CHATBOT: OVERVIEW,DESIGN,AND ARCHITECUTRE

Introduction

A dialogue system is an example of an intelligent agent that intends to hold natural conversations with humans. More specifically, it is a software application, with the help of natural language processing and machine learning, that stimulates human conversation in natural language via text or text-to-speech.

Dialogue systems have come a long way since their inception in the 1960s. Both the hardware and software aspects of computer science, and natural language processing and machine learning techniques have developed tremendously. Thanks to the advancement of these emerging technologies, chatbots have evolved from systems that generate machine-like responses to humanlike agents capable of developing long-term relationships with users. Among the most famous early chatbot implementations are ELIZA and PARRY . Modern chatbots include Apple's Siri and Amazon's Alexa, and Microsoft's XiaoIce.

Chatbots promise a range of potential benefits. Most notably, they provide responses and solutions that are instant, consistent, and reliable [53]. These characteristics make chatbots a powerful tool in many different areas, and their use has evolved rapidly in fields such as business, e-commerce, and healthcare.

This paper is a literature review on chatbot technology. Section 2 will provide a brief overview and history of chatbots. Section 3 will introduce two design approaches to chatbots. In Section 4, we will delve into the general architecture of chatbots and describe the technologies that support each component in detail. Finally, we will address existing chatbot applications and social and ethical considerations in Section 5.

2 Overview of Chatbots

2.1 The Rise of Chatbots

The idea of a chatbot was inspired by Alan Turing. In 1950, Turing proposed the Turing Test and asked whether machines can think. Since then, conversational systems have attracted extensive attention and become an important concept in artificial intelligence .

The first publicly known chatbot was **ELIZA**. ELIZA was developed in 1966 at the MIT Artificial Intelligent Laboratory by Joseph Weizenbaum. It simulates conversations based on hand-craft scripts that mimic a Rogerian psychotherapist. When a user chats with ELIZA, he or she types some statements in natural language. ELIZA analyzes the input text and looks for the presence of a keyword, and then generates responses according to a rule associated with the keyword. While it is able to engage in conversations, its communication ability and knowledge scope are limited. Below shows a typical conversation between ELIZA and a human.

User: Men are all alike.

ELIZA: In what way?

User: They're always bugging us about something or other. ELIZA:

Can you think of a specific example?

User: Well, my boyfriend made me come here.

ELIZA: Your boyfriend made you come here.

User: He says I'm depressed much of the time.

ELIZA: I am sorry to hear you are depressed.

Other well-known early chatbots include **PARRY** (developed by Colby in 1975) and **ALICE.** (developed by Wallace in 2009). Similar to ELIZA, they simulate how a human would behave in a text-based conversation using simple pattern-matching algorithms. PARRY is considered an improvement of ELIZA as it has a personality and a better controlling structure . The creation of ALICE was another step forward in the history of chatbots.It was the first online chatbot and was awarded for the best human-like system . ALICE pattern matches against 41,000 templates to generate responses, and in comparison, ELIZA contains only 200 keywords and rules.

In 2001, the development of **SmarterChild** marked an important advancement in chatbot technology. SmarterChild was available through messenger applications and was able to assist people with simple tasks using information retrieved from databases.

In 2003, a project called **CALO (Cognitive Assistant that Learns and Organizes)**, funded by Defense Advanced Research Projects Agency (DARPA) and coordinated by SIR International started. It was a five-year project that aimed to create a cognitive assistant that can learn from its experience and perform routine tasks for its users. The CALO project was very important in the history of chatbots as it integrated many areas of artificial intelligence and it helped software

systems to better understand the intentions of humans. Moreover, it had many spin-offs, most notably Apple Siri.

Apple Siri, created in 2010, was the first **virtual personal assistant**. It was followed by other assistants such as IBM Watson, Microsoft Cortana , Amazon Alexa , and Google Assistant. The virtual personal assistants were integrated into smartphones or smart speakers and could understand human speech, respond via voices, and handle more advanced tasks. Unlike earlier systems, virtual personal assistants are connected to the internet and are thus able to generate responses very quickly. However, misunderstandings happen very often as they cannot understand colloquial languages and cannot interpret the input within the dialogue context

.

Another chatbot worth our attention is **Microsoft's XiaoIce**. It is a social chatbot, and it demonstrates that conversational agents can not only conduct conversations and perform simple tasks but also satisfy our need for sociability.

More recently, there was another breakthrough in chatbot technology with the advancement of **open-domain chatbots**. Google's Meena and Facebook's Blender are dialogue systems that can chat about virtually anything and can achieve close-to-human-level performance in many aspects.

2.2 Classification of Chatbots

Chatbots can be classified based on different parameters, including the service provided, the knowledge domain, the response generation method, the goal, and the permission, and the amount of human-aid.

Classification based on the **service provided** considers the sentimental proximity of the chatbot to the user, the amount of intimate interaction that takes place, and the task the chatbot is performing. Interpersonal chatbots provide services like booking services or searches in FAQ without being a companion of the user. Intrapersonal chatbots live in the personal domain of the user and are expected to understand the user like a human does. Inter-agent chatbots communicate with other chatbots to accomplish a task.

The **knowledge domain** refers to the knowledge the chatbots access or the amount of the data they are trained upon. Open domain chatbots can respond to questions from any domain, whereas closed domain chatbots only has knowledge in a particular domain.

Classification based on the **response generation** method takes into account the method of generating responses. A chatbot can be classified as a rule-based, retrieval-based, or generativebased chatbot, and we will discuss this in more detail later in the paper .

Classification based on the **goals** considers the primary goal a chatbot aims to achieve. Information chatbots provide the user with specific information stored in a fixed source. Chat-based or conversational chatbots are designed to hold a natural conversation with the user like another human being would do. Task-based chatbots perform a task based on the user's requests.

Chatbots can also be classified according to the **permissions** provided by the development platforms. Open-source platforms make their code available so that developers have control over most aspects of implementation. Closed platforms are of proprietary code and are typically offered by large companies.

Finally, depending on the **amount of human-aid**, chatbots can be classified into human-mediated or fully autonomous chatbots. A human-mediated chatbot uses human computation in at least one component. Fully automated chatbots are fast in processing information while human computation provides flexibility and can help fill the gaps caused by the limitations of algorithms.

The figure below summarizes the classification criteria we discussed above.

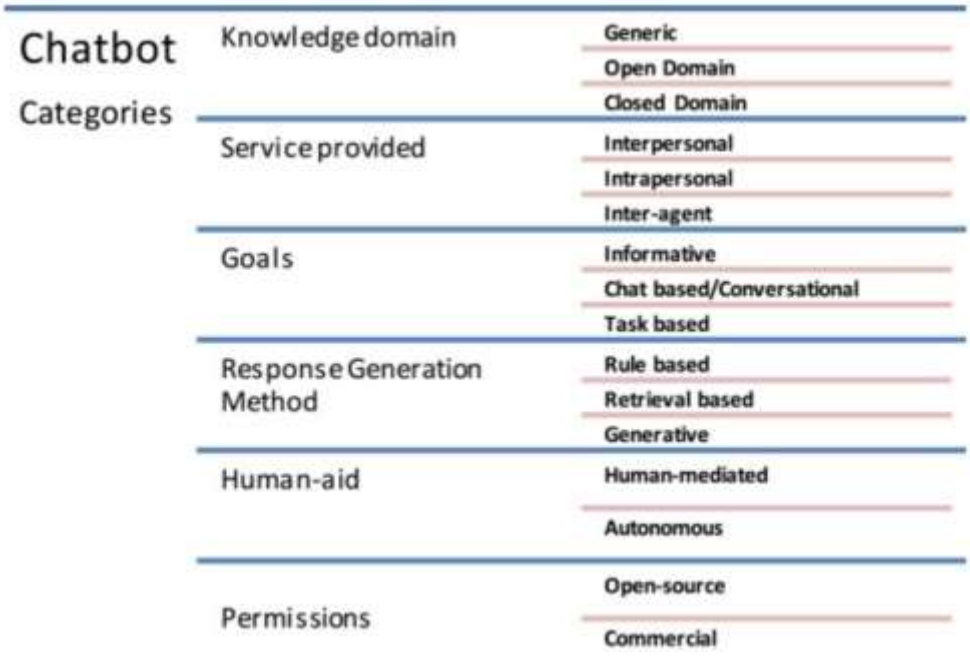

Figure 1: Classification of chatbots

3 Design Principles

There are two approaches that can be used to develop a chatbot depending on the algorithms and techniques adopted: rule-based approach and machine learning approach.

3.1 Rule-based

A rule-based chatbot processes information and provides responses based on a set of predefined rules with the use of **pattern matching** algorithms. Although the pattern matching techniques vary in complexity, the basic idea is the same. The user input is classified as a pattern, and the chatbot selects a predefined answer by matching the pattern with a set of stored responses. The pattern and response matching algorithms are handcrafted . Pattern matching is adopted by many chatbots and is especially popular among the early chatbots like ELIZA, PARRY, and ALICE. The advantage of the rule-based approach is its speed as it does not require any deep analysis of the input text [48]. However, the responses are repeated and lack flexibility and originality as the knowledge is set by the developer in advance . The following paragraphs will provide an overview of the three most commonly used languages for the implementation of rule-based chatbots.

Artificial Intelligence Mark-up Language (AIML) is a derivation of Extensible Mark-up Language (XML) . ALICE was the first chatbot implemented in the AIML language . AIML has a class of data object called an AIML objects, and these objects are responsible for modeling

conversational pattern. Each object consists of two units called topics and categories. A topic is an optional top-level element that has a name attribute and a set of categories associated with it. Categories are the most basic unit of knowledge and are the rules of the chatbot. Each category consists of two elements called pattern and template. The pattern matches against the input from the user to the template that contains the response of the chatbot . AIML is simple, flexible, and highly maintainable, and thus is one of the most commonly adopted languages for chatbot development. The biggest disadvantages of AIML are that the developer must specify a pattern for every possible input of the user and that it is inefficient when the knowledge base is large . The structure of an AIML object is demonstrated as below:

```
<category>
    <pattern>User Input</pattern>
        <template>
            Corresponding Response to Input
        </template>
</category>
```

RiveScript is a line-based scripting language that can be used to implement the Knowledge Base . Compared to AIML, RiveScript has more built-in features and tags, which means that the writer does not need to specify information about the chatbot in the additional configuration files
.

ChatScript is an open-source language for developing rule-based chatbots. It matches user inputs to outputs using rules created by human writers in program scripts through a process called dialog flow scripting . ChatScript uses concepts that are set of words that have the same meaning.

It consists of 2000 predefined concepts and scripters can also write their own concepts easily . Compared to AIML and RiveScript, ChatScript is a harder language, but it allows developers to combine rules in more complicated ways.

3.2 Machine Learning

The recent advancement in machine learning has made it possible to develop more intelligent chatbots. Chatbots that adopt machine learning approaches use machine learning algorithms to extract information and generate responses and are able to improve through previous conversations. An extensive training set is required for machine-learning-based chatbots . Two types of models can be used, retrieval or generative. Retrieval-based models involve choosing the optimal response from a set of responses , and generative models, on the other hand, use deep learning techniques to generate the response.

4 Architecture

In this section, we describe a general architectural design and delve deep into the important parts of each component. A general chatbot architecture consists of five main components, a User Interface component, a Natural Language Understanding (NLU) component, a Dialogue Management (DM) component, a Backend component, and a Response Generation (RG) Component , as illustrated in the figure below.

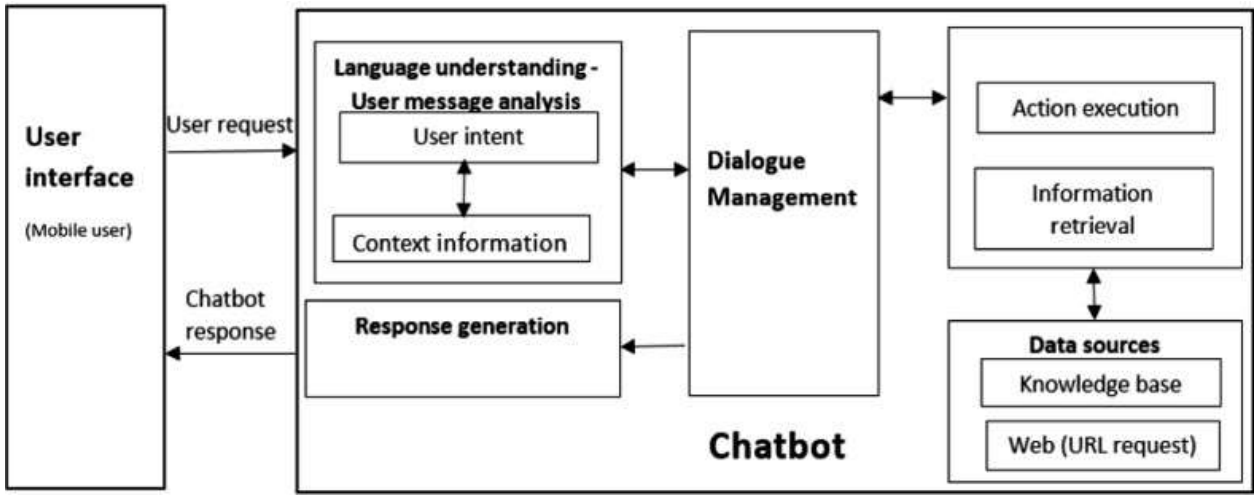

Figure 2: General chatbot architecture

4.1 User Interface

The user interface allows users to communicate and interact with a chatbot through messenger applications like Facebook Messenger, Cortana, or Slack. The operation of a chatbot begins with a user's request .

For speech-based conversational agents, an **automatic speech recognition** (ASR) system will first transform the user's input into text. The process can be modeled as a stochastic process and the desired output is the most probable sequence of words W that corresponds to the user's input X . Formally, this can be written as:

$$W^* = \underset{W}{\operatorname{argmax}}\, P(W|X)$$

$$= \underset{W}{\operatorname{argmax}} \frac{P(X|W)P(W)}{P(X)}$$

$$= \underset{W}{\operatorname{argmax}}\, P(X|W)P(W)$$ [67].

Here, *P(X|W)* represents the acoustic model and *P(W)* represents the language model. The acoustic model uses a corpus of phonetically transcribed and aligned data as training data. It creates the acoustic representation of the input, divided it into smaller frames, and then computes the likelihood of an input given a word. The language model contains information about which words are more probable in a given sequence. Finally, the output word sequence is determined in a process called decoding .

In addition, a speech-based conversational agent also has a **text-to-speech** (TTS) system that converts text back into speech after a text response is generated. TTS consists of two stages, text analysis, and waveform synthesis. Text analysis involves normalizing the text and performing phonetic (pronunciation) and prosodic (phrasing, pitch, loudness, tempo, and rhythm) analysis, and waveform synthesis involves selecting the prerecorded speech that meets the most requirements .

4.2 Natural Language Understanding

After the system receives the user's request, it uses the NLU component to extract information from the input and produce a representation of its meaning that can be used later on in the process . NLU generally deals with three tasks, dialogue act classification, intent classification, and slot filling.

Dialogue act classification deals with determining the function of the user's input, or more precisely, mapping the user's utterance to a dialogue act type. The utterance can be classified as a question, a statement, an offer, or some other type of dialogue act. Knowing the dialogue act being performed is critical to better comprehend the user's request and to decide an appropriate response .

Intent classification identifies the primary goal of the user. Intents are mainly domain-dependent. For example, a request can be in the domain of food ordering, hotel reservations, weather forecasts, and so on. The intent of an agent within the hotel reservations domain can be to book, cancel or change a reservation, and similarly, the intent of an agent within the food ordering domain can be to place, query, or change an order.

Slot filling is the final step in NLU. The agent extracts other necessary details, which when combined with the dialogue act and the intent, allow it to fully understand the user's request. Examples in the table below illustrate the dialogue act classification and intent classification processes described above.

Input	**Dialogue act**	**Intent**
Set a timer for 10 minutes.	A request for action	Set a timer
Is there a direct flight from Philadelphia to Beijing?	A question	Look up flight information
I'll be arriving at the hotel by 10pm tomorrow.	A statement	Check a hotel reservation

Table 1: Examples of dialogue act and intent classification

4.2.1 Slot Filling

Slot filling is the core of NLU and has received the most research attention. The main task of slot filling is to extract information from the input and to better understand its meaning. More specifically, slot filling aims to analyze the context of the input by separating the text into smaller units and then assigning the units tags or labels based on their functions and roles in the text. In this subsection, we will describe several different methods that are commonly used in slot filling.

Regular expression is a tool for describing text patterns. Each pattern is specified by a sequence of characters and each character is either a regular character or a metacharacter that has a special meaning.

Tokenization involves breaking up a text into units by words, punctuation marks, or numbers. Generally, English words are separated by white space, so tokenizing an English text should be fairly straightforward. However, there are some problematic cases where the boundaries of words or sentences are ambiguous. Contracted items (e.g. isn't), phrases (e.g. San Francisco), abbreviations (e.g. PhD.), and acronyms (e.g. AT&T) are examples of the special cases. One commonly used tokenization standard is known as the Penn Treebank tokenization standard. According to the Penn Treebank tokenization standard, contractions and punctuation between tokens are separated out and hyphenated words are kept together. A more advanced way to tokenizing text is to use data to determine the tokens automatically, and it is especially useful in dealing with unknown words. For example, bigger can be recognized even if only the words "big", "small", and "smaller" are contained in the training corpus. Usually, this is done by first inducing a vocabulary from a row training corpus and then segmenting the text into the tokens in the vocabulary . The vocabulary is composed of subwords, for example, "small" and "er" instead of "smaller". The three most widely used algorithms are: byte-pair encoding , unigram language modeling , and WordPiece.

Text normalization refers to converting the text into a standard form. Word normalization is the task of identifying words that have the same meaning but spelled differently (e.g. U.S.A. and USA). Case folding is the task of transforming everything to lower cases. Lemmatization is the process of identifying the roots of the words and mapping the morphological variants into their base form (e.g. produce, product, produces, production → produce).

Bag of words, also known as the **vector space model**, is one of the simplest approaches to analyze the input. We count the occurrences of each word but ignore the syntactic information or word order information. This can be done by performing text normalization and eliminating stop words, i.e. words that do not contribute to the meaning of the sentence. The bag-of-words approach is limited when the meaning of the text is subject to linguistic knowledge such as grammar or word order. For example, "Alice is taller than Bob" and "Bob is taller than Alice" yield the same representation but their meanings are different.

Latent Semantic Analysis (LSA), or **Latent Semantic Indexing**, also does not take any linguistic knowledge into account. However, LSA compares the meaning behind the words instead of the actual words, and terms that occur frequently in the same context are grouped together. In order to determine word similarities, we create a matrix where each row represents a term and each column represents a document, and each cell in the matrix represents the frequency of the word in the document. Singular Value Decomposition is then applied to transform the matrix and reduce its dimensionality .

Part-of-speech (POS) tagging is the process of marking up each word in the text with a tag that indicates its syntactic role, for example, whether it is a verb, noun, pronoun, conjunction, and so on. POS tagging is especially useful in cases where we encounter words that have different meanings and their use in the sentence may be ambiguous. For example, the word "ring" can be either a noun or a verb, but in the sentence "ring a bell," it should be categorized as a verb, and using a POS tag in cases like this can clear up the confusion. The figure below illustrates the task of POS tagging.

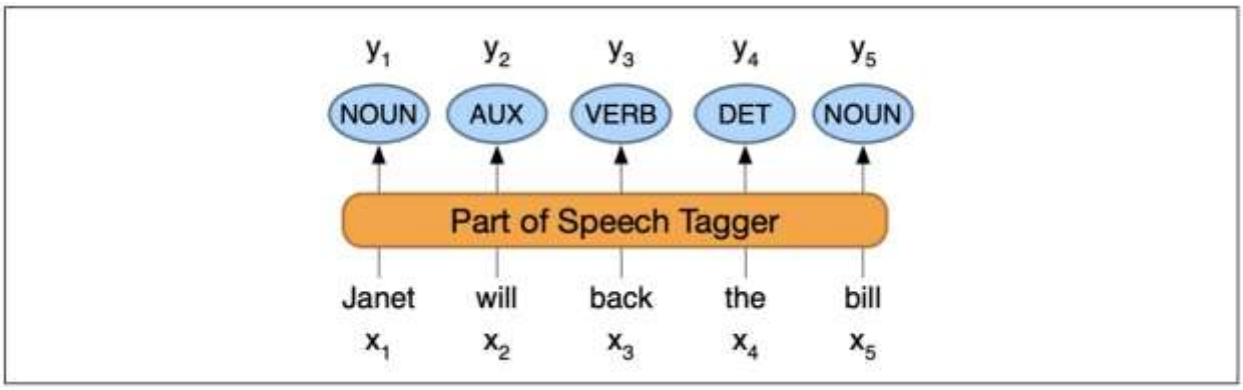

Figure 3: Part-of-speech tagging: mapping from words x_1, x_2,…,x_n to POS tags y_1, y_2,…,y_n **Name entity recognition** (NER) involves extracting items such as persons, dates, and organizations. POS tagging and NER are disambiguation tasks and provide useful information about sentence structure and meaning. NER is often more complicated than POS tagging because it involves determining the appropriate segmentation. A number of datasets exist for POS tagging and NER. The Universal Dependencies and the Penn Treebanks have POS tagged corpora in different languages, and OntoNotes has named entity tagged also in different languages.

Semantic role labeling deals with assigning the semantic roles to the arguments of a verb [30]. For example, in the sentence "book a flight to Philadelphia on Monday," "book" is the verb, and its object or theme is "a flight," "to Philadelphia" is a location, and "on Friday" is a date. One use case of semantic role labeling within the context of chatbot technology is to match the semantic structures in the input against similar structures in a collection of documents to find a potential response .

Once we extract the relevant parts from the user's input, a complete analysis of the utterance can be performed with the use of grammars. The utterance can be interpreted directly with semantic grammar, or it can be parsed syntactically before the semantic analysis is applied. The semantic grammar approach uses grammar rules where the words are classified based on their semantic roles. In syntax-driven semantic analysis, the units of syntactic analysis are mapped to units of semantic analysis using one of the two grammar formalisms, context-free grammar and dependency grammar.

4.2.2 *Statistical Approaches to NLU*

Traditionally, the grammar rules are handcrafted, but over the past decades, more research has focused on assigning the labels automatically. In this section, we are going to look at how statistical models are used in the labeling problems, the process of assigning a label to each word in a text.

The **Hidden Markov Model (HMM)** is based on augmenting the Markov chain model. A Markov chain is a model that describes the probability of sequences of possible events, with the assumption that the future depends only on the state attained in the current event. An HMM is a Markov model where the states are not fully observable, and it is useful because the labels in a text are not observable. In the context of NLU, an HMM is a probabilistic sequence model that takes a sequence of words $w_1 \ldots w_n$ as the input, computes the probability distribution over possible label sequences, and subsequently outputs the best sequence $t_1 \ldots t_n$. An HMM has the following component: N label types or states, an initial distribution over states, a sequence of words or observations, a transition probability matrix representing the probability of transitioning from one state to another, and emission probabilities describing the probability of an observation being generated from a state. To put this into context, the transition probability represents the likelihood of a label occurring given the previous label, and the emission probability represents the likelihood that a label is associated with a word when the label is given [50]. Formally, we capture the goal of HMM labeling in the equation below:

$$\hat{t_{1,\ldots,n}} = \underset{t_1 \ldots t_n}{\operatorname{argmax}} P(t_1 \ldots t_n | w_1 \ldots w_n)$$

HMM makes two assumptions. The first assumption is the Markov assumption, which says that the probability of a particular label depends only on the previous label. The second is that the probability of a word appearing depends only on its label and not on any other words or any other labels. Applying the two assumptions and the Bayes' rule, we can rewrite the equation as:

$$\hat{t}_{1,\ldots,n} = \underset{t_1 \ldots t_n}{\operatorname{argmax}} \frac{P(w_1 \ldots w_n | t_1 \ldots t_n) P(t_1 \ldots t_n)}{P(w_1 \ldots w_n)}$$

$$= \underset{t_1 \ldots t_n}{\operatorname{argmax}} P(w_1 \ldots w_n | t_1 \ldots t_n) P(t_1 \ldots t_n)$$

$$\approx \underset{t_1 \ldots t_n}{\operatorname{argmax}} \prod_{i=1}^{n} P(w_i | t_i) P(t_i | t_{i-1}),$$

which corresponds to the products of the emission and transition probabilities. The Viterbi algorithm can be used to find the solution, and the forward-backward or Baum-Welch algorithm can be used to train the model .

Figure 4 illustrates the transition probabilities A and the emission probabilities B for three labels VB, MD, and NN in an HMM model.

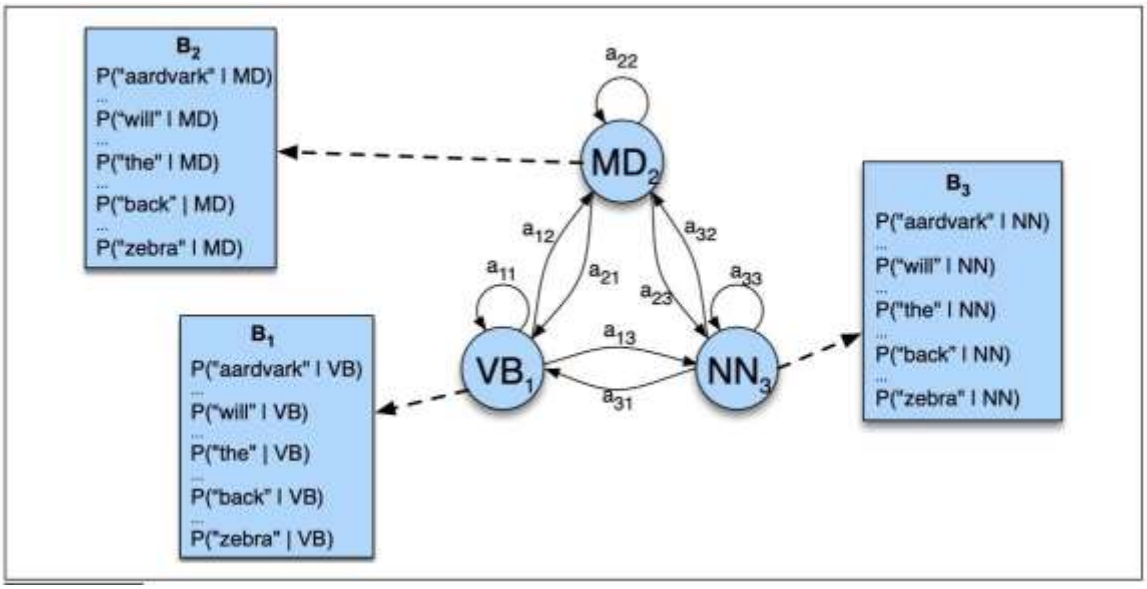

Figure 4: Parts of an HMM representation

Stochastic Finite State Transducers (SFST) considers NLU as a translation process where stochastic language models are implemented using Finite State Machines (FSM). The FSMs take a sequence of words *W* as input and output a sequence of labels *T*, and they can be either handcrafted or learned from an annotated corpus. Each elementary label has an FSM, and the transducers are grouped together as one transducer. A stochastic language model is computed as the joint probability *P(W, T)*:

$$P(W,T) = \prod_{ki=1} P(w_i t_i | w_{i-1} t_{i-1} \ldots w_1 t_1)$$

The model can be further improved by replacing some categories of words with tags, such as country, year, etc.

Conditional Random Fields (CRF) Models are discriminative sequence models based on loglinear models. Unlike HMM, CRF models can incorporate arbitrary features like word shape and prefix. The feature functions are the key to the CRF models. They help us to handle unknown words by incorporating adding features like word shape and morphology. (Words that are capitalized are more likely to be nouns and words that end with '-ed' are usually past tense.) A CRF takes in the input sequence and outputs a probability to the label sequence out of all possible sequences. It computes the posterior probability directly, and at each time step it computes loglinear function over a set of features, and then finds the global probability by aggregating and normalizing the local features. Suppose we have *K* features, and each feature F_k is the property of the entire input sequence and the entire output sequence and is composed of a sum of local features f_k and has a weight $\square_k$. We write the desired output as:

$$p(T|W) = \frac{\exp\left(\sum_{k=1}^{K} w_k F_k(W,T)\right)}{}$$

$$\frac{\exp\left(\sum_{k=1}^{K} w_k F_k(W, T')\right)}{\sum_{T'} \exp\left(\sum_{k=1}^{K} w_k F_k(W, T')\right)}$$

$$= \sum \frac{\exp\left(\sum_{Kk=1} \lambda_k \sum_{ni=1} f_k(t_{i-1}, t_i, W, i)\right)}{\exp\left(\sum_{k=1}^{K} \lambda_k \sum_{ni=1} f_k(t_{i-1}, t_i, W, i)\right)}$$

Note that we constraint each local feature f_k to be associated with only the current and previous words, and this limitation allows us to train a CRF model in a similar way as we train an HMM [50].

Other statistical models include **Support Vector Machines Models** , **Dynamic Bayesian Networks Models** , the **Hidden Vector State Models** .

4.3 Dialogue Management

The Dialogue Management component deals with information coming from the other components and is responsible for controlling and updating the context of conversations and governing the actions of the chatbot.

4.3.1 Design Issues

As mentioned, DM coordinates other modules and encapsulates the logic of the speech application. This means that it is a very important component in the chatbot framework, and we need to make deliberate and thoughtful design decisions.

Unfortunately, the process of designing a robust DM strategy is far from trivial as there it is difficult to foresee which form of system behavior will lead to high user satisfaction. In this subsection, we discuss two frequently arising design issues of DM, interaction strategies and the choice of a confirmation strategy.

The interaction strategy of a dialogue system dictates who has control over the conversation. The conversation can be user-directed, system-directed, or mixed-initiative. When the conversation is user-directed, the user takes the initiative and the system just answers the user's queries and commands. When the conversation is system-directed, the system has the initiative and the user simply responds to the system's queries. When the dialogue is mixed-initiative, both the user and the system can take control.

The DM keeps track of the state of the conversation, or the conversational behavior of the system, which is based on the interactive strategy it adopts. The table below summarizes the different states that a conversation can be in.

State	Conversation control	Description
Grounded	System	Acknowledging the user's input while deciding on the agent's actions
Slot Filling	System	Requesting extra information form the user to resolve actions

Initiative	System / User	Steering of the conversation by either the user or the agent
Context Switch	User	Change of the basis or premise of the conversation

Table 2: States in Dialogue Management

In addition, the DM component typically includes an error handling module that copes with the uncertainty and ambiguity issues with the information it receives. The ASR and SLU modules are not perfect, and one way to alleviate errors is to ask the user for confirmation when necessary. Typically, confidence scores are assigned to the ASR and SLU results, and results with scores below a certain threshold need to be rejected or confirmed. There are two types of **confirmation strategies**, explicit confirmation, and implicit confirmation. When the explicit confirmation strategy is used, the system confirms its understanding by asking the user another question. The following conversation is an example of an explicit confirmation:

User: I want to know the hours of the closest CVS.

System: Do you want to go to the closest CVS?

User: Yes.

When the implicit confirmation strategy is employed, the system includes some of the information it gets in its response. Consider the following example:

User: I want to know the hours of the closest CVS.

System: When do you want to arrive at CVS ?

Note that it is the user's responsibility to correct the system if he or she thinks the system has misunderstood the request.

There are two types of DM systems, goal/task-oriented systems and non-task-oriented systems. The task-oriented systems are commonly used in agents designed for performing tasks such as making reservations, answering FAQs, and scheduling meetings. In order to complete the predefined tasks, they direct the dialogue from one state to another.

4.3.2 Handcrafted Approaches to DM

We will first go over some of the traditional approaches to DM.

The simplest form of a DM system is a sizeable **Switch Statement**. A Switch Statement is a programming control structure that triggers a predefined response for each possible input or intent. One disadvantage of this approach is that the user is always the one leading the conversation. Also, the actions of the chatbot depend on the predefined set of rules. It cannot engage in conversations or provide responses that are outside the scope.

Finite state-based DM uses finite state machines to track the flow or the states of the conversation [42]. This approach is simple and is especially suitable when there is a limited set of possible inputs. Its disadvantage is that it lacks flexibility and adaptability. Conversations that are working toward the same goal have to go through the same process even though not all states are relevant.

Frame-based DMs use a frame structure to keep track of all information the system needs from the user. One advantage of this approach is that several data can be captured at the same time and

the information can be gathered in any order. Also, frame-based DMs believe humans may change their minds during conversations, so they treat utterances as dialog acts in which the users communicate their goals.

Some of the other more advanced approaches include **Information State Theory** , **plan-based theory** , and **conversational game theory**.

4.3.3 Statistical Approaches to DM

Handcrafting DM strategies is time-consuming and requires a lot of effort, so the research focus has now shifted toward machine learning approaches to DM strategy design. The main idea of these machine learning approaches is to learn optimal strategies using data collected from live use of the system. There are three machine learning approaches: reinforcement learning-based, corpus-based, and example-based DM. Example-based approaches can be considered a specific case of corpus-based approaches. In this subsection, we are going to review reinforcement learning-based and corpus-based approaches.

Reinforcement learning-based approaches model a dialogue strategy as a Markov Decision Process (MDP). In the context of a dialogue system, at each step, the system is in some dialogue state *s*, and it must choose an action *a* from a set of system dialogue actions. The system moves into a new state *s'* according to a transition probability *P(s'|s, a)* which describes the probability of moving from *s* to *s'* after action *a* is performed in state *s*. The transition is associated with a reward *r*, which evaluates the result of performing action *a* in state *s*. Examples of rewards include task completion and dialogue length. A strategy $\pi(s) = P(a|s)$ or $\pi(s) = a$ if the policy is deterministic specifies the action of the system in each state, and the goal is to learn the best strategy π^* that maximizes the overall reward.

The process can be solved by two approaches, model-based approach, and simulation-based approach. The model-based approach first estimates the transition probabilities from a corpus of real human-computer dialog data and then finds the solution using dynamic programming. This approach is also known as "partial strategies" since it can only learn the decisions in some states and the actions for the other states need to be specified manually. Only in some very rare cases where the corpus contains exploratory data for every state can this approach be used to learn the full strategies. The simulation-based approach learns from sample returns. In this approach, the strategy is learned as the DM interacts with a user simulation. A user simulation is a model that simulates user responses based on a dialogue corpus. The effectiveness of this approach is highly dependent on the simulation's accuracy and ability to generalize unseen dialogue situations. This is a very challenging task, but if such a simulation exists, then the simulation-based approach can explore the state space much more exhaustively, and the "full strategies" can be learned as a result.

An extension of the MDP model is a partially observable MDP (POMDP), and it is used when not all states are directly observable. At each step, the system is in an unobserved state *s*, and it transits to another unobserved state *s'* after performing action *a*. Although the exact state cannot be uniquely identified, the system receives an observation *o'*, which provides some evidence about the new state. The belief state *b(s)* describes the probability of being in state *s* .

The system chooses an action based on the belief state, and the belief state is updated after the observation is received. The process can be described as follows:

$$'(s') = P(s'|o', a,b) = \frac{P(o'|sP',(ao,'b|)aP, b(s)'|a, b)\ b}{}$$

$$= \frac{P(o'|s', a, b)\ \sum_{s\in S} P(s'|a, b, s)P(s|a, b)}{} P(o'|a, b)$$

$$= k \cdot P(o'|s', a)\sum_{s\in S} P(s'|a,s)b(s)\ ,$$

where

$k = P(o'|a, b)$ is a normalization constant.

At each time, there is a reward associated with the chosen action, and the accumulated reward throughout the conversation can be found using the equation below:

$$R = \sum_{t=0}^{\infty} \lambda^t R_{(b_t, a_t)} = \sum_{t=0}^{\infty} \lambda^t \sum_{s\in S} b_t(s) r(s, a_t)\ \text{[108]}.$$

As in the case of MDP, the goal of the POMDP model is to find the strategy that maximizes the total reward at every point b. However, finding the optimal strategy is very expensive computationally and cannot be done in practice. As a result, POMDPs are not commonly used even though they are theoretically appealing for DM.

A **corpus-based approach** to DM generates the next response through a classification process using the complete dialogue history. The DM represents conversations as a sequence of (A_i, U_i) pairs, where A_i is the system's response and U_i is the semantic representation of the user's input at time i . At each moment, the goal is to find the optimal response out of all the possible responses given the history of the conversation, which is denoted as:

$$\hat{A}_i = \underset{A_i\in A}{\text{argmax}}\ P(A_i|(A_1, U_1),\dots, (A_i, U_{i-1})).$$

As described by the equation above, the DM selects the next answer taking the entire dialogue history into account. The biggest problem is that the equation is hard to solve when there are many possible sequences of states. To overcome this difficulty, a data structure called dialog register (DR) is defined to keep track of the information provided by the user throughout the dialog. After introducing the new data structure, we say the output depends on the DR that results from turn 1 to turn i-1 and the previous state. We explicitly consider the previous state since a user turn supplies both task-dependent and task-independent information and are more informative than the DR. Formally, we can then rewrite the above equation as:

$$\hat{A}_i = \underset{A_i\in A}{\text{argmax}}\ P(A_i|DR_{i-1}(A_{i-1}, U_{i-1}))\ \text{[33]}.$$

The problem can be solved using a classification process. At each moment, the objective of the DM is to choose a class from a set of classes, and then provide an answer based on the selected class. The classification function is defined using a multiplayer perceptron , where the input layer represents the input pair, and the output layer represents the probability of the input belonging to

the associated class . As mentioned before, the DR contains information about the labels and values for the user's inputs. However, it is assumed that the exact values of these attributes are not needed to predict the next action of the system. What is relevant is whether they are present or not. Consequently, DR codifies each slot based on whether the concept and the attribute are present or not and the confidence measures of the values.

4.4 Backend

Chatbots retrieve information that is needed for performing the required tasks from the Backend and then forward the message to the Dialogue Management Component and Response Generation Component .

Rule-based chatbots require a **Knowledge Base (KB)** to store the handcrafted rules. The rules in the Knowledge Base should be as diverse and as holistic as possible to ensure the robustness of the chatbot . A chatbot can also use a Relationship Data Base (RDB) to recall past conversations. Taking the previous information into account allows the chatbot to communicate in a more consistent, precise, and reliable manner.

The development of the KB is necessary and essential but can be very time-consuming and demanding as it involves manual work. To overcome this difficulty, developers came up with a way that can build a new KB automatically from the KB of an existing chatbot. There's also a program that can transform a corpus to an AIML KB . Many times, rule-based chatbots guide the users and complete the KB using the users' responses.

Recently, there has been a trend toward keeping the information in digitally stored texts on the World Wide Web or in other online sources. These large repositories store information in a machine-readable and accessible form. Examples of these knowledge bases include Google's Knowledge Graph , DBpedia, Freebase, and Wolfram Alpha. Knowledge Graph was introduced in 2012. It provides structured information about a topic or a summary using crowd-sourced and manually curated data. DBpedia data can be accessed online, and the source of its information is structured data from Wikipedia.

Typically, these repositories consist of subject-predicate-object triples and are represented by a graphical structure. The nodes indicate the entities, or the subjects and objects, and the edges indicate the relationships between the entities. The figure below is a subgraph of Freebase related to the TV show *Family Guy*. Note that the nodes include names and dates which are connected using directed edges that specify their relationships.

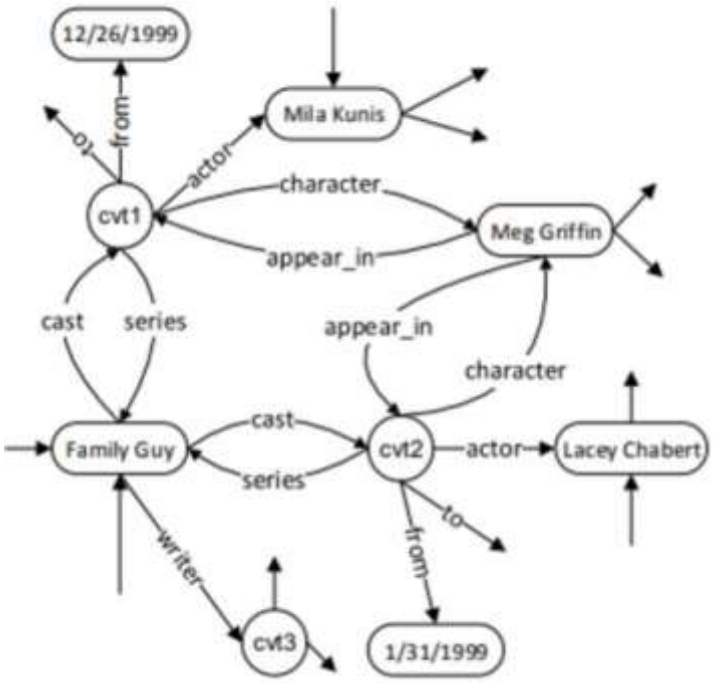

Figure 5: Subgraph of Freebase

4.5 Response Generation

Once the appropriate information has been retrieved, the next step for the dialogue system is to determine the content of the response and the best way to express it. The Response Generation component is responding for generating responses in user understandable format.

4.5.1 The RG pipeline

RG involves five stages of processing: signal analysis, data interpretation, document planning, microplanning, and realization.

Signal analysis and **data interpretation** are preliminary stages. Input that is in the form of data, such as numerical data in weather predictions, goes through these two stages. The signal analysis identifies patterns and trends in data using pattern matching algorithms. Data interpretation uses symbolic reasoning methods based on domain knowledge to analyze complex messages and relations between the messages.

Document planning can be further composed into two substages: content determination and discourse planning. Content determination involves deciding what information needs to be delivered to the user. The system only communicates some of the information retrieved, and it may communicate the information in stages. When necessary, content determination also involves filtering and summarizing the information. Discourse planning is concerned with organizing and structuring the text to aid human comprehension.

Microplanning, or sentence planning, involves coming up with the sentences that delivery the information. It consists of three tasks: referring expressions, aggregation, and lexical selection.

Dealing with referring expression involves choosing the appropriate way to address an entity given a particular context. For example, an entity should be referred with a pronoun if it has already been

mentioned earlier in the text. Aggregation is the task of joining content with the use of conjunctions, ellipsis, etc. Lexical selection is the issue of looking for the appropriate words for the text.

Realization deals with linguistic knowledge of the content, for example, to choose the correct tenses, enforce the subject-verb agreement, and apply word ordering rules, and so on.

4.5.2 Statistical Approaches to RG

In this subsection, we are going to review some commonly used language models, models that assign probabilities to sequences of words.

N-gram is the simplest language model. Formally, we write the desired output of assigning probabilities to sequences of *n* words as:

$$P(w_1, \dots, w_n)$$

We apply the chain rule and the Markov assumption that the current word depends only on the previous *n*-1 words. For example, a bigram model approximates the probability of a word using only the conditional probability of the preceding word. We can rewrite the expression above for the bigram model as:

$$\prod_{k=1}^{n} P(w_k|w_{1:k-1}) = \prod_{k=1}^{n} P(w_k|w_{k-1}).$$

The probabilities are estimated using the maximum likelihood estimation. To get the estimate, we estimate the bigram probability by counting the number of occurrences w_1w_2 and then normalizing it by the sum of all bigrams that start with the same word as follows:

$$p(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)} = \frac{C(w_{n-1}w_n)}{C(w_{n-1})}$$

In practice, trigram, 4-gram, or even 5-gram models are used if sufficient training data is available. To use these models, extra contexts are appended at the beginning and the end of sentences .

A **Recurrent Neural Network (RNN)** is a network that contains a cycle in the network connections, which means that previous outputs can be used as inputs . It overcomes a major drawback in N-gram models that the context from which information can be extracted is limited. RNN-based language models take sequences of words as input and predict the next word in a sequence. At each step, the model first computes a hidden layer using the word embedding, or vector representation of the current word and the previous hidden layer. The new hidden layer then passes through a softmax layer to generate a probability distribution over the entire vocabulary.

More formally, we describe the process at time *t* as follows:

$$e_t = E^T x_t$$

$$h_t = g(Uh_{t-1} + We_t)$$

$$y_t = \text{softmax}(Vh_t),$$

where *x* denotes the input, *y* the output, *E* the word embedding matrix, and *U*, *V*, and *W* the weight matrices. Concretely, the input sequence consists of word embeddings and the word embedding matrix is used to retrieve the embedding for the current word. Then, the sum of the embedding

multiplied by W and the previous hidden layer multiplied by U is passed through an activation function g to find the activation value for the current hidden layer h_t. Vh represents a set of scores over the possible words given the information provided in h, and the scores are normalized into a probability distribution through the softmax. Using the probability distribution, we can find the probability of a particular word being the next word in the sequence as follows:

$$P(w_{t+1} = i | w_{1:t}) = y_{ti},$$

and the probability of the word sequence is simply the product of these individual word probabilities:

$$P(w_{1:n}) = \prod_{i=1}^{n} P(w_i | w_{1:i-1}) = \prod_{i=1}^{n} y_{w_i i}.$$

The RNN is trained on a corpus of text using a regimen called teacher forcing, which minimizes the difference between the predicted and actual probability distributions. However, in practice, RRNs are hard to train because they cannot carry forward distant information and because of the so-called vanishing gradients problem during the backpropagation step.

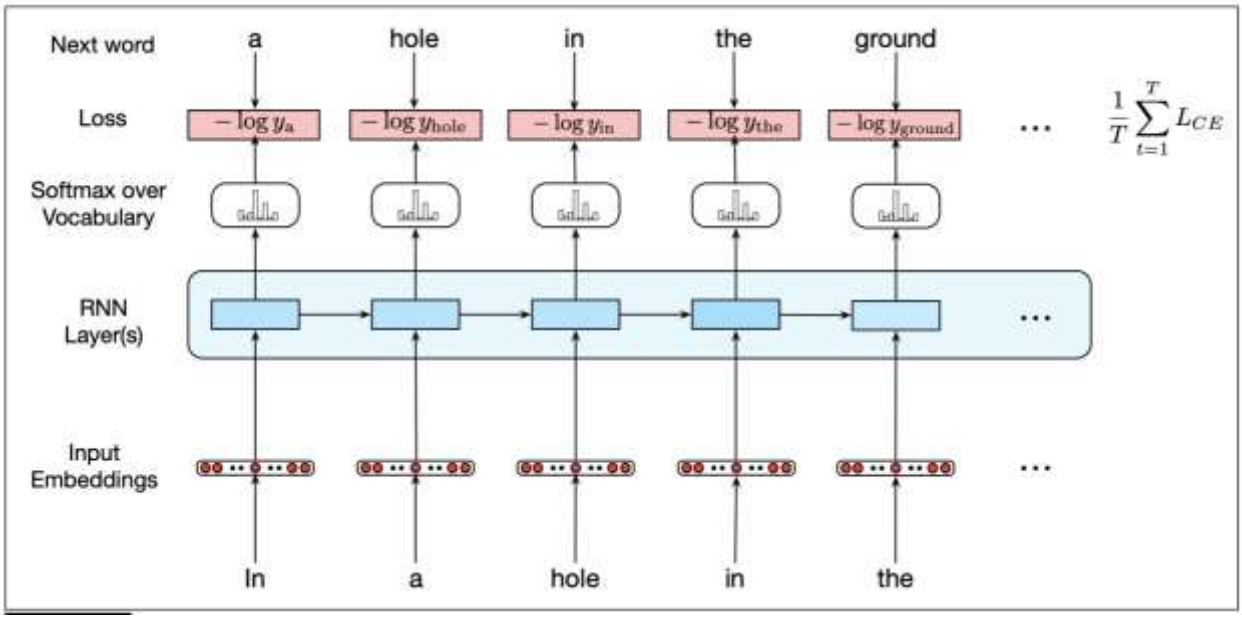

Figure 6: Training RNNs as language models

Long Short Term Memory (LSTM) networks is a variant of RNN that was introduced to overcome the difficulties of RNN. An LSTM network manages the context by removing irrelevant information and adding information likely to be useful in the future. This is achieved by explicitly adding a context layer and specialized units called gates to control the information flow. Three gates are introduced into the architecture, the forget gate, the add gate, and the output gate. The forget gate is responsible for getting rid of information that is no longer useful, the add gate selects new information that is needed for the current context, and the output gate determines whether a piece of information is required for the current hidden state. Each gate is composed of a feedforward layer, a sigmoid activation function, and a pointwise multiplication with the layer being gated .

Transformers map input sequences to output sequences with a similar approach to RNN. At each step, given an input sequence, the Transformer produces an output distribution over all possible words. However, it does so without the recurrent connections with the use of additional layers called self-attention. Self-attention is the key to transformers. Information can be directedly processed without going through intermediate recurrent connections. A self-attention layer takes word sequences as input and output sequences of the same length. At each step, it has access to the current input and all previous inputs. An output is computed using the relevance of input in a given context, which can be found by comparing the input with some relevant items. More formally, the result of the comparisons is referred to as scores, and the output is the weighted sum of normalized scores:

$$y_i = \sum_{j \le i} \frac{\exp\left(score(x_i, x_j)\right)}{\sum_{k=1}^{i} \exp(score(x_i, x_k))} x_j \, .$$

Also, additional parameters are included in the transformers to provide opportunities for learning. Each parameter is a set of weight matrices that are used to operate over the inputs. The selfattention layers, along with some other components like feedforward layers, residual connections, and normalizing layers form a transformer block as illustrated in the figure below. A transformer is made of stacks of transformer blocks .

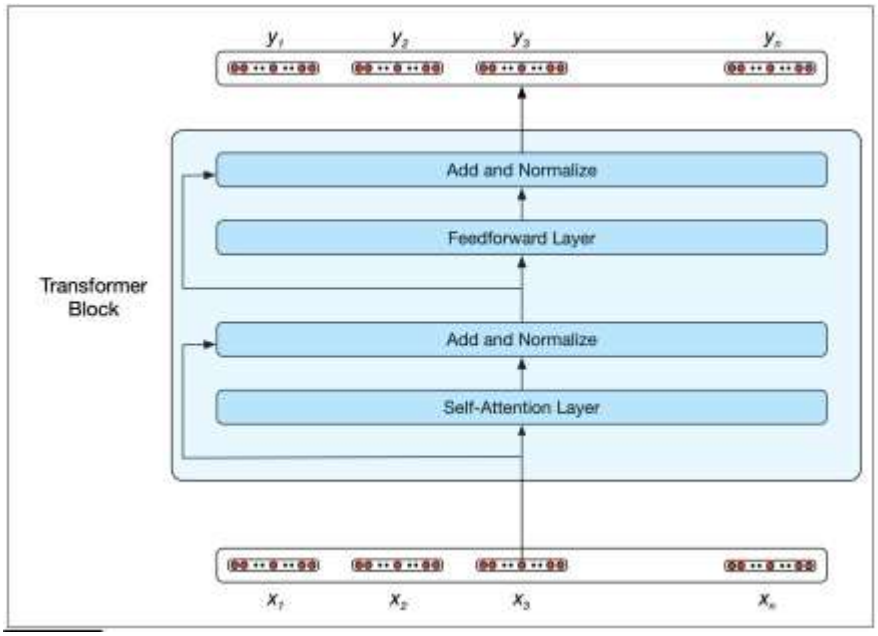

Figure 7: A transformer block

An extension of the transformer model uses multihead self-attention layers. Each multihead selfattention layer is a set of self-attention layers and has its distinct set of parameters. This characteristic makes the extension very useful when words in the sentences are related to each other in different ways. Another extension combines the input with positional embedding so that the input incorporates information about the order .

Encoder-decoder networks or **sequence-to-sequence** networks area special class of RNN capable of generating arbitrary-length output sequences within the context. An encoder-decoder network has three components, an encoder, a context vector, and a decoder. The encoder converts the input sequence into a contextualized representation called the context, and then the decoder accepts the context as input to generate an output sequence. Both the encoder and decoder are either LSTM or Transformer models. The first hidden state of the decoder is initialized to the final state of the encoder, and each other hidden state takes the previous hidden state, the output generated in the previous state, and the context vector as input. More formally, this process can be illustrated with the equation below:

$$h_{td} = g(\hat{y}_{t-1}, h_{td-1}, c),$$

where c denotes the context vector, and the superscripts e and d are used to distinguish the hidden states of the encoder and the decoder. The full equations for the decoder are then:

$$c = h_n^e$$

$$h_0^d = c$$

$$h_t^d = g(\hat{y}_{t-1}, h_t^d{}_{-1}, c)$$

$$z_t = f(h_t^d)$$

$$y_t = softmax(z_t).$$

We then use the output y to find the most likely output at each step .

The figure below illustrates the entire process of the model.

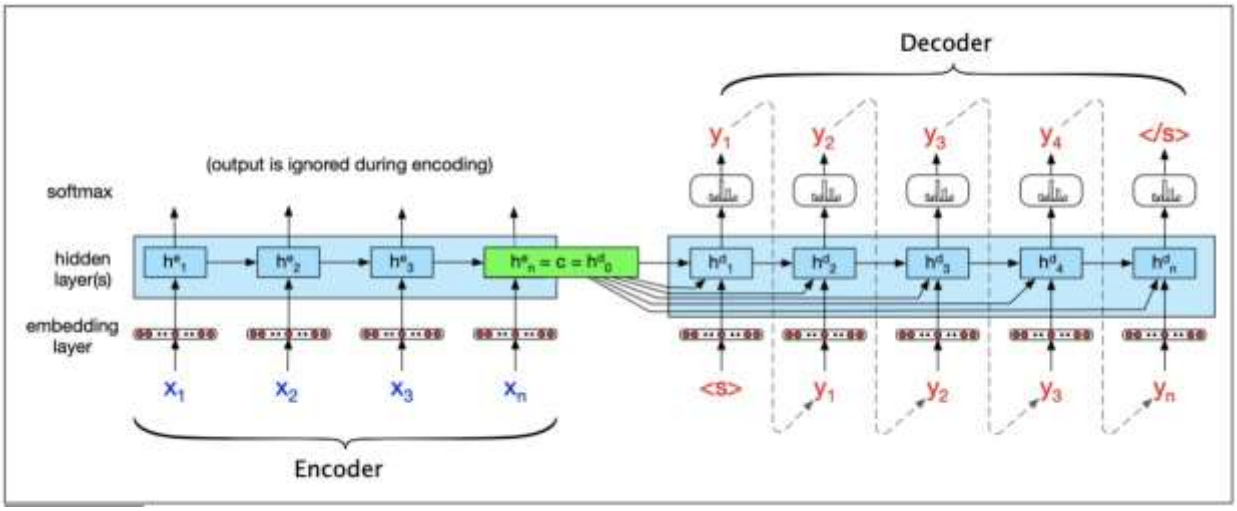

Figure 8: The basic RNN-based encoder-decoder architecture

Encoder-decoder architectures are trained on tuples of paired strings, each consists of a source and a target .