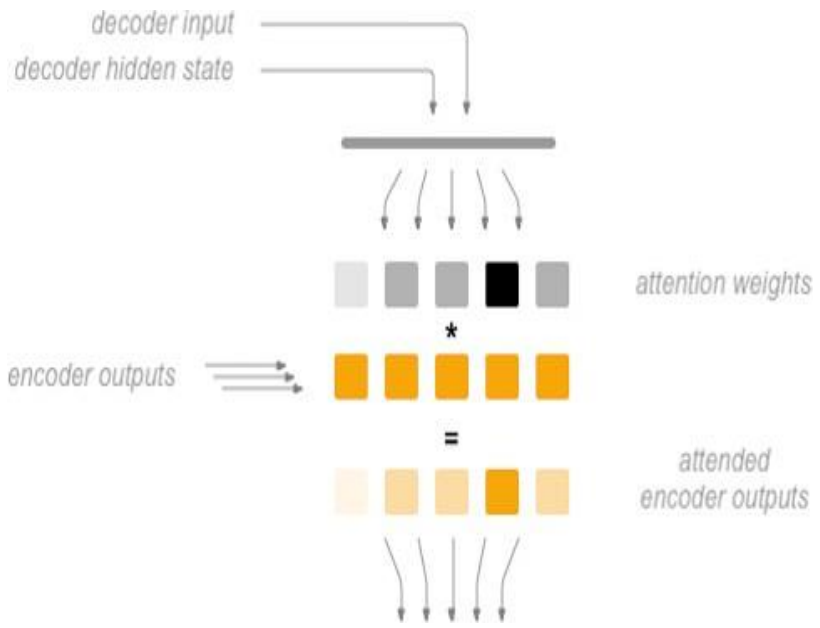


Figure4. 8: Architecture of Attention

[4] Luong improved upon Bahdanau's groundwork by creating "Global attention". The key difference is that with "Global attention", we consider all of the encoder's hidden states, as opposed to Bahdanau's "Local attention", which only considers the encoder's hidden state from the current time step. Another difference is that with "Global attention", we calculate attention weights, or energies, using the hidden state of the decoder from the current time step only. Bahdanau's attention calculation requires knowledge of the decoder's state from the previous time step. Also, Luong provides various methods to calculate the attention energies between the encoder output and decoder output, which are called "score functions":

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^\top \boldsymbol{W_a} \bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W_a}[\boldsymbol{h}_t; \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases}$$

Figure4. 9:Working of Attention

Overall, the Global attention mechanism can be summarized by the following figure.

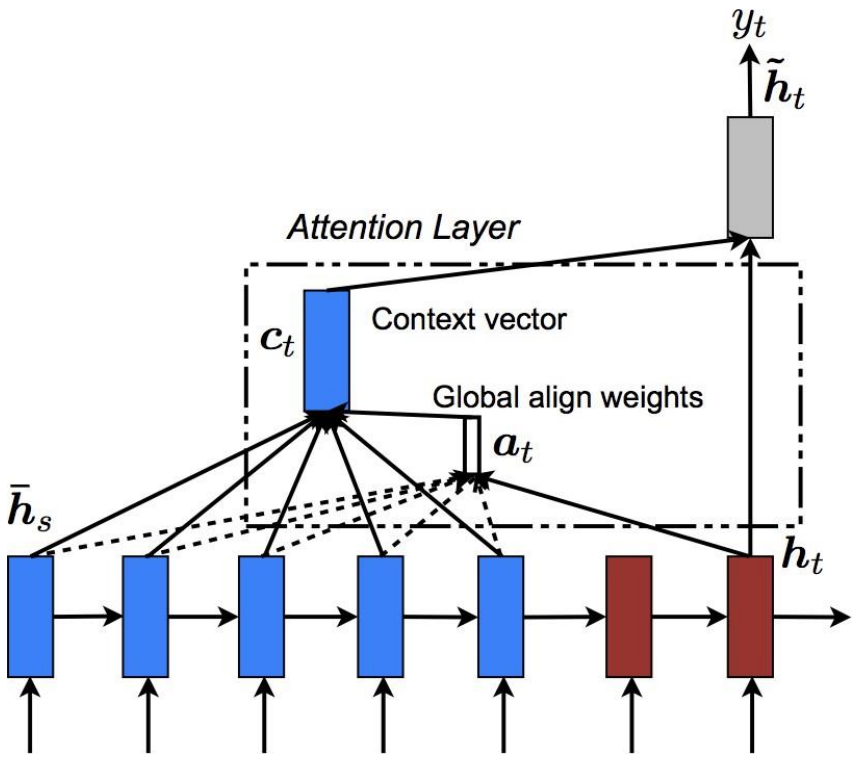


Figure4. 10: Block diagram of Global attention

The training contains the algorithm for a single training iteration (a single batch of inputs).

[8]We will use a couple of clever tricks to aid in convergence:

- The first trick is using **teacher forcing**. This means that at some probability, set by the teacher forcing ratio, we use the current target word as the decoder's next input rather than using the decoder’s current guess. This technique acts as training wheels for the decoder, aiding in more efficient training. However, teacher forcing can lead to model instability during inference, as the decoder may not have a sufficient chance to truly craft its output sequences during training. Thus, we must be mindful of how we are setting the teacher forcing ratio, and not be fooled by fast convergence.
- The second trick that we implement is **gradient clipping**. This is a commonly used technique for countering the “exploding gradient” problem. In essence, by clipping or thresholding gradients to a maximum value, we prevent the gradients from growing exponentially and either overflow (NaN), or overshoot steep cliffs in the cost function.

Chapter Five: Implementation and Testing

5.1. Implementation

Chatbot is a generative neural machine translation project based on artificial intelligence, where Sequence-to-Sequence(using GRU) is used to converse between humans and artificial intelligence. The chatbot is developed in the programming language Python, with a large movie corpus called cornell-movie-corpus and different libraries such as torch, re, math, cuda, etc have been used to build the model.

5.1.1. Tools Used

Python: Python is our main coding language for this project.

Pytorch: Pytorch is an open-source machine learning library used for building models.

Microsoft Word 365: Microsoft Word 365 is used for documentation of a project.

Draw.io: Draw.io is used to draw a UML diagram.

Google Colab: Google Colab is used for training our project using a cloud GPU.

Jupyter notebook: Jupyter notebook is used for running our final model.

5.1.2. Implementation Details of Modules

The algorithm of the overall system is as follows:

1. Data Collection: Using the Cornell Movie Dialogs Corpus dataset, which is a large dataset with some of the features like 220,579 conversational exchanges between 10,292 pairs of movie characters-involves 9,035 characters from 617 movies- in total 304,713 utterances.
2. Data Preprocessing: To remove noise in a dataset or other object removal, different pre-processing techniques are considered removing stopwords, removing patterns, stemming, forming conversation pairs, padding tokens, etc.
3. Preparing Data for Models: preparing mini-batches for speeding up the training process and GPU parallelization.
4. Define Models: Here, we form the encoder using GRU and the decoder with a Luong attention layer.
5. Define Training Procedure: Here, masked loss, single training iteration, and training iteration are done.
6. Define Evaluation: After training a model, we want to be able to talk to the bot ourselves. So greedy decoding, evaluation are done.
7. Run Model & training: Here we load our model with parameters and train the model for the given iteration.
8. Run Evaluation: To chat with the model, we run the evaluation.

5.2. Testing

Testing is an important part of the software development process. If a system is timely tested according to various references, then it improves the output of the system and enhances its functionality. Testing can also be termed as 'Verification and Validation' on a timely basis. Both verification and validation are important for the system to be as it was expected. Validation refers to the process of using the new software for the developed system in a live environment. The validation phase reveals the failures and bugs in the developed system. It will become known about the practical difficulties the system faces when operated in the true environment.

5.2.1. Test Cases for System Testing

5.2.1.1 Integration Testing

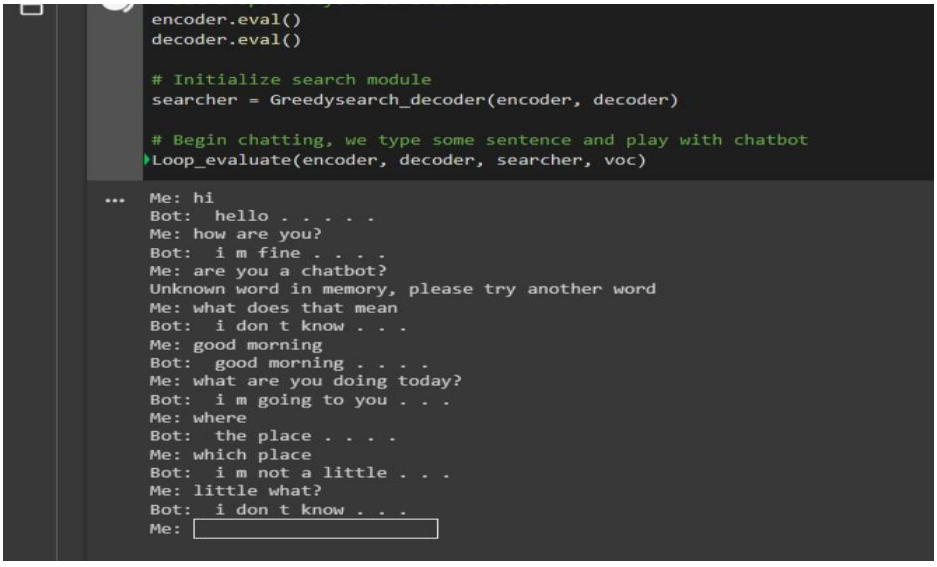
This testing was mainly emphasized to check whether the integrated part worked well or not. Separate development units that make up a component of the system are tested to ensure that they work together.

5.2.1.1 Test Cases for Unit Testing

Test Case 1

Test objectives: test chatbot for 10000 iterations of training, 2 encoder and decoder layers.

Output:



Test Case 2

Test objectives: test chatbot for 20000 iterations of training with 3 encoder and decoder layers.

Output:

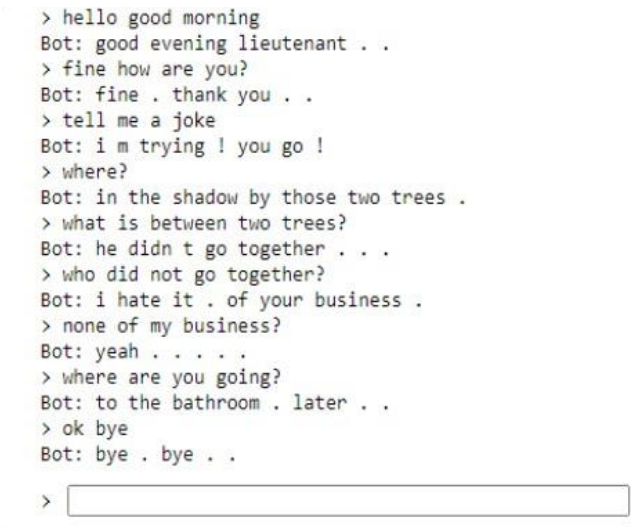
```
> hi good morning
Bot: hi honey . how s it going ?
> fine how are you?
Bot: fine . thank you . .
> what are you doing today?
Bot: i m looking for a hospital . .
> why?
Bot: because i m expecting a call .
> whose?
Bot: yours . . . !
> but i am not calling you
Bot: i know that i know .
> you are weird
Bot: i am ! my name
> okay what is your name?
Bot: phoebe . of course . !
> so what do you do?
Bot: i m a dentist of a southland .
> goodbye have a nice day
Bot: thank you .

>
```

Test Case 3

Test objectives: test chatbot for 30000 iterations of training with 3 encoder and decoder layers.

Output:



5.3. Result Analysis

The result analysis of our project, where we built a chatbot using a sequence-to-sequence (Seq2Seq) model. The chatbot was trained using a PyTorch framework, employing a 3x3 encoder-decoder layer architecture.

Training Iterations and Loss:

We conducted training for a total of 20,000 iterations. The loss was measured using masked entropy loss, which is a common loss function for sequence generation tasks. The average loss at the end of training was 0.6448, indicating the model's ability to minimize the discrepancy between the predicted responses and the ground truth responses.

Comparison between 3x3 and 2x2 Encoder-Decoder Layers:

We experimented with two different configurations of the encoder-decoder layer: 3x3 and 2x2. The chatbot utilizing the 3x3 layer architecture consistently provided better responses compared to the 2x2 layer. This suggests that increasing the depth and complexity of the model enhances its capability to generate more accurate and coherent responses.

Iteration Progress and Response Quality:

The quality of responses improved as the training progressed. Comparing the responses after 20,000 iterations to those after 10,000 iterations, we observed an enhancement in the chatbot's ability to generate more relevant replies. The reduction in average loss over iterations further supports the notion that the model's performance improved as it learned from the training data.

Initial and Final Cross-Entropy Loss:

At the beginning of training (1 iteration), the cross-entropy loss was 0.9795. By the end of training (20,000 iterations), the average loss decreased to 0.6448. This reduction in loss signifies the model's progress in capturing the patterns and nuances of the training data, leading to more accurate response generation.

In conclusion, our chatbot, implemented using a Seq2Seq model with a 3x3 encoder-decoder layer architecture, achieved promising results. The chatbot's responses improved with increased training iterations, and the use of a deeper encoder-decoder layer contributed to enhanced response quality. The reduction in average loss demonstrated the model's ability to learn and generate contextually appropriate replies.

Chapter Six: Conclusion and Future Recommendations

6.1. Conclusion

Chatbots are made up of special software where the conversation will be similar to a human interaction, but the replies are given by a bot. It results in giving automatic replies, and it is available 24*7, and it also reduces the human effort. Many organizations are using chatbots as their query solver in the help section. In this project, we took a movie titles dataset and preprocessed it until we got the cleaned data. Using the closed domain models, the user will get the right response, which is already predefined in the repository. Here we used a Seq2Seq model, an attention mechanism, Recurrent Neural Network to build the chatbot. Although chatbots make the work of the organization easy by providing hasty responses to users, the research area in generative models makes it difficult to make a life-like chatbot.

6.2. Future Recommendations

In the future our model can be trained further with more datasets with different languages and can be built in a way such that the chatbot can be able to give answers in many languages. Given the bigger dataset and good infrastructure we can build more accurate and context-relevant chatbots.

References

[1] J. Weizenbaum, "A computer program for the study of natural language communication between man and machine," in *Commun, ACM*, 1966, pp. 36-45.

[2] A. Kamphaung, "Towards Open Domain Chatbots — A GRU Architecture for Data Driven Conversations".

[3] K. Cho, "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation".

[4] S. Maxim and N. Trang, "A Neural Network-based Vietnamese Chatbot".

[5] A. Rasool, G. Hajela and V. G. Krishna, "Chatbot-A Deep Neural Network Based Human to Machine Conversation Model".

[6] O. Vinyals, "A Neural Conversational Model".

[7] S. Ilya, O. Vinyals and V. L. Quoc, "Sequence to Sequence Learning with Neural Networks".

[8] "Chatbot Tutorial — PyTorch Tutorials 2.0.0+cu117 documentation".

[9] M. G. LIKHITH, "Conversational Bot using Seq2Seq Model".

[10] In *CS224n: Natural Language Processing with Deep Learning 1*, p. Lecture Notes: Part V.

Appendices

Data Preprocessing

```python
# Splits each line of the file to create lines and conversations
def loadLinesAndConversations(fileName):
    lines = {}
    conversations = {}
    with open(fileName, 'r', encoding="iso-8859-1') as f:
        for line in f:
            lineJson = json.loads(line)
            # Extract fields for line object
            lineObj = {}
            lineObj["lineID"] = lineJson["id"]
            lineObj["characterID"] = lineJson["speaker"]
            lineObj["text"] = lineJson["text"]
            lines[lineObj['lineID']] = lineObj

            # Extract fields for conversation object
            if lineJson["conversation_id"] not in conversations:
                convObj = {}
                convObj["conversationID"] = lineJson["conversation_id"]
                convObj["movieID"] = lineJson["meta"]["movie_id"]
                convObj["lines"] = [lineObj]
            else:
                convObj = conversations[lineJson["conversation_id"]]
                convObj["lines"].insert(0, lineObj)
            conversations[convObj["conversationID"]] = convObj

    return lines, conversations


# Extracts pairs of sentences from conversations
def extractSentencePairs(conversations):
    qa_pairs = []
    for conversation in conversations.values():
        # Iterate over all the lines of the conversation
        for i in range(len(conversation["lines"]) - 1):  # We ignore the last line (no answer for it)
            inputLine = conversation["lines"][i]["text"].strip()
            targetLine = conversation["lines"][i+1]["text"].strip()
            # Filter wrong samples (if one of the lists is empty)
            if inputLine and targetLine:
                qa_pairs.append([inputLine, targetLine])
    return qa_pairs
```

```python
class Voc:
    def __init__(self, name):
        self.name = name
        self.trimmed = False
        self.word2index = {}
        self.word2count = {}
        self.index2word = {PAD_token: "PAD", SOS_token: "SOS", EOS_token: "EOS"}
        self.num_words = 3  # Count SOS, EOS, PAD

    def addSentence(self, sentence):
        for word in sentence.split(' '):
            self.addWord(word)

    def addWord(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.num_words
            self.word2count[word] = 1
            self.index2word[self.num_words] = word
            self.num_words += 1
        else:
            self.word2count[word] += 1

    # Remove words below a certain count threshold
    def trim(self, min_count):
        if self.trimmed:
            return
        self.trimmed = True

        keep_words = []

        for k, v in self.word2count.items():
            if v >= min_count:
                keep_words.append(k)

        print('keep_words {} / {} = {:.4f}'.format(
            len(keep_words), len(self.word2index), len(keep_words) / len(self.word2index)
        ))

        # Reinitialize dictionaries
        self.word2index = {}
        self.word2count = {}
        self.index2word = {PAD_token: "PAD", SOS_token: "SOS", EOS_token: "EOS"}
        self.num_words = 3 # Count default tokens

        for word in keep_words:
            self.addWord(word)
```

```
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
    )

# Lowercase, trim, and remove non-letter characters
def normalizeString(s):
    s = unicodeToAscii(s.lower().strip())
    s = re.sub(r"([.!?])", r" \1", s)
    s = re.sub(r"[^a-zA-Z.!?]+", r" ", s)
    s = re.sub(r"\s+", r" ", s).strip()
    return s

# Read query/response pairs and return a voc object
def readVocs(datafile, corpus_name):
    print("Reading lines...")
    # Read the file and split into lines
    lines = open(datafile, encoding='utf-8').\
        read().strip().split('\n')
    # Split every line into pairs and normalize
    pairs = [[normalizeString(s) for s in l.split('\t')] for l in lines]
    voc = Voc(corpus_name)
    return voc, pairs

# Returns True if both sentences in a pair 'p' are under the MAX_LENGTH threshold
def filterPair(p):
    # Input sequences need to preserve the last word for EOS token
    return len(p[0].split(' ')) < MAX_LENGTH and len(p[1].split(' ')) < MAX_LENGTH

# Filter pairs using the ``filterPair`` condition
def filterPairs(pairs):
    return [pair for pair in pairs if filterPair(pair)]

# Using the functions defined above, return a populated voc object and pairs list
def loadPrepareData(corpus, corpus_name, datafile, save_dir):
    print("Start preparing training data ...")
    voc, pairs = readVocs(datafile, corpus_name)
    print("Read {!s} sentence pairs".format(len(pairs)))
    pairs = filterPairs(pairs)
    print("Trimmed to {!s} sentence pairs".format(len(pairs)))
    print("Counting words...")
    for pair in pairs:
        voc.addSentence(pair[0])
        voc.addSentence(pair[1])
    print("Counted words:", voc.num_words)
    return voc, pairs
```

Encoder

```
class EncoderRNN(nn.Module):
    def __init__(self, hidden_size, embedding, n_layers=1, dropout=0):
        super(EncoderRNN, self).__init__()
        self.n_layers = n_layers
        self.hidden_size = hidden_size
        self.embedding = embedding

        # Initialize GRU; the input_size and hidden_size parameters are both set to 'hidden_size'
        #   because our input size is a word embedding with number of features == hidden_size
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers,
                          dropout=(0 if n_layers == 1 else dropout), bidirectional=True)

    def forward(self, input_seq, input_lengths, hidden=None):
        # Convert word indexes to embeddings
        embedded = self.embedding(input_seq)
        # Pack padded batch of sequences for RNN module
        packed = nn.utils.rnn.pack_padded_sequence(embedded, input_lengths)
        # Forward pass through GRU
        outputs, hidden = self.gru(packed, hidden)
        # Unpack padding
        outputs, _ = nn.utils.rnn.pad_packed_sequence(outputs)
        # Sum bidirectional GRU outputs
        outputs = outputs[:, :, :self.hidden_size] + outputs[:, : ,self.hidden_size:]
        # Return output and final hidden state
        return outputs, hidden
```

Attention

```
# Luong attention layer
class Attn(nn.Module):
    def __init__(self, method, hidden_size):
        super(Attn, self).__init__()
        self.method = method
        if self.method not in ['dot', 'general', 'concat']:
            raise ValueError(self.method, "is not an appropriate attention method.")
        self.hidden_size = hidden_size
        if self.method == 'general':
            self.attn = nn.Linear(self.hidden_size, hidden_size)
        elif self.method == 'concat':
            self.attn = nn.Linear(self.hidden_size * 2, hidden_size)
            self.v = nn.Parameter(torch.FloatTensor(hidden_size))

    def dot_score(self, hidden, encoder_output):
        return torch.sum(hidden * encoder_output, dim=2)

    def general_score(self, hidden, encoder_output):
        energy = self.attn(encoder_output)
        return torch.sum(hidden * energy, dim=2)

    def concat_score(self, hidden, encoder_output):
        energy = self.attn(torch.cat((hidden.expand(encoder_output.size(0), -1, -1), encoder_output), 2)).tanh()
        return torch.sum(self.v * energy, dim=2)

    def forward(self, hidden, encoder_outputs):
        # Calculate the attention weights (energies) based on the given method
        if self.method == 'general':
            attn_energies = self.general_score(hidden, encoder_outputs)
        elif self.method == 'concat':
            attn_energies = self.concat_score(hidden, encoder_outputs)
        elif self.method == 'dot':
            attn_energies = self.dot_score(hidden, encoder_outputs)

        # Transpose max_length and batch_size dimensions
        attn_energies = attn_energies.t()

        # Return the softmax normalized probability scores (with added dimension)
        return F.softmax(attn_energies, dim=1).unsqueeze(1)
```

Decoder

```
class LuongAttnDecoderRNN(nn.Module):
    def __init__(self, attn_model, embedding, hidden_size, output_size, n_layers=1, dropout=0.1):
        super(LuongAttnDecoderRNN, self).__init__()

        # Keep for reference
        self.attn_model = attn_model
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout = dropout

        # Define layers
        self.embedding = embedding
        self.embedding_dropout = nn.Dropout(dropout)
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers, dropout=(0 if n_layers == 1 else dropout))
        self.concat = nn.Linear(hidden_size * 2, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)

        self.attn = Attn(attn_model, hidden_size)

    def forward(self, input_step, last_hidden, encoder_outputs):
        # Note: we run this one step (word) at a time
        # Get embedding of current input word
        embedded = self.embedding(input_step)
        embedded = self.embedding_dropout(embedded)
        # Forward through unidirectional GRU
        rnn_output, hidden = self.gru(embedded, last_hidden)
        # Calculate attention weights from the current GRU output
        attn_weights = self.attn(rnn_output, encoder_outputs)
        # Multiply attention weights to encoder outputs to get new "weighted sum" context vector
        context = attn_weights.bmm(encoder_outputs.transpose(0, 1))
        # Concatenate weighted context vector and GRU output using Luong eq. 5
        rnn_output = rnn_output.squeeze(0)
        context = context.squeeze(1)
        concat_input = torch.cat((rnn_output, context), 1)
        concat_output = torch.tanh(self.concat(concat_input))
        # Predict next word using Luong eq. 6
        output = self.out(concat_output)
        output = F.softmax(output, dim=1)
        # Return output and final hidden state
        return output, hidden
```

Creating Inference
Console log:

```
def evaluate(encoder, decoder, searcher, voc, sentence, max_length=MAX_LENGTH):
    ### Format input sentence as a batch
    # words -> indexes
    indexes_batch = [indexesFromSentence(voc, sentence)]
    # Create lengths tensor
    lengths = torch.tensor([len(indexes) for indexes in indexes_batch])
    # Transpose dimensions of batch to match models' expectations
    input_batch = torch.LongTensor(indexes_batch).transpose(0, 1)
    # Use appropriate device
    input_batch = input_batch.to(device)
    lengths = lengths.to("cpu")
    # Decode sentence with searcher
    tokens, scores = searcher(input_batch, lengths, max_length)
    # indexes -> words
    decoded_words = [voc.index2word[token.item()] for token in tokens]
    return decoded_words


def evaluateInput(encoder, decoder, searcher, voc):
    input_sentence = ''
    while(1):
        try:
            # Get input sentence
            input_sentence = input('> ')
            # Check if it is quit case
            if input_sentence == 'q' or input_sentence == 'quit': break
            # Normalize sentence
            input_sentence = normalizeString(input_sentence)
            # Evaluate sentence
            output_words = evaluate(encoder, decoder, searcher, voc, input_sentence)
            # Format and print response sentence
            output_words[:] = [x for x in output_words if not (x == 'EOS' or x == 'PAD')]
            print('Bot:', ' '.join(output_words))

        except KeyError:
            print("Error: Encountered unknown word.")
```

Tkinter interface:

```
import tkinter as tk

def evaluateInput(encoder, decoder, searcher, voc):
    def get_input(event=None):
        input_sentence = input_entry.get()
        input_entry.delete(0, tk.END)

        # Check if it is quit case
        if input_sentence.lower() == 'q' or input_sentence.lower() == 'quit':
            window.quit()
            return

        # Normalize sentence
        input_sentence = normalizeString(input_sentence)

        # Add user's input to the conversation history
        conversation_text.config(state=tk.NORMAL)
        conversation_text.insert(tk.END, 'User: ' + input_sentence + '\n', 'user')

        try:
            # Evaluate sentence
            output_words = evaluate(encoder, decoder, searcher, voc, input_sentence)

            # Format and print response sentence
            output_words[:] = [x for x in output_words if not (x == 'EOS' or x == 'PAD')]
            conversation_text.insert(tk.END, 'Bot: ' + ' '.join(output_words) + '\n', 'bot')
        except KeyError:
            # Print error message for unknown word
            conversation_text.insert(tk.END, 'Bot: Error: Encountered unknown word.\n', 'error')

        conversation_text.config(state=tk.DISABLED)

        # Scroll to the bottom of the conversation
        conversation_text.yview(tk.END)

    window = tk.Tk()
    window.title('Chatbot')

    input_frame = tk.Frame(window)
    input_frame.pack(side=tk.BOTTOM, pady=10, fill=tk.X)

    input_label = tk.Label(input_frame, text='User Input:')
    input_label.pack(side=tk.LEFT)

    input_entry = tk.Entry(input_frame)
    input_entry.pack(side=tk.LEFT, padx=(0, 10), expand=True, fill=tk.X)
    input_entry.focus()  # Set focus to the input field

    input_frame.columnconfigure(1, weight=1)  # Stretch the input field

    input_entry.bind("<Return>", get_input)  # Bind the Enter key event

    conversation_frame = tk.Frame(window)
    conversation_frame.pack(side=tk.BOTTOM, padx=10, pady=10, fill=tk.BOTH, expand=True)

    conversation_text = tk.Text(conversation_frame, state=tk.DISABLED, wrap=tk.WORD)
    conversation_text.pack(side=tk.BOTTOM, fill=tk.BOTH, expand=True)

    # Configure tags for user, bot, and error messages
    conversation_text.tag_configure('user', justify='right', foreground='blue')
    conversation_text.tag_configure('bot', justify='left', foreground='green')
    conversation_text.tag_configure('error', justify='left', foreground='red')

    conversation_text.bind("<Enter>", lambda event: conversation_text.bind_all("<MouseWheel>", lambda event: conversation_text.yv
    conversation_text.bind("<Leave>", lambda event: conversation_text.unbind_all("<MouseWheel>"))

    window.mainloop()
```

LOGS OF VISIT TO SUPERVISOR

S.N.	Date	Topic Discussed	Signature of Supervisor	Remarks
1	Dec 23, 2022	Proposal Submission		
2	Jan 18, 2023	Discuss related to the project topic with the supervisor.		
3	Jan 23, 2023	Discussion regarding the datasets we are going to use		
4	March 12, 2023	Discussion about the ongoing project		
5	May 2, 2023	Demonstration of the project to the supervisor		