St. Xavier’s College

Autonomous

Kolkata, WB

**Generative Chatbot using Transformer Models**

By

Arighna Chakraborty

Shreyashi Choudhury

Tiyasa Chatterjee

Under Guidance of

Dr. Asoke Nath

**Abstract**

Dialogue generation or development of intelligent conversational chatbots using Artificial Intelligence or Deep Learning techniques is an interesting problem in the field of Natural Language Processing. Conversational chatbots are predominantly used by businesses, government organizations and financial organizations. There are many frameworks and existing chatbots available, but they generally lack the flexibility in developing real-world conversations. The functionality of these bots are limited since most of them are retrieval based (are used in closed domain scenarios and rely on a collection of predefined responses), and also they are not aimed at holding conversations that emulate real human interaction.

This project aims to develop and implement a simple *Generative Chatbot* using *Transformer* models.*Generative chatbots* are *open domain* systems that generate original combinations of language rather than selecting responses from predefined data. The new combinations of responses are generated using various machine learning and deep learning techniques feeding upon a lot of historic data and previous conversations. The *Transformer* model is a network architecture , based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train.

**Acknowledgement**

We are thankful to our project mentor, Dr. Asoke Nath, for guiding us throughout the process of documenting our project work and pointing us in the right direction. We thank the college authorities and our Head Of Department Prof. Romit Beed, for facilitating project work even in the midst of the COVID-19 pandemic.

We profusely thank previous and current researchers in the field of Artificial Intelligence, and more specifically, Natural Language Processing, without whose contributions our own work would not have been possible.

**Table of Contents**

1. Introduction
   1. Background
   2. Purpose and Scope
   3. Application
2. Survey of Technologies
   1. Types of Chatbots
   2. Underlying Concepts
      1. Recurrent Neural Networks
      2. Long Short Term Memory Networks
      3. Sequence to Sequence Models
      4. Transformer Models
3. \* Requirements and Analysis
   1. Challenges
   2. Dependencies
   3. Hardware Requirements
4. \* System Design
   1. Data Preprocessing
   2. Basic Modules
      1. Embedding Module
      2. Positional Encoding Module
      3. Encoder Stack
      4. Decoder Stack
   3. Flow Chart
5. References

**\*** Topics and its contents are subject to changes depending on the progress and the challenges faced during the actual implementation and testing of our project.

**Chapter 1**

**Introduction**

* 1. **Background**

Natural Languages are those that came into existence for the purpose of being used by humans in order to interact with one another. Such languages have evolved alongside humans, without any conscious planning. Natural Language Processing resides at the overlap between Linguistics, Computer Science and Artificial Intelligence and involves understanding, analyzing, processing and manipulating Natural Language data. Various NLP tasks include, machine translation, sentiment classification, language modeling etc.

Conversational AI combines natural language processing with machine learning. These NLP processes flow into a constant feedback loop with machine learning processes to continuously improve the AI algorithms. Conversational AI has principle components that allow it to process, understand, and generate response in a natural way.

* 1. **Purpose and Scope**

For a Conversational Chatbot to be able to hold a conversation with a certain amount of randomness in their responses, it is a must that the bot learns and understands human language. Chatbots can even be infused with various personality traits such as: empathy, humor, kindness, introversion/extroversion, cheerfulness etc. by selectively training them on specialized datasets and even by controlling the training of the bot using reinforcement learning where a reward point is given to the bot every time it generates a desirable output.

The field of conversational chatbots can prove to be a gateway for machines to understand humans and for us to replicate human thoughts and emotions into a machine. The conversational experience can further be enhanced by enabling text based sentiment analysis in a chatbot model which shall help the bot to better understand the sentiment and feelings behind a user input.

The **scope** of the chatbot built in this project covers the action of conducting casual, friendly conversation with its user. Learning to build a Conversational AI Chatbot takes machines one step closer to succeeding in the Turing Test. Such chatbot can further be developed to learn from ongoing conversations. The chatbot can be taught to speak on various different topics by using different kinds of datasets to train.

The **limitations** faced by such conversational bots include being prone to grammatical error and arbitrariness.

* 1. **Applications**

While an AI chatbot is the most popular form of conversational AI, there are still many other use cases across the enterprise. Some examples include:

* **Online customer support**:  Online chatbots are replacing human agents along the customer journey. They answer frequently asked questions (FAQs) around topics, like shipping, or provide personalized advice, cross-selling products or suggesting sizes for users, changing the way we think about customer engagement across websites and social media platforms and so on.
* **Health care**: Conversational AI can make health care services more accessible and affordable for patients, while also improving operational efficiency and the administrative process, such as claim processing, more streamlined.
* **Internet of things (IoT) devices**: Most households now have at least IoT device, from Alexa speakers to smart watches to their cell phones. These devices use automated speech recognition to interact with end users. Popular applications include Amazon Alexa, Apple Siri and Google Home.
* **Computer software**: Many tasks in an office environment are simplified by conversational AI, such as search autocomplete when you search something on Google and spell check.

While most AI chatbots and apps currently have rudimentary problem-solving skills, they can reduce time and improve cost efficiency on repetitive customer support interactions, freeing up personnel resources to focus on more involved customer interactions. Overall, conversational AI apps have been able to replicate human conversational experiences well, leading to higher rates of customer satisfaction.

**Chapter 2**

**Survey of Technologies**

**2.1 Types of Chatbots**

Chatbots are software systems that simulate human-like conversations. Chatbots are powered by pre-programmed responses, artificial intelligence, or both. Based on the applied mechanism, a chatbot processes a user’s question to deliver a matching answer. There are two main types of chatbots, and those types also tell us how they communicate. They are retrieval based and generative chatbots,

* **Retrieval Based Chatbots** – Retrieval based chatbots communicate using predefined answers. Such chatbot models are provided with a database containing predefined sets of responses. The chatbot model takes as input - a context and a set of responses and outputs the relevance score of each response with respect to the context. The set of responses comprises both positive and negative responses. The positive responses are the ones more likely to be valid with respect to the context and the negative responses are those with little relevance to the context. Customer Service bots, business bots and in some cases even therapy bots can fall into this category.

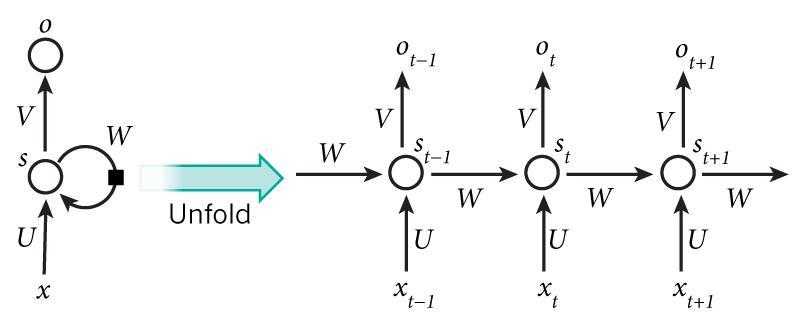
Since the retrieval-based chatbots predefine well-purified question-answer pairs in advance, they do not return responses including grammatical errors. However, they have some deficiencies - the responses are restricted into a predefined set and are not sensitive to changes of user’s queries. It’s worth underlining that rule-based chatbots can't learn from past experiences. They respond based on what they know at that moment. The only way to make a rule-based chatbot better is to equip it with more predefined answers and improve its rule-based mechanisms.

* **Generative Chatbots** - A generative chatbot is an open-domain chatbot program that generates original combinations of language rather than selecting from predefined responses. These chatbots are trained on very large conversational data. Generative chatbots use a combination of supervised learning, [unsupervised learning](https://mobidev.biz/blog/unsupervised-machine-learning-improve-data-quality), reinforcement learning, and adversarial learning for multi-step training. Supervised learning structures a conversation as a sequence-to-sequence problem. Sequence-to-sequence learning maps user inputs to a computer-generated response.

**2.2 Architectures for Conversational AI**

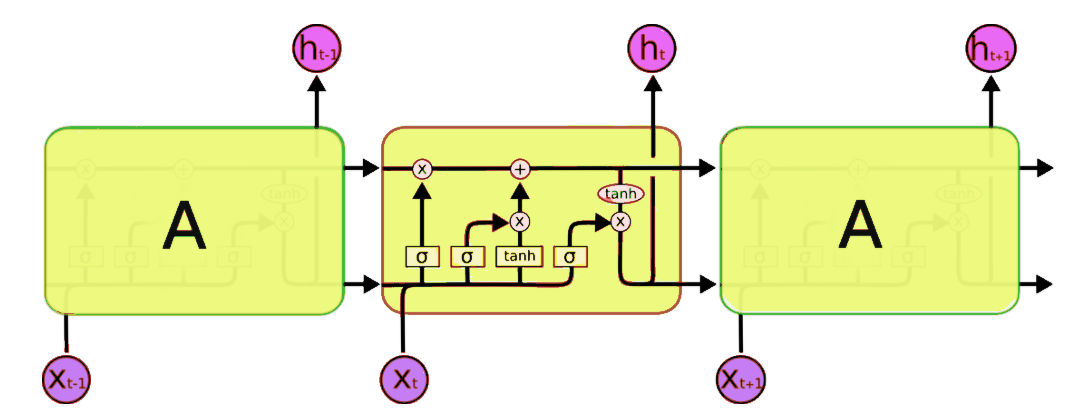
**2.2.01 Recurrent Neural Networks**

Recurrent Neural Networksare powerful models that are uniquely capable of dealing with sequential data, like natural language, speech synthesis etc. They take a fixed-sized vector as input and produce a vector output. RNNs are called recurrent because they perform the same task for every element of a sequence with the output being dependent on the previous computations.



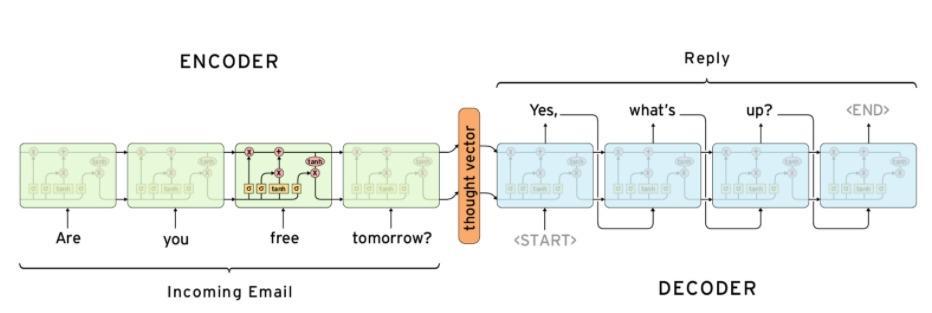
**2.2.02 Long Short Term Memory Networks**

Long Short Term Memory networks – usually just called “LSTMs” – are a special kind of RNN, capable of learning long-term dependencies. LSTMs are explicitly designed to avoid the long-term dependency problem or in other words, the vanishing gradient problem in traditional RNNs. Remembering information for long periods of time is practically their default behaviour.



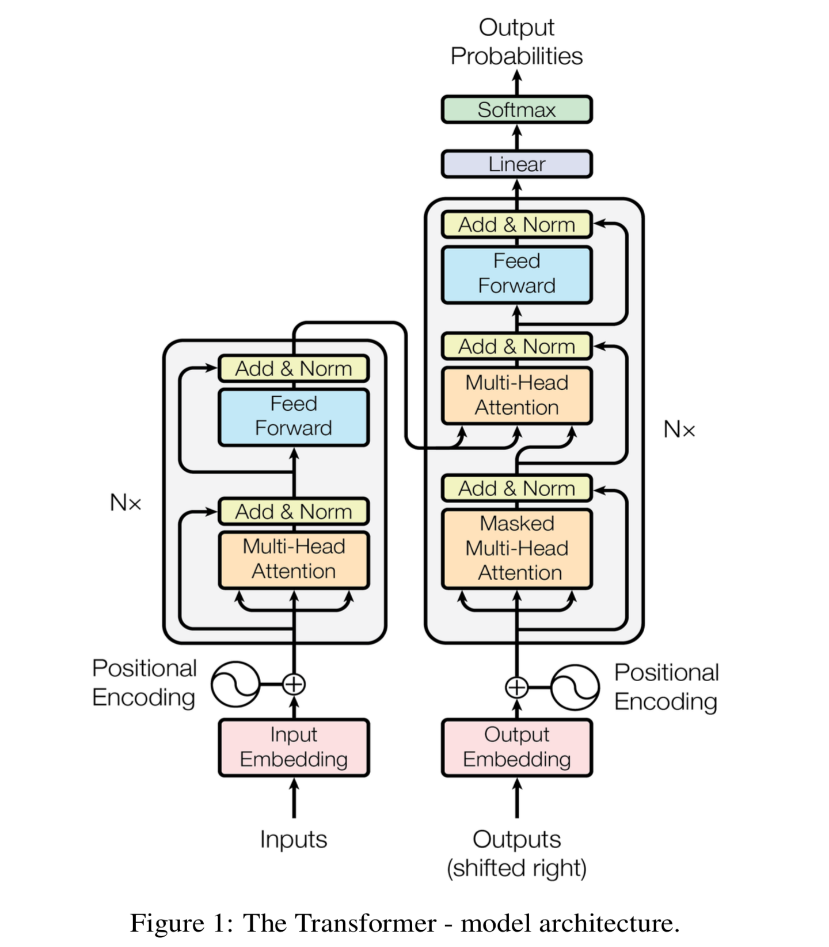
**2.2.03 Sequence to Sequence Models**

Sequence To Sequence model in recent times has become the go-to model for Dialogue Systems and Machine Translation. It consists of two RNNs (more specifically two LSTMs) - An Encoder and a Decoder. The encoder takes a sequence (sentence) as input and processes one symbol (word) at each timestep. The output from the encoder is called the context or thought vector, as it represents the intention of the sequence. From the context, the decoder generates another sequence, one symbol (word) at a time.



**2.2.04 Transformer Model**

Most sequence generation models have an encoder-decoder structure ( as in sequence to sequence model ). The transformer model follows the same overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder. Self-attention, sometimes called intra-attention, is an attention mechanism relating different positions of a single sequence to compute a representation of the same sequence. The outputs are aggregates of these interactions and attention scores.



**Chapter 3**

**Requirements and Analysis**

**3.2 Challenges**

Deploying a fully functional open domain generative conversational bot from scratch is a fairly complicated endeavour considering the lack of hardware resources and computing power in a personal computer system. So, the main challenge for us is to develop a model and train it with satisfying results.

We are planning on incorporating a pre trained Transformer model (available on various open source platforms like *github* and *huggingface*), and then preprocessing our dataset and tuning the model’s hyper-parameters further to our own needs, to construct our generative module pipeline and significantly speed up the training and execution process and completely avoid the heavily resource intensive training process *(****only a concept at this point****).*

Current state-of-the-art models are trained using millions (if not billions) of parameters and on large varied datasets which if implemented from scratch requires a significantly powerful system to train. Furthermore, these pretrained models are much more efficient and optimized for development purposes. Hence, it seemed viable to use a pretrained model instead of building one from scratch.

**3.2 Dependencies**

The following are the recommended dependencies or packages which are required to build and properly execute the project:

* OS : Windows 10 or higher, Linux or Mac
* Python 3.8 or higher
* PyTorch
* Numpy
* Matplotlib
* Nvidia CUDA libraries for Nvidia GPU utilization **OR**
* OpenCL library for AMD GPU utilization

**3.3 Hardware Requirements**

These are the minimum hardware requirements for executing this project:

* CPU – Intel Core I5
* RAM – At Least 8 GB of Memory
* GPU – At Least 4GB of VRAM
* Atleast 30GB of HDD space (required for dataset and data preprocessing)

**Chapter 4**

**System Design**

**4.1 Data Preprocessing**

1. **Selection of  a dataset**

An effective chatbot requires a massive amount of training data to quickly and efficiently reply to user questions. However, the main obstacle to the development of a chatbot is obtaining realistic and task-oriented dialogue data to train these machine learning-based systems. *A deep learning model is only as good as the training data fed into it.* As the chatbot not only answers user questions but also converses with them, it becomes imperative that the correct dataset is used for training the model. We are using corpus from *Twitter* and a corpus of Reddit Data fetched and stored using SQL as well as the *Cornell movie* dataset which contains a large collection of fictional conversations extracted from raw movie scripts.

1. **Preprocessing the dataset**

The training process of such complicated deep learning models is very resource-intensive. To keep the training simple and fast, the datasets need to be further processed before feeding them to the neural network model.

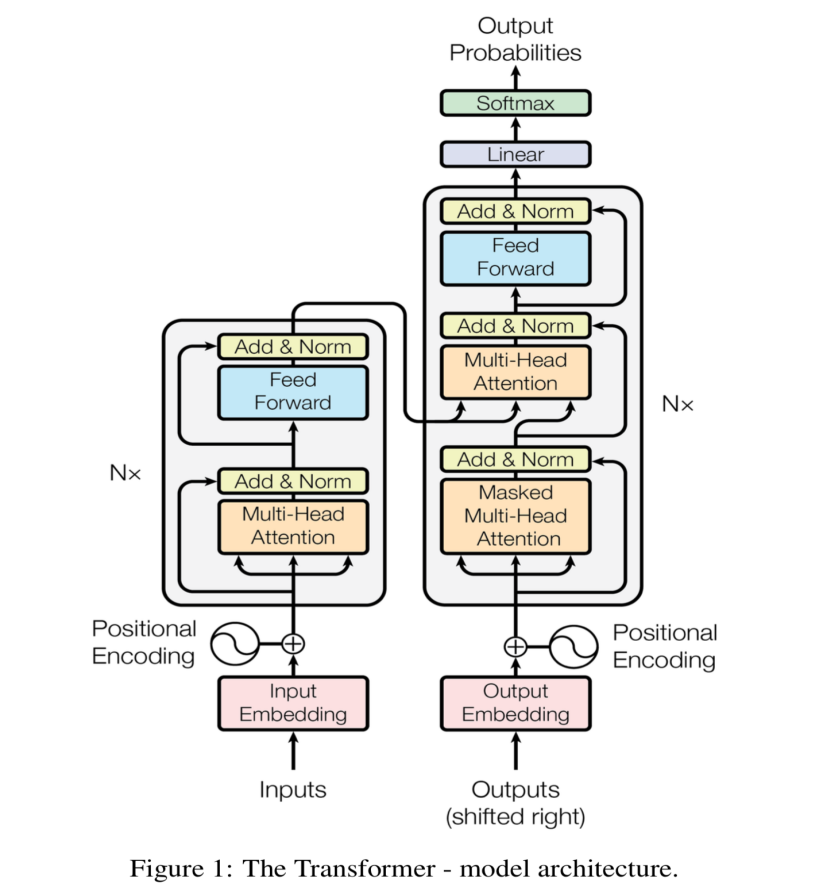
Processing of the dataset further includes,

* Extract conversation pairs into a list of questions and answers.
* Removing special characters and abbreviated texts (I’m, he’s, you’re and so on).
* *Tokenization* - mapping words to IDs and IDs to words (each word is a token) and creating the vocabulary.
* The padding algorithm to handle the variable sequence length problem

**4.2 Basic Modules**

Transformers are taking the natural language processing world by storm. These incredible models are breaking multiple NLP records and pushing the state of the art. They are used in many applications like machine language translation, conversational chatbots, and even to power better search engines.

Most competitive neural sequence transduction models have an encoder-decoder structure much similar to the sequence to sequence models. In the case of the Transformer model, the encoder maps an input sequence of symbol representations ***(x1, . . . , xn)*** to a sequence of continuous representations ***z = (z1, ..., zn)***. Given *z*, the decoder then generates an output sequence ***(y1, . . . , ym)*** of symbols one element at a time. At each step the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next.



A Transformer Model uses Encoder and Decoder models but these are different from the Encoder and Decoder model used in the Seq2Seq architecture in that the Encoder and Decoder models used in the Transformer do not use LSTM, GRU or RNNs hence there are no recurrent connections and thus no “memory” of previous states are implemented. Transformers get around this lack of memory by perceiving entire sequences simultaneously. A detailed study of the Transformer model is discussed below.

**4.1.01 Embedding Module**

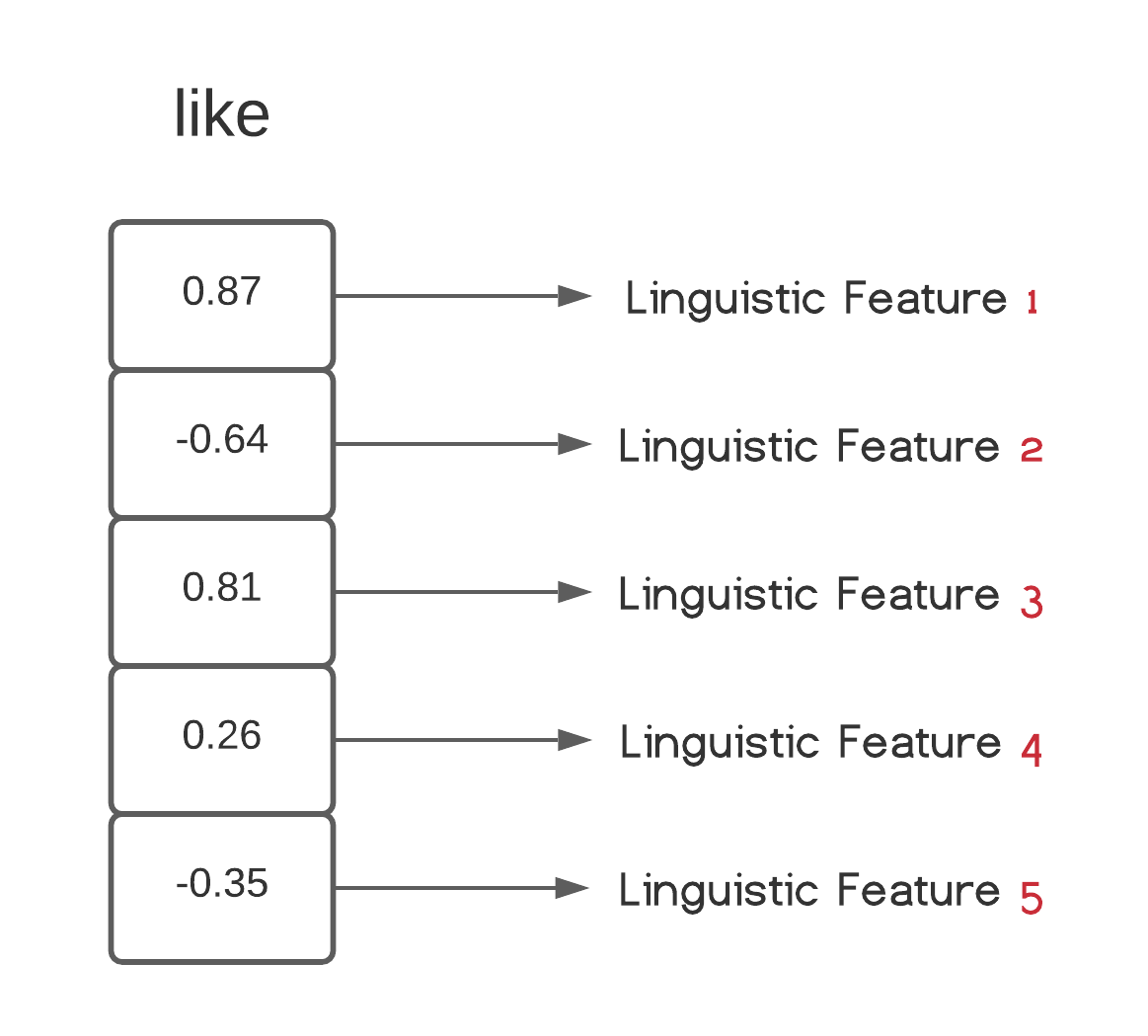
Here is a detailed visualisation of the embedding process in Natural Language Processing models. Consider an input from the user,

***Input*** *– Do you like Game of Thrones*

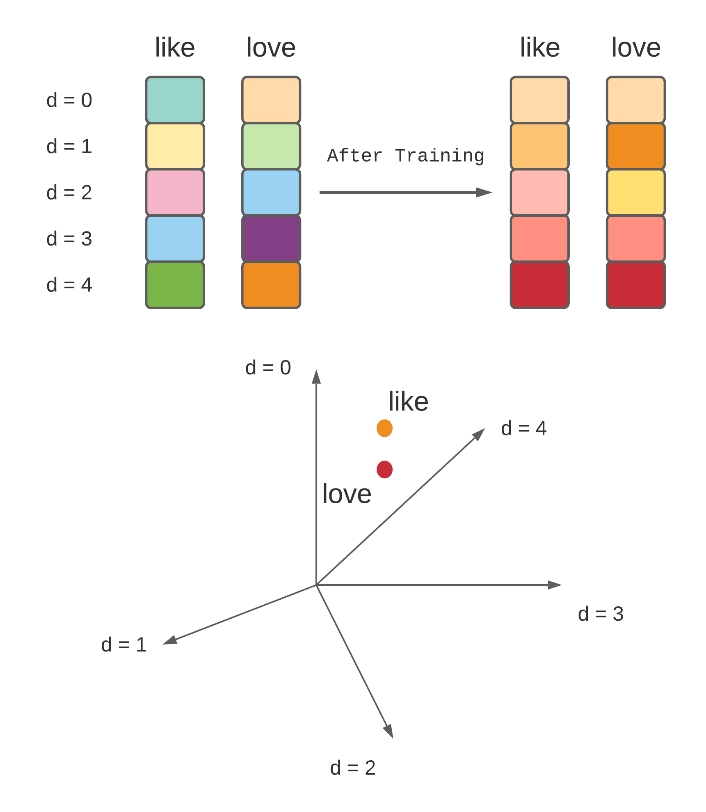
The first step is to fetch the indices of the words occurring in the input sequence from the vocabulary of all the learned words by the model. Therefore we get,

***Vocabulary Indices*** *– [ 2456, 56, 6674, 5345, 86, 145 ]*

Now these indices are fed as the input to the ***Input Embedding Module*** of the network. Here, against each of these word indices obtained from the vocabulary, a vector is generated. Initially, the vectors are filled with random numbers. Later on, during training, the model updates them. These vectors are called ***word embeddings.*** In the transformer model presented in the paper, “Attention is All you Need”, the ***embedding size*** or the length of these vectors are 512.



Each element of the vector or in other words, each “*dimension*” of the word embedding tries to capture a unique linguistic feature for that word. These could be things like whether it is a verb or a pronoun or something else. Now, *n-dimensional* (here it's 5-dimensional) word embeddings can be represented on a *n-dimensional hyperspace*, where words sharing similar linguistic features are plotted closer to each other while dissimilar word embeddings are plotted farther apart from each other.



*Word Embeddings and their representation*

Hence, the main purpose of the embedding layer is to select the proper embedding of the input words and pass them on to the positional encoding module.

**4.1.02 Positional Encoding Module**

Position and order of words in a sentence is an essential part of any language. They define the grammar and thus the actual semantics of a sentence. Recurrent Neural Networks (RNNs) inherently take the order of word into account i.e they parse a sentence word by word in a sequential manner, but the Transformer architecture ditched the recurrence mechanism in favor of *multi-head self-attention mechanism* (discussed later). Avoiding the RNNs’ method of recurrence will result in massive speed-up in the training time and theoretically, it can capture longer dependencies in a sentence.

As each word in a sentence simultaneously flows through the Transformer’s encoder or decoder stack (layers of encoder or decoder), the model by itself does not have any sense of the position or order for each word. Consequently, there’s still the need for a way to add some information about the positions into the input embeddings.

So a new set of vectors called the ***position embeddings*** are introduced, one for each word embedding. Thay have the same *embedding size* (vector length) as that of the word embeddings. Generally, the *position embeddings* and the *word embeddings* are simply added to generate a new set of embeddings; only this time these embeddings also contain the position as well as the linguistic information of the words in the sentence.

+

+

Position Embedding (pos = 0)

Position Embedding (pos = 1)

***e0  p0 e1 p1***

***ej*** – Word Embedding for Word i

***pj***– Position Embedding for Word i

***i = 0***

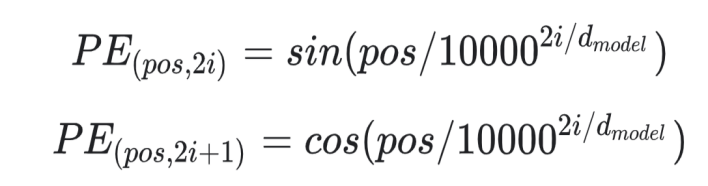
***i = 1***

***i = 2***

***i = 3***

***i = 4***

The tricky part here is to set the values of the various position embeddings. The authors of the paper “Attention is All you Need” came up with a clever trick to use *wave functions* (*sine* and *cosine*) to capture position information.



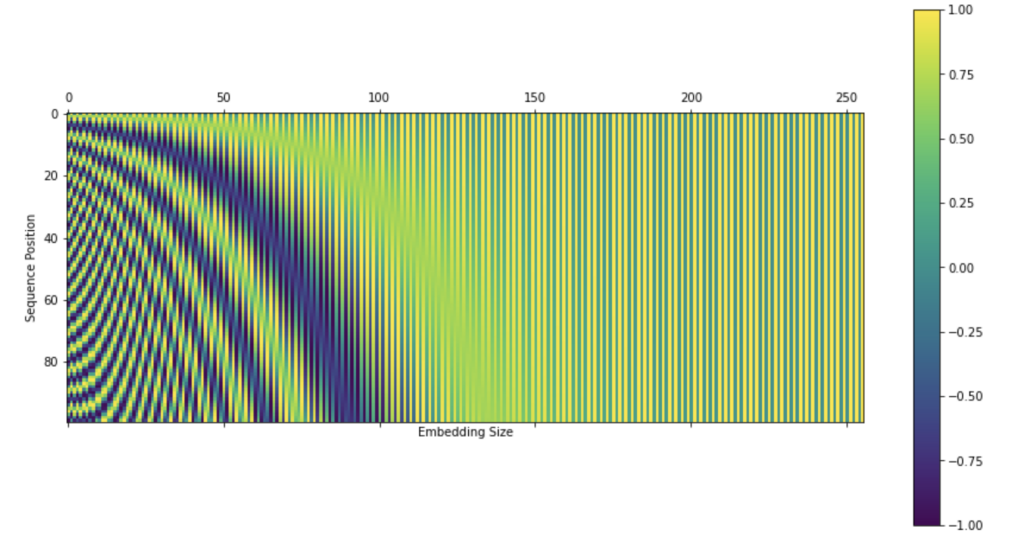
Here,

***pos*** is the position index of the word in the input sequence.

***dmodel*** is the embedding size (which is 512).

***i*** represents the indices of each of the position embedding dimensions

So, basically, for every odd value of ***pos*** on the position vector, create a vector using the *cosine* function. For every even value of ***pos***, create a vector using the *sine* function.

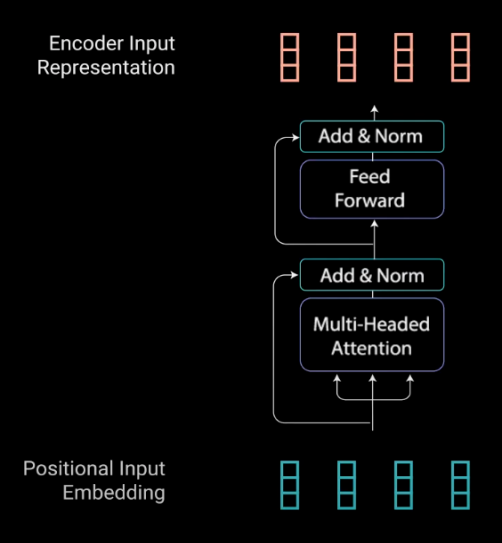


Here is how the position embedding curves looks like when plotted in full scale against the embedding size ***(i)***. The above plot is a great way to visualise the position embedding dimensions (or values) for a particular embedding size. Finally, the position embeddings are added to their corresponding word embeddings.

From now on, the final embeddings of the words will be referred to as the ***position embeddings*** for easier understanding.

**4.1.03 The Encoder Stack**

This module consists of a stack of ***N*** Encoder Layers. These encoder layers, each of which has two sublayers – the ***Multi-Head Attention Layer*** and the ***Position-wise Full Connected Feed-Forward Network layer.*** Finally each encoder layer has an ***Add-Norm*** (Addition and Normalization) layer component.



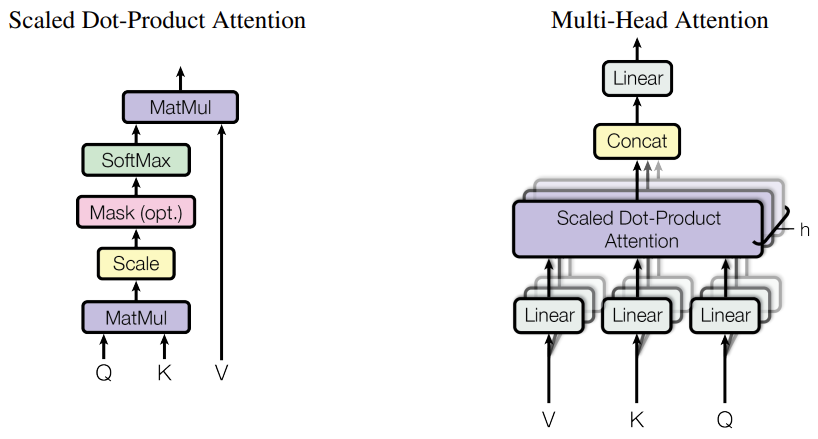
*An Encoder Layer*

All the operations inside each encoder stack are implemented to encode the input embeddings to a continuous representation with attention information. This will help the decoder focus on the appropriate words in the input during the decoding process. The encoder can be stacked on top of each other N times to further encode the information, where each stack has the opportunity to learn different attention representations therefore potentially boosting the predictive power of the transformer network and offer more parallelization.

**Multi-head Attention Layer**

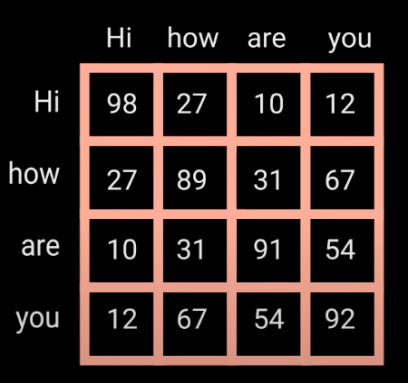
The *Multi-Head Attention* layer in the encoder applies a specific attention mechanism called the ***self-attention***. Self-Attention allows the model to associate each word in the input sequence, to other words. In other words, the task of self attention is to find out the relationship of each word in a sequence with all the other words in that sequence.

The input to the attention layer, in the case of encoders, is the *position embedding matrix.* Three identical copies of the input embeddings are made – *Query (Q)*, *Key (K)* and *Value (V)*. The concept of query, key and value comes from retrieval systems like search engines. For example, the youtube search engine will map the user query against a set of keys (video title, description etc.) associated with candidate videos in the database, then retrieve the best matched videos (values).



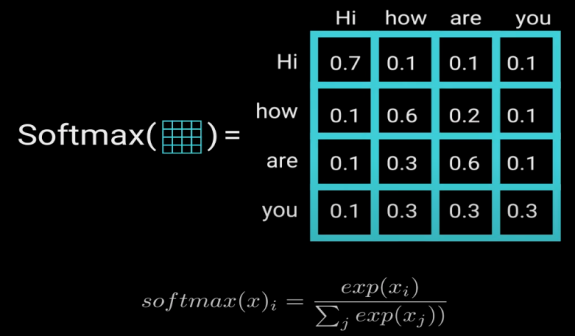
*Self Attention Module*

After feeding the query, key and value matrices through a linear layer, the queries and keys undergo a *dot-product matrix multiplication* operation to produce a *score matrix*. The score matrix determines how much focus should a particular word put on other words. So each word will have a score that corresponds to other words in the time-step. The higher the score the more focus. This is how the queries are mapped to the keys.



*Example of a Score Matrix*

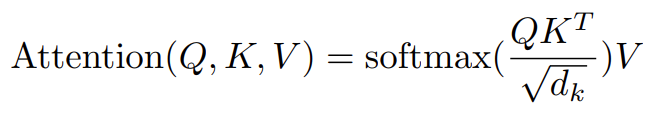
Then these scores are scaled down by dividing each score by the square root of the dimension of the query and key (which is ***dk***). The scaling is done in order to achieve more stable and normalized gradients since multiplication can have exploding effects.



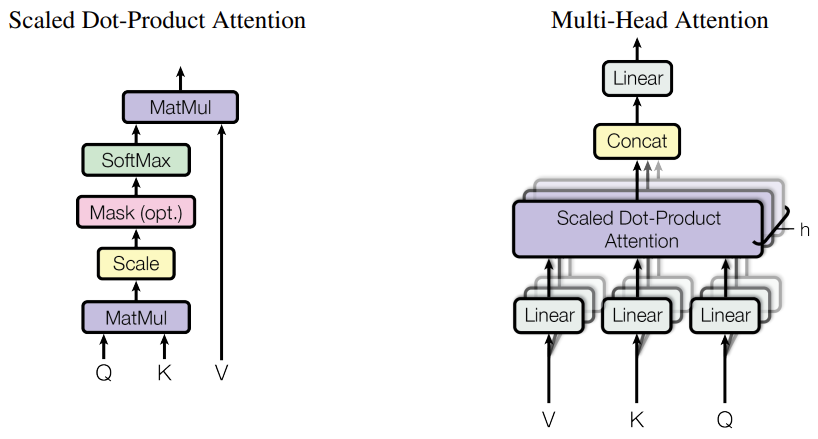
*Softmax operation on the scaled scores to get attention weights*

A ***softmax*** of the scaled scores is performed to get the *attention weights*, which outputs probability values between 0 and 1. By doing a softmax the higher scores get elevated, and lower scores are depressed. This allows the model to be more confident about which words to attend too.

Finally, the attention weights are multiplied to the value matrix to get an output matrix. The higher softmax scores will keep the value of words the model learns is more important. The lower scores will drown out the irrelevant words. The following equation summarizes the whole process,



The query, key and value matrices go through the self-attention process individually. Each self-attention process is called a *head*. Each head produces an output vector that gets concatenated into a single vector before going through the final linear layer. In theory, each head would learn something different therefore giving the encoder model more representation power.



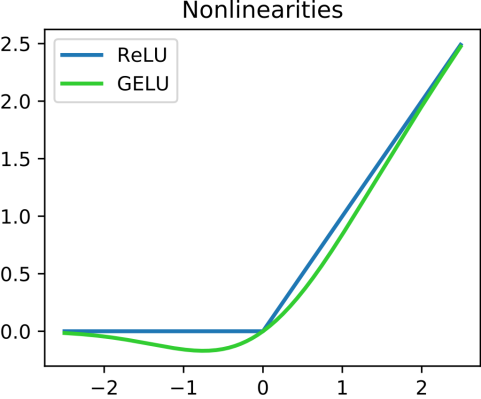
*Multi-Head Attention Layer*

To sum it up, multi-headed attention is a module in the transformer network that computes the attention weights for the input and produces an output vector with encoded information on how each word should attend to all other words in the sequence.

The multi-headed attention output vector is added to the original positional input embedding. This is called a *residual connection*. The output of the residual connection goes through a *layer normalization*.

**Feed Forward Network Layer**

The normalized residual output gets projected through a pointwise feed-forward network for further processing. The pointwise feed-forward network is a couple of linear layers with a ***ReLU*** (Rectified Linear Unit) activation function in between. The output of that is then again added to the input of the Add and Normalize layer and further normalized.



*Rectified Linear Unit activation function*

The residual connections help the network train, by allowing gradients to flow through the networks directly. The layer normalizations are used to stabilize the network which results in substantially reducing the training time necessary. The pointwise feedforward layer is used to project the attention outputs potentially giving it a richer representation.

**4.1.04 The Decoder Stack**

The main purpose of the decoder is to generate text sequences. The decoder stack also comprises multiple decoder layers. Each decoder layer has similar sub-layers as the encoder layers – it has two multi-head attention layers (one is masked, discussed later), a pointwise feed-forward layer, and layer normalization (add and normalization) layers after each sub-layer.

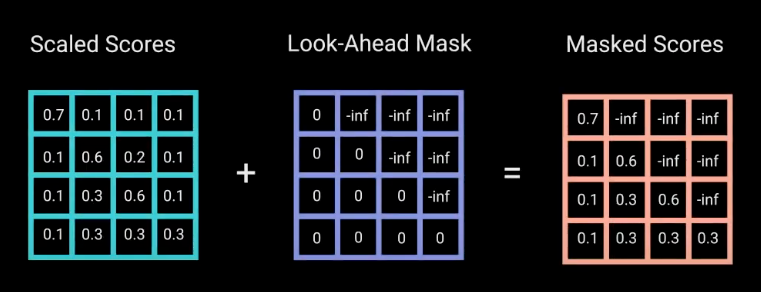
These sub-layers behave similarly to the layers in the encoder but each multi-headed attention layer has a different job. The decoder is autoregressive, it begins with a start token, and it takes in a list of previous outputs as inputs, as well as the encoder outputs that contain the attention information from the input.

The beginning of the decoder is pretty much the same as the encoder. The input goes through an embedding layer and positional encoding layer to get positional embeddings. The positional embeddings get fed into the first multi-head attention layer which computes the attention scores for the decoder’s input only.

**Masked Multi-head Attention Layer**

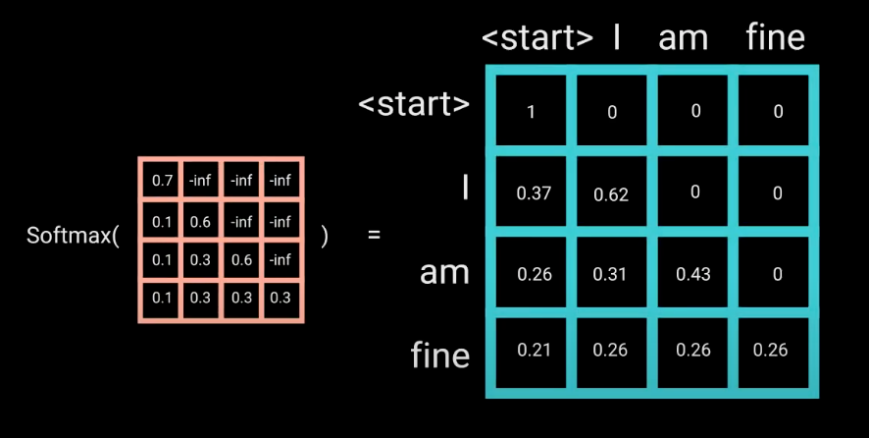
This multi-headed attention layer operates slightly differently. Since the decoder is autoregressive and generates the sequence word by word, the model needs to prevent it from taking future words from the encoder into account. For example, when computing attention scores on the word “am”, the model should not have access to the word “fine”, because that word is a future word that was generated after. The word “am” should only have access to itself and the words before it. This is true for all other words, where they can only attend to previous words.

Therefore, the model needs to prevent computing attention scores for future word embeddings. This method is called *masking* and hence this attention layer is called *Masked Multi-Head Attention Layer*. To prevent the decoder from taking future words into account, a ***look-ahead mask*** is applied to the *scaled score matrix*. The look-ahead mask is a matrix with the same size as the score matrix with values of *zeros* and *negative infinities* only. The mask is simply added to the scales score matrix, and a *masked score matrix* is obtained.



*Using Look-Ahead Mask to generate Masked Score Matrix*

The reason for the mask is because once the model calculates the softmax of the masked scores, the negative infinities get zeroed out, leaving zero attention scores for future tokens. This essentially tells the model to put no focus on those words whose attention scores are zero. This masking is the **only difference** in how the attention scores are calculated in the first multi-headed attention layer. This layer still has multiple heads that the mask is being applied to, before getting concatenated and fed through a linear layer for further processing. The output of the first multi-headed attention is a masked output vector with information on how the model should attend to the decoder’s input.



*Softmax Operation on the Masked Score Matrix*

For the second multi-headed attention layer, the encoder’s outputs are the queries and the keys, and the first multi-headed attention layer outputs act as the values. This process matches the encoder’s input to the decoder’s input, allowing the decoder to decide which encoder input is relevant to put a focus on. The output of the second multi-headed attention goes through a pointwise feedforward layer for further processing.

**4.3 Flowchart**

This is an overall proposed workflow for our open domain generative chatbot. The modules described in the flowchart are only an overview of an ongoing planning process which are subject to further changes (addition or removal of some modules) as we delve deeper into the implementation of our project.

Start

Selection of Conversational Dataset for Training the Transformer Model

Writing Code for Data Preprocessing for Preparing data for training purpose.

Writing Code for Creating Training and Testing Data.

Writing Code for the Transformer Architecture for Chatbot.

Training the Transformer Model

Writing Code for Conversing with the Transformer Bot

Interacting with the Bot

End

**References**

*Asoke Nath, Rupamita Sarkar, Swastik Mitra, Rohitaswa Pradhan*, **“Designing and Implementing Conversational Intelligent Chatbot Using Natural Language Processing”**, *International Journal of Scientific in Computer Science, Engineering and Information Technology (IJSRCSEIT)*, 22nd May 2021.

**[Transformer model for language understanding](https://www.tensorflow.org/text/tutorials/transformer),** *Tensorflow Documentation*

*Denny Britz*, **“Recurrent Neural Network Series”**, *Wild ML,* 17th September 2015

*Toma´s Mikolov, Stefan Kombrink, Luka´s Burget, Jan “Honza” Cernock, Sanjeev Khudanpur,* **“Extensions of Recurrent Neural Networks Language Model”**, *John Hopkins University USA*, 1st June 2011

*Christopher Olah*, **“Understanding LSTM Networks”**, Colah’s Blog, 27th August 2015

*Ilya Sutskever, Oriol Vinyals, Quoc V.Le,* **“Sequence to Sequence Learning with Neural Networks”**, *[arXiv:1409.3215](https://arxiv.org/abs/1409.3215),* 10th September 2014

*Kyunghyun Cho, Bart vanMerrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, Yoshua Bengio,* **“Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation”**, *[arXiv:1406.1078](https://arxiv.org/abs/1406.1078)***,** 3rd June 2014

*Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, Illia Polosukhin,* **“Attention is All You Need”,** *[arXiv:1706.03762](https://arxiv.org/abs/1706.03762)*, 6th Dec 2017

*Buomsoo Kim*, **“Attention in Neural Networks”,** 11th November 2020

*EleutherAI,* **GPT-neo and GPT-neox,** *[EleutherAI (github.com)](https://github.com/EleutherAI)*

*Michael Phi*, **Illustrated Guide to Transformers- Step by Step Explanation,** 1st May 2020