ABOUT THE PROJECT

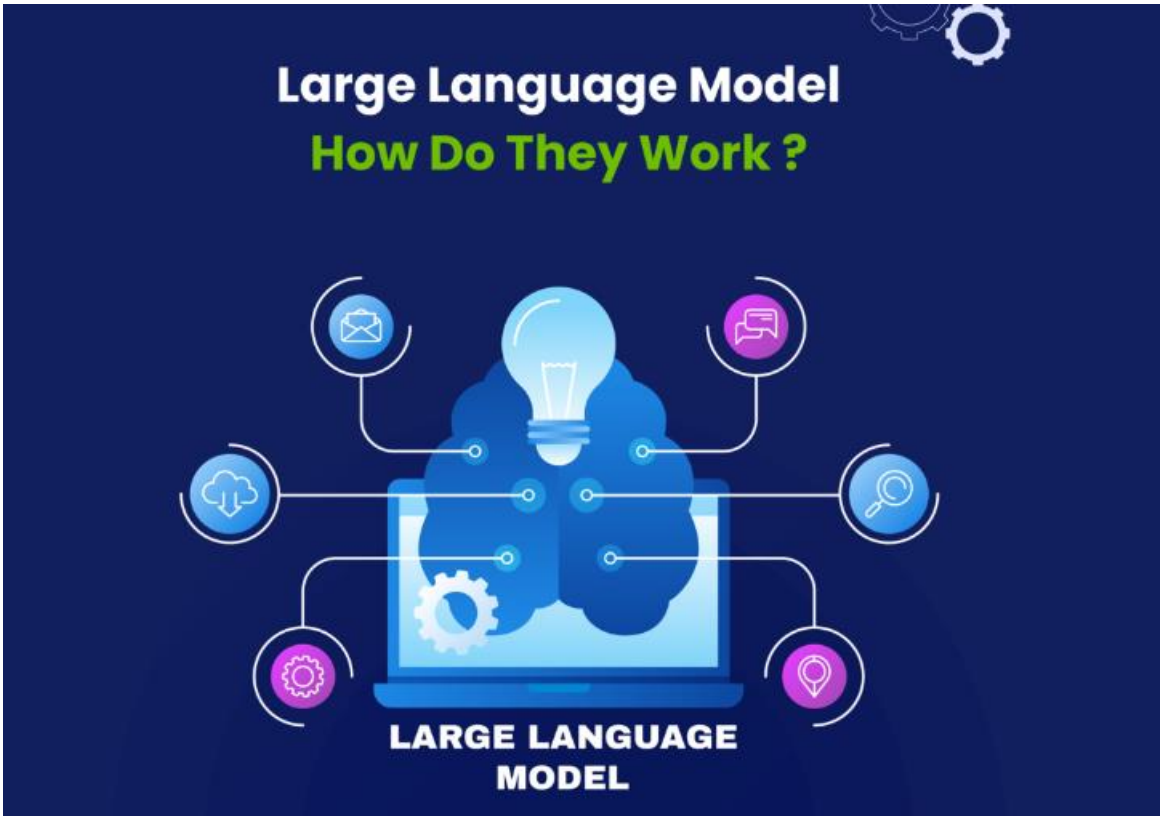
Flan T5 Chatbot for Dialogue Summarization

The Large Language Model (LLM)-Empowered Chatbot for Dialogue Summarization represents a cutting-edge initiative aimed at revolutionizing the efficiency and precision of information condensation within the realm of natural language processing. The project addresses the pressing need for advanced Dialogue Summarization capabilities in the face of an ever-expanding volume of textual data. Leveraging the robust capabilities of Google's Flan T5, a preeminent large language model, the chatbot is meticulously fine-tuned to excel in the nuanced task of distilling information while maintaining contextual richness.

Users are empowered to interact with the chatbot via a user-friendly interface, initiating collaborative engagements for Dialogue Summarization. What distinguishes this project is its strategic integration of Flan T5, a pre-trained model with a formidable reputation in handling diverse NLP tasks. The fine-tuning process involves adapting the model specifically to the intricacies of Dialogue Summarization, ensuring optimal performance in distilling information accurately.

Experimentation with various hyperparameter configurations, such as learning rate and

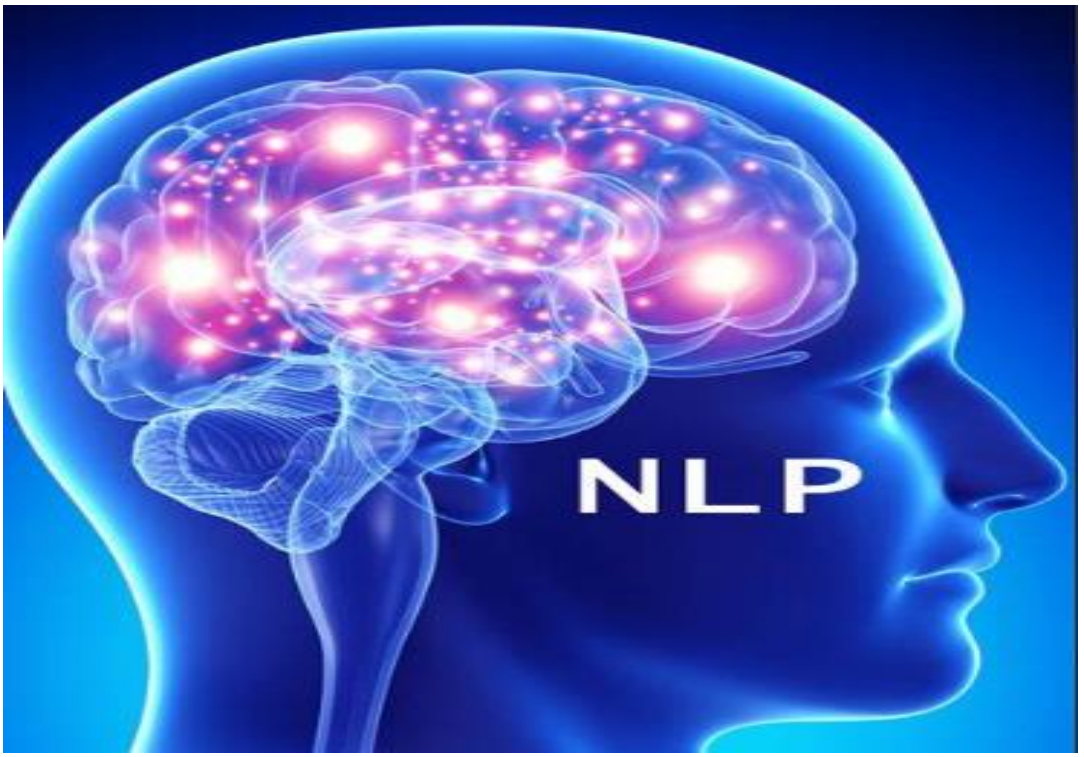
batch size, is a pivotal phase in optimizing the chatbot's proficiency. The evaluation metrics, including accuracy, BLEU score, and ROUGE score, provide quantifiable insights into the model's effectiveness in summarizing text. This iterative refinement process aims at honing the chatbot's abilities to align with real-world applications.



Beyond the technical aspects, the project encompasses the deployment of the chatbot into practical applications and services, exemplifying the tangible impact of LLMs in enhancing user experiences. Furthermore, the culmination of the internship is encapsulated in a research paper within the domain, contributing valuable insights and perspectives to the broader discourse on large language models and their applications.

In essence, the Large Language Model-Empowered Chatbot for Dialogue Summarization stands as an exemplar of innovation, seamlessly merging the advanced capabilities of Flan T5 with the practicality of a collaborative chatbot. This project not only showcases the prowess of LLMs in handling complex NLP tasks but also positions them as pivotal tools in shaping the future of information distillation and Dialogue Summarization. The successful execution of this internship underscores the significance of integrating state-of-

the-art language models in solving real-world challenges, setting a precedent for future advancements in natural language processing.



PROBLEM STATEMENT:

The problem we're tackling is how to make summarizing lots of text better and more accurate. The usual methods often miss important details, so we're using a powerful tool called Flan T5, a large language model, to improve how we summarize information. We want to make this process easier for users, so we're creating a friendly chatbot that uses Flan T5 to quickly and accurately distill information. The aim is to provide a solution that not only improves how we understand and summarize text but also makes it more collaborative and user-friendly.

MOTIVATION

This project comes from an exciting chance to make summarizing information better and more accurate than current methods. We're diving into the world of Flan T5, a powerful language tool, to build a friendly chatbot. The goal is to revolutionize how we quickly and accurately distill information from large amounts of text. Current methods sometimes miss important details, and we want to fix that. This project isn't just about improving text summarization; it's also a cool opportunity for us to learn and explore new technologies. By doing this, we're stepping into the forefront of innovation, gaining skills that will help shape the future of how we process information in the ever-changing tech world.

This project stems from an exciting opportunity to overcome the limitations inherent in current text summarization methods and explore the transformative potential of Flan T5, a formidable large language model, in a practical and innovative context. In the realm of information condensation, where precision and accuracy are paramount, the integration of Flan T5 offers a sophisticated solution. By leveraging the capabilities of this advanced language model within a user-friendly chatbot, we aim to revolutionize the process of summarizing vast amounts of text, addressing the shortcomings of existing methods that often miss essential details.

In essence, the motivation behind developing a Flan T5-powered chatbot for text summarization is deeply rooted in the desire to elevate the accuracy and efficiency of summarization processes. This project not only seeks to advance the capabilities of text summarization but also represents an exciting opportunity to immerse ourselves in the world of cutting-edge natural language processing technologies. By undertaking this venture, we position ourselves as digital innovators, equipped with the skills and insights needed to shape the future of information processing in the dynamic landscape of NLP.

THEORTICAL CONCEPTS

Natural Language Processing (NLP)

Natural language processing (NLP) is a form of Artificial Intelligence (AI) that allows computers to understand human language, whether it be written, spoken, or even scribbled. As AI-powered devices and services become increasingly more intertwined with our daily lives and world, so too does the impact that NLP has on ensuring a seamless human-computer experience.

Natural language processing (NLP) is a subset of artificial intelligence, computer science, and linguistics focused on making human communication, such as speech and text, comprehensible to computers.

NLP is used in a wide variety of everyday products and services. Some of the most common ways NLP is used are through voice-activated digital assistants on smartphones, email-scanning programs used to identify spam, and translation apps that decipher foreign languages.

Natural language techniques :

NLP encompasses a wide range of techniques to analyze human language. Some of the most common techniques you will likely encounter in the field include:

Sentiment analysis: An NLP technique that analyzes text to identify its sentiments, such as "positive," "negative," or "neutral." Sentiment analysis is commonly used by businesses to better understand customer feedback.

Summarization: An NLP technique that summarizes a longer text, in order to make it more manageable for time-sensitive readers. Some common texts that are summarized include reports and articles.

Keyword extraction: An NLP technique that analyzes a text to identify the most important keywords or phrases. Keyword extraction is commonly used for search engine optimization (SEO), social media monitoring, and business intelligence purposes.

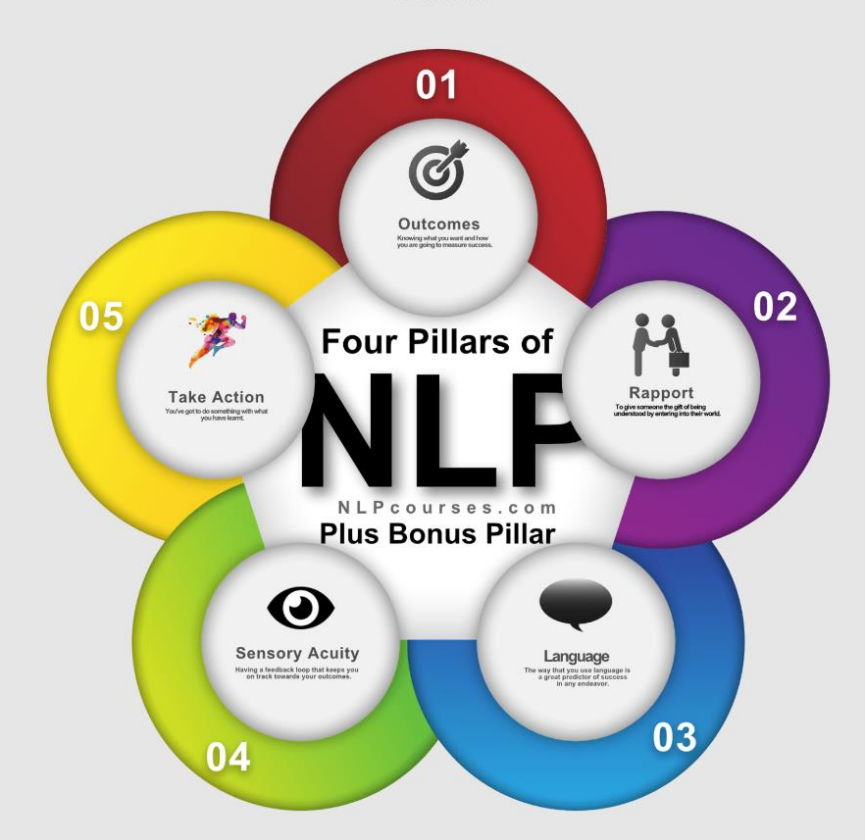
Tokenization: The process of breaking characters, words, or subwords down into "tokens" that can be analyzed by a program. Tokenization undergirds common NLP tasks like word modeling, vocabulary building, and frequent word occurrence.

Natural language processing examples

Although natural language processing might sound like something out of a science fiction novel, the truth is that people already interact with countless NLP-powered devices and services every day.

Online chatbots, for example, use NLP to engage with consumers and direct them toward appropriate resources or products. While chat bots can't answer every question that customers may have, businesses like them because they offer cost-effective ways to troubleshoot common problems or questions that consumers have about their products.

Another common use of NLP is for text prediction and autocorrect, which you've likely encountered many times before while messaging a friend or drafting a document. This technology allows texters and writers alike to speed-up their writing process and correct common typos.



GENERATIVE AI

Generative AI is a type of artificial intelligence technology that can produce various types of content, including text, imagery, audio and synthetic data. The recent buzz around generative AI has been driven by the simplicity of new user interfaces for creating high-quality text, graphics and videos in a matter of seconds.

The technology, it should be noted, is not brand-new. Generative AI was introduced in the 1960s in chatbots. But it was not until 2014, with the introduction of generative adversarial networks, or GANs -- a type of machine learning algorithm -- that generative AI could create convincingly authentic images, videos and audio of real people.

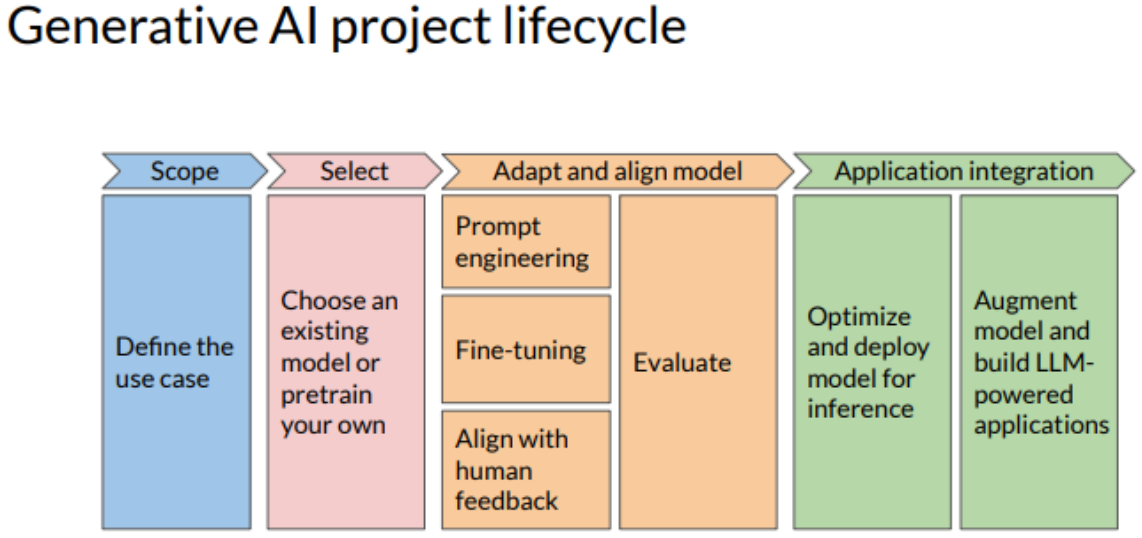
On the one hand, this newfound capability has opened up opportunities that include better movie dubbing and rich educational content. It also unlocked concerns about deepfakes -- digitally forged images or videos -- and harmful cybersecurity attacks on businesses, including nefarious requests that realistically mimic an employee's boss.

Two additional recent advances that will be discussed in more detail below have played a critical part in generative AI going mainstream: transformers and the breakthrough language models they enabled. Transformers are a type of machine learning that made it possible for researchers to train ever-larger models without having to label all of the data in advance. New models could thus be trained on billions of pages of text, resulting in answers with more depth. In addition, transformers unlocked a new notion called attention that enabled models to track the connections between words across pages, chapters and books rather than just in individual sentences. And not just words: Transformers could also use their ability to track connections to analyze code, proteins, chemicals and DNA.

The rapid advances in so-called large language models (LLMs) -- i.e., models with billions or even trillions of parameters -- have opened a new era in which generative AI models can write

engaging text, paint photorealistic images and even create somewhat entertaining sitcoms on the fly. Moreover, innovations in multimodal AI enable teams to generate content across multiple types of media, including text, graphics and video. This is the basis for tools like Dall-E that automatically create images from a text description or generate text captions from images.

These breakthroughs notwithstanding, we are still in the early days of using generative AI to create readable text and photorealistic stylized graphics. Early implementations have had issues with accuracy and bias, as well as being prone to hallucinations and spitting back weird answers. Still, progress thus far indicates that the inherent capabilities of this generative AI could fundamentally change enterprise technology how businesses operate. Going forward, this technology could help write code, design new drugs, develop products, redesign business processes and transform supply chains.



LARGE LANGUAGE MODEL(LLM)

A large language model (LLM) is a deep learning algorithm that can perform a variety of Natural Language Processing (NLP) tasks**.** Large language models use transformer models and are trained using massive datasets — hence, large. This enables them to recognize, translate, predict, or generate text or other content.

Large language models are also referred to as neural networks (NNs), which are computing systems inspired by the human brain. These neural networks work using a network of nodes that are layered, much like neurons.

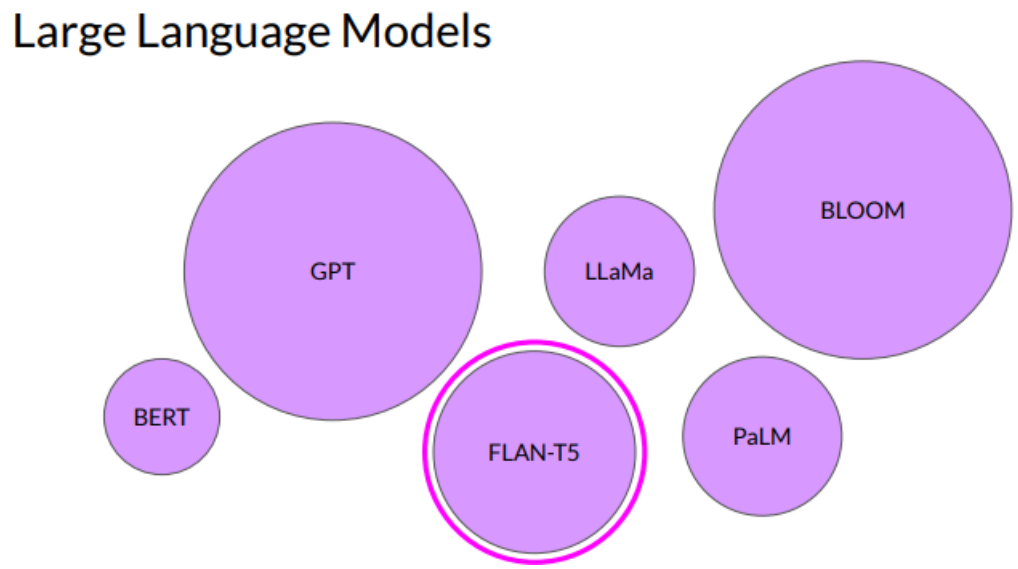
In addition to teaching human languages to artificial intelligence (AI) applications, large language models can also be trained to perform a variety of tasks like understanding protein structures, writing software code, and more. Like the human brain, large language models must be pre-trained and then fine-tuned so that they can solve text classification, question answering, document summarization, and text generation problems. Their problem-solving capabilities can be applied to fields like healthcare, finance, and entertainment where large language models serve a variety of NLP applications, such as translation, chatbots, AI assistants, and so on.

Large language models also have large numbers of parameters, which are akin to memories the model collects as it learns from training. Think of these parameters as the model's knowledge bank.

In recent years, the field of natural language processing (NLP) has witnessed a paradigm shift with the advent of Large Language Models (LLMs). These models, built on deep neural network architectures, have demonstrated unparalleled capabilities in understanding, generating, and processing human language at an unprecedented scale. At the forefront of this transformative wave is the utilization of vast amounts of data and computational resources, allowing LLMs to learn intricate patterns, contextual nuances, and syntactic structures inherent in diverse linguistic contexts.

One notable exemplar in this domain is OpenAI's GPT-3 (Generative Pre-trained Transformer 3), a state-of-the-art LLM boasting a staggering 175 billion parameters. The essence of these models lies in their pre-training phase, where they are exposed to extensive corpora of text,

enabling them to learn hierarchical representations of language. This pre-training is followed by fine-tuning on specific tasks, allowing the model to adapt its learned features to more nuanced applications, such as text summarization.



The architectural backbone of LLMs, often based on transformer architectures, facilitates the modeling of long-range dependencies and contextual relationships within text. This attention mechanism allows the model to weigh the significance of different words in a sequence, enabling a more holistic understanding of language structure. As a result, LLMs excel not only in language generation but also in tasks such as translation, sentiment analysis, and, crucially, text summarization.

The theoretical underpinnings of LLMs are rooted in their ability to capture semantic meaning, infer relationships, and generate coherent text. This stems from their training on diverse datasets, enabling them to generalize across a wide range of linguistic contexts. However, the sheer scale of these models poses computational challenges, necessitating access to high-performance computing resources, specialized hardware, and efficient training strategies.

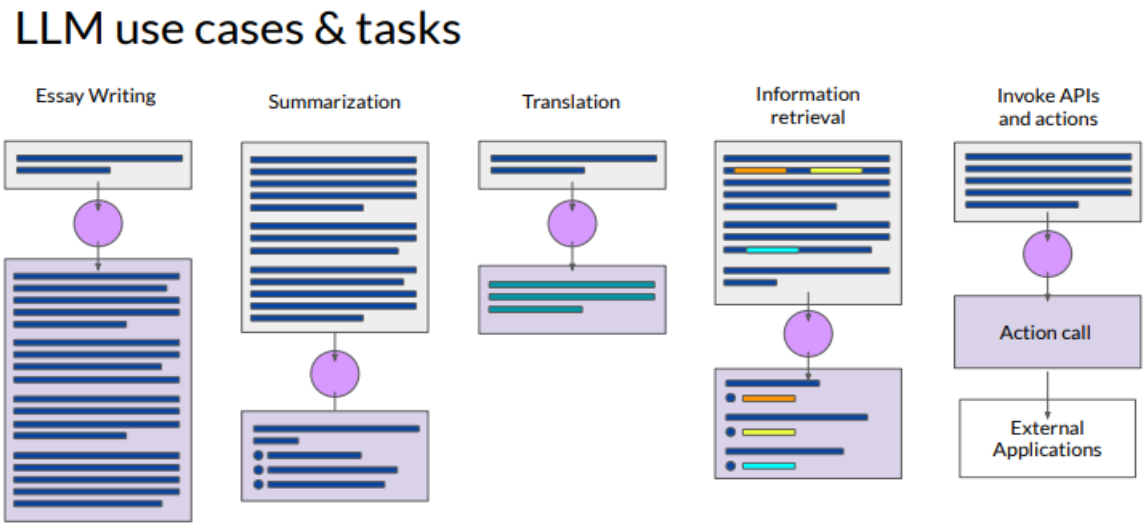
As we delve into the theoretical exploration of LLMs, it becomes evident that these models mark a revolutionary leap in NLP capabilities. Their prowess extends beyond mere language understanding, shaping the landscape of applications ranging from conversational agents to advanced text summarization, as exemplified by the integration of models like Flan T5. Understanding the theoretical foundations of LLMs lays the groundwork for unlocking their potential in various real-world applications, paving the way for continued advancements in natural language processing.

What are the Large Language Models used for?

The main reason behind such a craze about the LLMs is their efficiency in the variety of tasks they can accomplish. From the above introductions and technical information about the LLMs you must have understood that the Chat GPT is also an LLM so, let's use it to describe the use cases of Large Language Models.

- **Code Generation** – One of the craziest use cases of this service is that it can generate quite an accurate code for a specific task that is described by the user to the model.
- **Debugging and Documentation of Code** – If you are struggling with some piece of code regarding how to debug it then ChatGPT is your savior because it can tell you the line of code which are creating issues along with the remedy to correct the same. Also now you don't have to spend hours writing the documentation of your project you can ask ChatGPT to do this for you.
- **Question Answering** – As you must have seen that when AI-powered personal assistants were released people used to ask crazy questions to them well you can do that here as well along with the genuine questions.

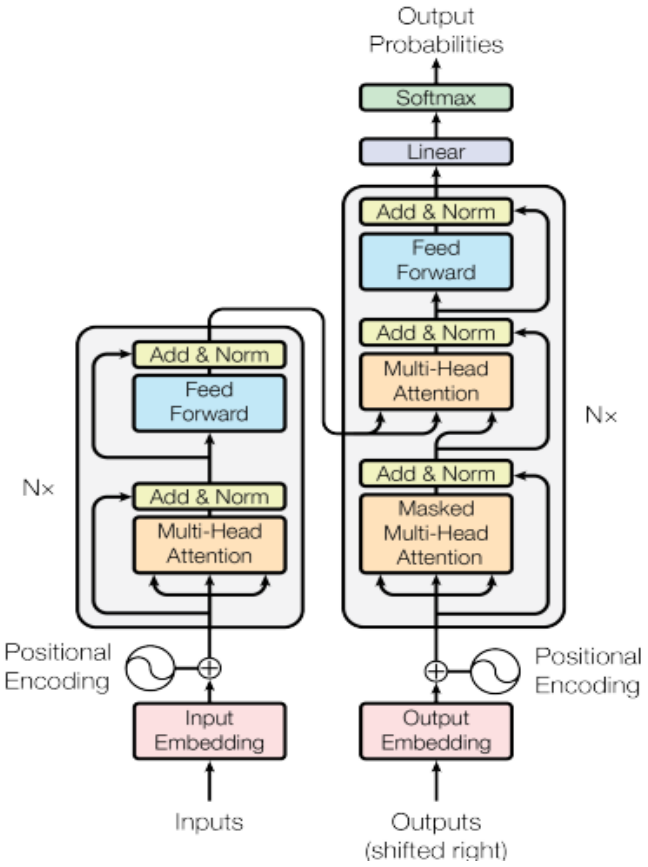
- **Language Transfer** – It can convert a piece of text from one language to another as it supports more than 50 native languages. It can also help you correct the grammatical mistakes in your content.



TRANSFORMERS

"Attention is All You Need" is a research paper published in 2017 by Google researchers, which introduced the Transformer model, a novel architecture that revolutionized the field of natural language processing (NLP) and became the basis for the LLMs we now know - such as GPT, PaLM and others. The paper proposes a neural network architecture that replaces traditional recurrent neural networks (RNNs) and convolutional neural networks (CNNs) with an entirely attention-based mechanism.

The Transformer model uses self-attention to compute representations of input sequences, which allows it to capture long-term dependencies and parallelize computation effectively. The authors demonstrate that their model achieves state-of-the-art performance on several machine translation tasks and outperforms previous models that rely on RNNs or CNNs.

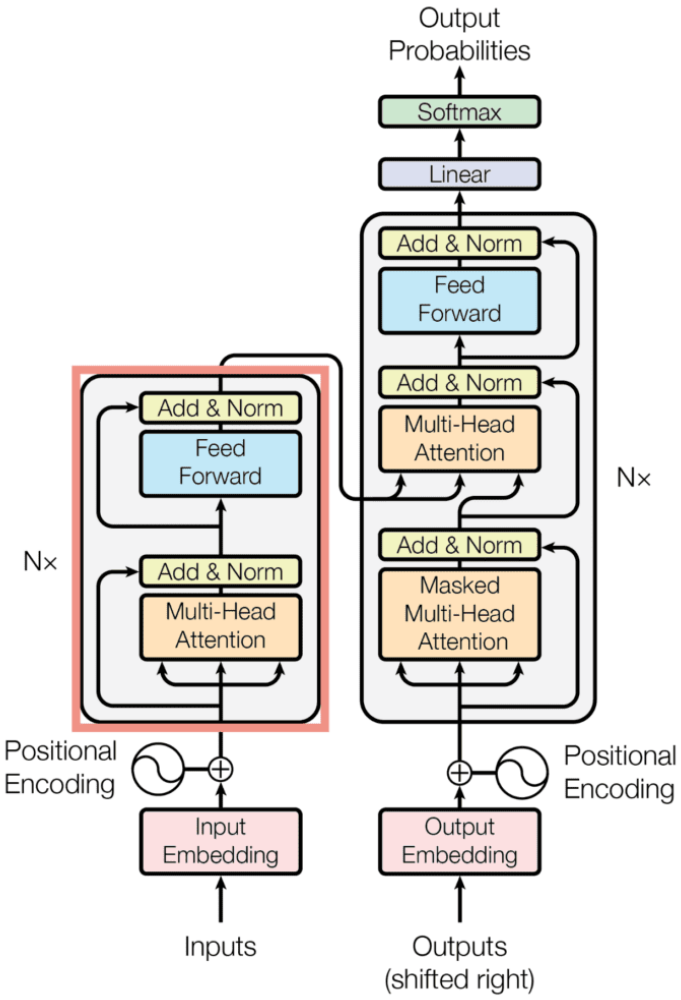


The Transformer architecture consists of an encoder and a decoder, each of which is composed of several layers. Each layer consists of two sub-layers: a multi-head self-attention mechanism and a feed-forward neural network. The multi-head self-attention mechanism allows the model to attend to different parts of the input sequence, while the feed-forward network

applies a point-wise fully connected layer to each position separately and identically.

The Transformer model also uses residual connections and layer normalization to facilitate training and prevent overfitting. In addition, the authors introduce a positional encoding scheme that encodes the position of each token in the input sequence, enabling the model to capture the order of the sequence without the need for recurrent or convolutional operations.

The Encoder



The encoder consists of a stack of N= 6 identical layers, where each layer is composed of two sublayers:

The first sublayer implements a multi-head self-attention mechanism. You have seen that the multi-head mechanism implements *h* heads that receive a (different) linearly projected version of the queries, keys, and values, each to produce *h* outputs in parallel that are then used to generate a final result.

The second sublayer is a fully connected feed-forward network consisting of two linear transformations with Rectified Linear Unit (ReLU) activation in between The six layers of the Transformer encoder apply the same linear transformations to all the words in the input sequence, but each layer employs different weight and bias parameters to do so.

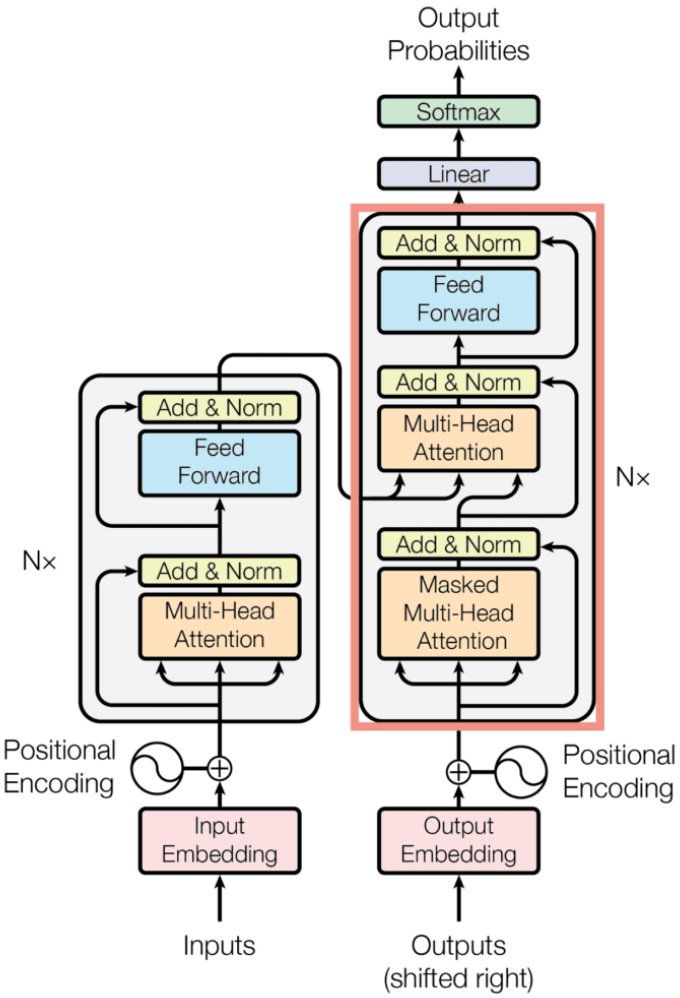
Furthermore, each of these two sublayers has a residual connection around it.

Each sublayer is also succeeded by a normalization layer, layernorm(.), which normalizes the sum computed between the sublayer input, and the output generated by the sublayer itself, sublayer.

An important consideration to keep in mind is that the Transformer architecture cannot inherently capture any information about the relative positions of the words in the sequence since it does not make use of recurrence. This information has to be injected by introducing positional encodings to the input embeddings.

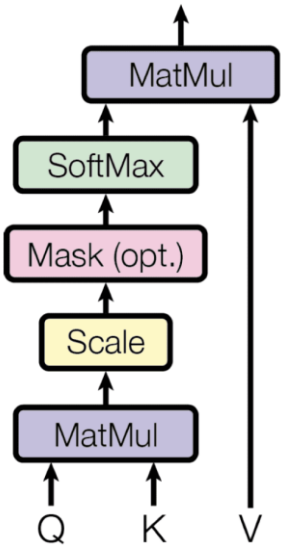
The positional encoding vectors are of the same dimension as the input embeddings and are generated using sine and cosine functions of different frequencies. Then, they are simply summed to the input embeddings in order to inject the positional information.

The Decoder



The decoder also consists of a stack of N = 6 identical layers that are each composed of three sublayers:

1. The first sublayer receives the previous output of the decoder stack, augments it with positional information, and implements multi-head self-attention over it. While the encoder is designed to attend to all words in the input sequence *regardless* of their position in the sequence, the decoder is modified to attend *only* to the preceding words. Hence, the prediction for a word at position can only depend on the known outputs for the words that come before it in the sequence. In the multi-head attention mechanism (which implements multiple, single attention functions in parallel), this is achieved by introducing a mask over the values produced by the scaled multiplication of matrices .



The multi-head attention in the decoder implements several masked, single-attention functions

The second layer implements a multi-head self-attention mechanism similar to the one implemented in the first sublayer of the encoder. On the decoder side, this multi-head mechanism receives the queries from the previous decoder sublayer and the keys and values from the output of the encoder. This allows the decoder to attend to all the words in the input sequence.

3. The third layer implements a fully connected feed-forward network, similar to the one implemented in the second sublayer of the encoder.

Furthermore, the three sublayers on the decoder side also have residual connections around them and are succeeded by a normalization layer.

Positional encodings are also added to the input embeddings of the decoder in the same manner as previously explained for the encoder.

Sum Up: The Transformer Model

The Transformer model runs as follows:

1. Each word forming an input sequence is transformed into a model-dimensional embedding vector.
2. Each embedding vector representing an input word is augmented by summing it (element-wise) to a positional encoding vector of the same model length, hence introducing positional information into the input.
3. The augmented embedding vectors are fed into the encoder block consisting of the two

sublayers explained above. Since the encoder attends to all words in the input sequence, irrespective if they precede or succeed the word under consideration, then the Transformer encoder is *bidirectional*.

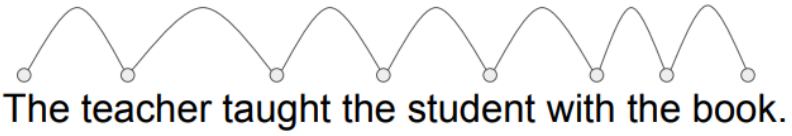
4. The decoder receives as input its own predicted output word at time-step.
5. The input to the decoder is also augmented by positional encoding in the same manner done on the encoder side.
6. The augmented decoder input is fed into the three sublayers comprising the decoder block explained above. Masking is applied in the first sublayer in order to stop the decoder from attending to the succeeding words. At the second sublayer, the decoder also receives the output of the encoder, which now allows the decoder to attend to all the words in the input sequence.
7. The output of the decoder finally passes through a fully connected layer, followed by a softmax layer, to generate a prediction for the next word of the output sequence.

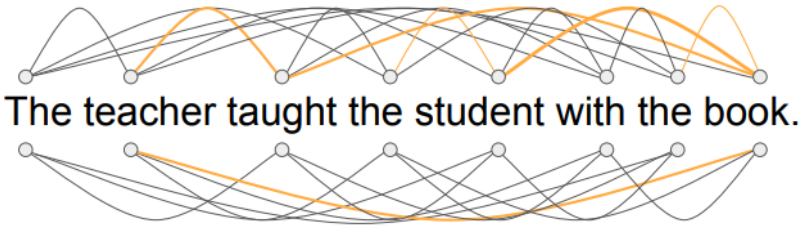
Comparison to Recurrent and Convolutional Layers

explain that their motivation for abandoning the use of recurrence and convolutions was based on several factors:

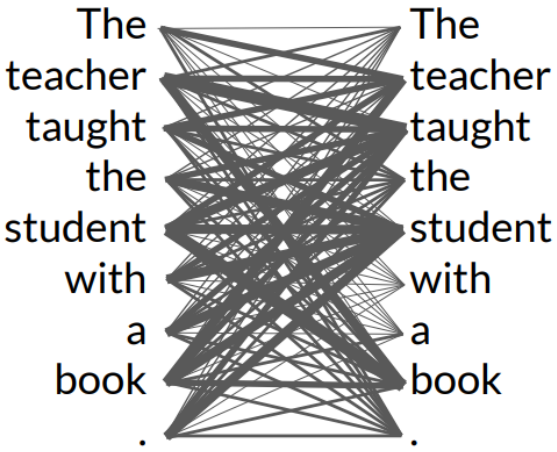
1. Self-attention layers were found to be faster than recurrent layers for shorter sequence lengths and can be restricted to consider only a neighborhood in the input sequence for very long sequence lengths.
2. The number of sequential operations required by a recurrent layer is based on the sequence length, whereas this number remains constant for a self-attention layer.
3. In convolutional neural networks, the kernel width directly affects the long-term dependencies that can be established between pairs of input and output positions. Tracking long-term dependencies would require using large kernels or stacks of convolutional layers that could increase the computational cost.

Transformers





Self-attention



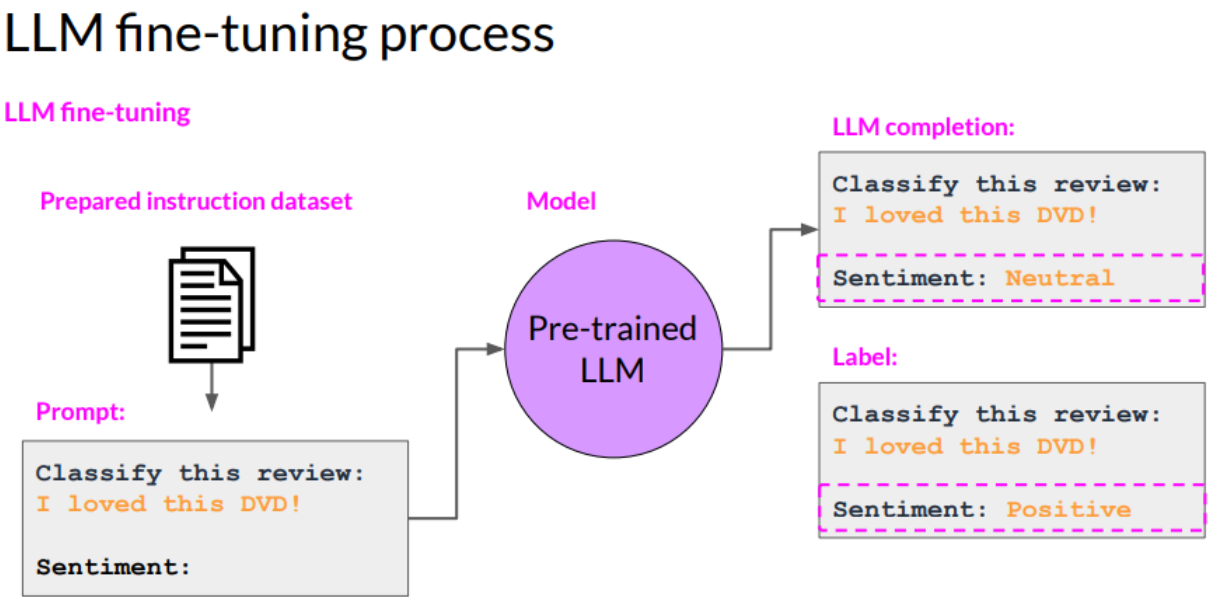
FINE TUNING

Fine-tuning involves updating the weights of a pre-trained language model on a new task and dataset.

The key distinction between training and fine-tuning is that training starts from scratch with a randomly initialized model dedicated to a particular task and dataset. On the other hand, fine-tuning adds to a pre-trained model and modifies its weights to achieve better performance.

Take the task of performing a sentiment analysis on movie reviews as an illustration. Instead of training a model from scratch, you may leverage a pre-trained language model such as GPT-3 that has already been trained on a vast corpus of text. To fine-tune the model for the specific goal of sentiment analysis, you would use a smaller dataset of movie reviews. This way, the model can be trained to analyze sentiment.

Compared to starting from zero, fine-tuning has a number of benefits, including a shorter training period and the capacity to produce cutting-edge outcomes with less data.



TOOLS:-

1. VSCODE: 

Visual Studio Code (VS Code) is a free and widely used source code editor developed by Microsoft. It is designed for developers to write, edit, and debug code across various programming languages. VS Code offers a lightweight and highly customizable environment, equipped with features like syntax highlighting, intelligent code completion, and code navigation, which enhance productivity.

One of the standout features is its extensive extension marketplace, enabling users to tailor their coding experience through various extensions for languages, frameworks, and tools. Its integrated version control system supports Git, making collaborative coding and code management seamless. Debugging capabilities, integrated terminal, and task automation further simplify the development process.

Overall, Visual Studio Code is a versatile code editor that caters to programmers from different domains,promoting efficiency, collaboration, and a smooth coding experience.

2. GOOGLE COLAB: 

Google Colab, developed by Google, is a cloud-based platform revolutionizing data science and machine learning collaboration. Operating as a Jupyter Notebook in the cloud, it eliminates the need for local installations and seamlessly integrates with Google Drive. This collaborative environment offers real-time editing, commenting, and simultaneous collaboration, enhancing teamwork on data science projects.

Colab provides access to powerful computing resources, including GPUs and TPUs, accelerating tasks like machine learning model training. With support for popular libraries and frameworks, it offers a versatile coding space for data scientists and machine learning enthusiasts. Its integration with Google's ecosystem, coupled with easy access to GitHub, promotes efficient version control and collaborative coding practices.

In essence, Google Colab is a dynamic platform that streamlines collaborative coding, making data science and machine learning endeavors more accessible, efficient, and interactive.

3. HUGGING FACE TRANSFORMERS:

Hugging Face Transformers is a leading toolkit in the field of natural language processing (NLP), offering a versatile set of tools and pre-trained models. Developed as an open-source library, it simplifies the integration of state-of-the-art transformer-based models, renowned for their effectiveness in capturing contextual nuances in text. The toolkit's model hub provides easy access to a diverse range of pre-trained models, fostering efficiency in tasks like text classification, language translation, and sentiment analysis. With a user-friendly interface and support for popular transformer architectures, Hugging Face Transformers has become a go-to resource for both seasoned researchers and developers new to NLP. Its collaborative model hub and community-driven approach further contribute to the advancement of NLP research and development, making it an indispensable tool for innovation in language understanding and generation.

4. PYTORCH:

PyTorch, developed by Facebook's AI Research lab, stands out as a dynamic and user-friendly toolkit for deep learning. Its distinctive feature is a dynamic computational graph that allows on-the-fly model modification, making it intuitive and flexible for model design and experimentation. With an easy-to-understand syntax and support for imperative programming, PyTorch is accessible to both beginners and seasoned practitioners.

The framework's modular design and extensive ecosystem, including specialized libraries like torchvision and torchaudio, contribute to its versatility across various machine learning domains. PyTorch seamlessly integrates with hardware accelerators, optimizing performance on GPUs and TPUs. Its efficient deployment tools, such as TorchServe and TorchScript, make it a preferred choice for transitioning models from research to real-world applications.

In essence, PyTorch's simplicity, flexibility, and deployment-friendly features have established it as a cornerstone in the deep learning community, catering to a wide range of users and applications.

CODES

```
%pip install --upgrade pip
%pip install --disable-pip-version-check \
    torch==1.13.1 \
    torchdata==0.5.1 --quiet

%pip install \
    transformers==4.27.2 \
    datasets==2.11.0  --quiet

from datasets import load_dataset
from transformers import AutoModelForSeq2SeqLM
from transformers import AutoTokenizer
from transformers import GenerationConfig

huggingface_dataset_name = "knkarthick/dialogsum"

dataset = load_dataset(huggingface_dataset_name)

example_indices = [40, 200]

dash_line = '-'.join('' for x in range(100))

for i, index in enumerate(example_indices):
    print(dash_line)
    print('Example ', i + 1)
    print(dash_line)
    print('INPUT DIALOGUE:')
    print(dataset['test'][index]['dialogue'])
    print(dash_line)
    print('BASELINE HUMAN SUMMARY:')
    print(dataset['test'][index]['summary'])
    print(dash_line)
    print()

model_name='google/flan-t5-base'

model = AutoModelForSeq2SeqLM.from_pretrained(model_name)

tokenizer = AutoTokenizer.from_pretrained(model_name, use_fast=True)
```

```
sentence = "What time is it, Tom?"

sentence_encoded = tokenizer(sentence, return_tensors='pt')

sentence_decoded = tokenizer.decode(
        sentence_encoded["input_ids"][0],
        skip_special_tokens=True
    )

print('ENCODED SENTENCE:')
print(sentence_encoded["input_ids"][0])
print('\nDECODED SENTENCE:')
print(sentence_decoded)

for i, index in enumerate(example_indices):
    dialogue = dataset['test'][index]['dialogue']
    summary = dataset['test'][index]['summary']

    inputs = tokenizer(dialogue, return_tensors='pt')
    output = tokenizer.decode(
        model.generate(
            inputs["input_ids"],
            max_new_tokens=50,
        )[0],
        skip_special_tokens=True
    )

    print(dash_line)
    print('Example ', i + 1)
    print(dash_line)
    print(f'INPUT PROMPT:\n{dialogue}')
    print(dash_line)
    print(f'BASELINE HUMAN SUMMARY:\n{summary}')
    print(dash_line)
    print(f'MODEL GENERATION - WITHOUT PROMPT ENGINEERING:\n{output}\n')

for i, index in enumerate(example_indices):
    dialogue = dataset['test'][index]['dialogue']
    summary = dataset['test'][index]['summary']

    prompt = f"""
Summarize the following conversation.
```

```
{dialogue}

Summary:
    """

     Input constructed prompt instead of the dialogue.
    inputs = tokenizer(prompt, return_tensors='pt')
    output = tokenizer.decode(
        model.generate(
            inputs["input_ids"],
            max_new_tokens=50,
        )[0],
        skip_special_tokens=True
    )

    print(dash_line)
    print('Example ', i + 1)
    print(dash_line)
    print(f'INPUT PROMPT:\n{prompt}')
    print(dash_line)
    print(f'BASELINE HUMAN SUMMARY:\n{summary}')
    print(dash_line)
    print(f'MODEL GENERATION - ZERO SHOT:\n{output}\n')

for i, index in enumerate(example_indices):
    dialogue = dataset['test'][index]['dialogue']
    summary = dataset['test'][index]['summary']

    prompt = f"""
Dialogue:

{dialogue}

What was going on?
"""

    inputs = tokenizer(prompt, return_tensors='pt')
    output = tokenizer.decode(
        model.generate(
            inputs["input_ids"],
            max_new_tokens=50,
```

```
            )[0],
            skip_special_tokens=True
        )

        print(dash_line)
        print('Example ', i + 1)
        print(dash_line)
        print(f'INPUT PROMPT:\n{prompt}')
        print(dash_line)
        print(f'BASELINE HUMAN SUMMARY:\n{summary}\n')
        print(dash_line)
        print(f'MODEL GENERATION - ZERO SHOT:\n{output}\n')

    def make_prompt(example_indices_full, example_index_to_summarize):
        prompt = ''
        for index in example_indices_full:
            dialogue = dataset['test'][index]['dialogue']
            summary = dataset['test'][index]['summary']
```

The stop sequence '{summary}\n\n\n' is important for FLAN-T5. Other models may have their own preferred stop sequence.

```
            prompt += f"""
    Dialogue:

    {dialogue}

    What was going on?
    {summary}


    """

        dialogue = dataset['test'][example_index_to_summarize]['dialogue']

        prompt += f"""
    Dialogue:

    {dialogue}

    What was going on?
    """

        return prompt
```

```
example_indices_full = [40]
example_index_to_summarize = 200

one_shot_prompt = make_prompt(example_indices_full, example_index_to_summarize)

print(one_shot_prompt)

summary = dataset['test'][example_index_to_summarize]['summary']

inputs = tokenizer(one_shot_prompt, return_tensors='pt')
output = tokenizer.decode(
    model.generate(
        inputs["input_ids"],
        max_new_tokens=50,
    )[0],
    skip_special_tokens=True
)

print(dash_line)
print(f'BASELINE HUMAN SUMMARY:\n{summary}\n')
print(dash_line)
print(f'MODEL GENERATION - ONE SHOT:\n{output}')

example_indices_full = [40, 80, 120]
example_index_to_summarize = 200

few_shot_prompt = make_prompt(example_indices_full, example_index_to_summarize)

print(few_shot_prompt)

summary = dataset['test'][example_index_to_summarize]['summary']

inputs = tokenizer(few_shot_prompt, return_tensors='pt')
output = tokenizer.decode(
    model.generate(
        inputs["input_ids"],
        max_new_tokens=50,
    )[0],
    skip_special_tokens=True
)
```

```
print(dash_line)
print(f'BASELINE HUMAN SUMMARY:\n{summary}\n')
print(dash_line)
print(f'MODEL GENERATION - FEW SHOT:\n{output}')

 generation_config = GenerationConfig(max_new_tokens=50)
 generation_config = GenerationConfig(max_new_tokens=10)
 generation_config = GenerationConfig(max_new_tokens=50, do_sample=True, temperature=0.1)
generation_config = GenerationConfig(max_new_tokens=50, do_sample=True, temperature=0.5)
 generation_config = GenerationConfig(max_new_tokens=50, do_sample=True, temperature=1.0)

inputs = tokenizer(few_shot_prompt, return_tensors='pt')
output = tokenizer.decode(
    model.generate(
        inputs["input_ids"],
        generation_config=generation_config,
    )[0],
    skip_special_tokens=True
)

print(dash_line)
print(f'MODEL GENERATION - FEW SHOT:\n{output}')
print(dash_line)
print(f'BASELINE HUMAN SUMMARY:\n{summary}\n')

 Fine-Tuning

%pip install \
    evaluate==0.4.0 \
    rouge_score==0.1.2 \
    loralib==0.1.1 \
    peft==0.3.0 --quiet

import torch
import time
import evaluate
import pandas as pd
import numpy as np

original_model = AutoModelForSeq2SeqLM.from_pretrained(model_name,
torch_dtype=torch.bfloat16)
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

```
    def print_number_of_trainable_model_parameters(model):
        trainable_model_params = 0
        all_model_params = 0
        for _, param in model.named_parameters():
            all_model_params += param.numel()
            if param.requires_grad:
                trainable_model_params += param.numel()
        return f"trainable model parameters: {trainable_model_params}\nall model parameters: {all_model_params}\npercentage of trainable model parameters: {100 * trainable_model_params / all_model_params:.2f}%"

    print(print_number_of_trainable_model_parameters(original_model))

    index = 200

    dialogue = dataset['test'][index]['dialogue']
    summary = dataset['test'][index]['summary']

    prompt = f"""
    Summarize the following conversation.

    {dialogue}

    Summary:
    """

    inputs = tokenizer(prompt, return_tensors='pt')
    output = tokenizer.decode(
        original_model.generate(
            inputs["input_ids"],
            max_new_tokens=200,
        )[0],
        skip_special_tokens=True
    )

    dash_line = '-'.join('' for x in range(100))
    print(dash_line)
    print(f'INPUT PROMPT:\n{prompt}')
    print(dash_line)
    print(f'BASELINE HUMAN SUMMARY:\n{summary}\n')
    print(dash_line)
    print(f'MODEL GENERATION - ZERO SHOT:\n{output}')
```

```
    def tokenize_function(example):
        start_prompt = 'Summarize the following conversation.\n\n'
        end_prompt = '\n\nSummary: '
        prompt = [start_prompt + dialogue + end_prompt for dialogue in example["dialogue"]]
        example['input_ids'] = tokenizer(prompt, padding="max_length", truncation=True,
return_tensors="pt").input_ids
        example['labels'] = tokenizer(example["summary"], padding="max_length", truncation=True,
return_tensors="pt").input_ids

        return example

     The dataset actually contains 3 diff splits: train, validation, test.
     The tokenize_function code is handling all data across all splits in batches.
    tokenized_datasets = dataset.map(tokenize_function, batched=True)
    tokenized_datasets = tokenized_datasets.remove_columns(['id', 'topic', 'dialogue', 'summary',])

    tokenized_datasets = tokenized_datasets.filter(lambda example, index: index % 100 == 0,
with_indices=True)

    print(f"Shapes of the datasets:")
    print(f"Training: {tokenized_datasets['train'].shape}")
    print(f"Validation: {tokenized_datasets['validation'].shape}")
    print(f"Test: {tokenized_datasets['test'].shape}")

    print(tokenized_datasets)

    from transformers import GenerationConfig, TrainingArguments, Trainer

    output_dir = f'./dialogue-summary-training-{str(int(time.time()))}'

    training_args = TrainingArguments(
        output_dir=output_dir,
        learning_rate=1e-5,
        num_train_epochs=1,
        weight_decay=0.01,
        logging_steps=1,
        max_steps=1
    )

    trainer = Trainer(
        model=original_model,
        args=training_args,
        train_dataset=tokenized_datasets['train'],
```

```
    eval_dataset=tokenized_datasets['validation']
)

trainer.train()

 !aws s3 cp --recursive s3://dlai-generative-ai/models/flan-dialogue-summary-checkpoint/ ./flan-dialogue-summary-checkpoint/

original_model.save_pretrained("fine-tuned_gpt2")

ls


 !ls -alh ./flan-dialogue-summary-checkpoint/pytorch_model.bin

!ls  /content/fine-tuned_gpt2/pytorch_model.bin

instruct_model = AutoModelForSeq2SeqLM.from_pretrained("/content/fine-tuned_gpt2", torch_dtype=torch.bfloat16)

 instruct_model = AutoModelForSeq2SeqLM.from_pretrained("./flan-dialogue-summary-checkpoint", torch_dtype=torch.bfloat16)

index = 200
dialogue = dataset['test'][index]['dialogue']
human_baseline_summary = dataset['test'][index]['summary']

prompt = f"""
Summarize the following conversation.

{dialogue}

Summary:
"""
 import torch

 ...

 Move the input tensor to the same device as the model
input_ids = tokenizer(prompt, return_tensors="pt").input_ids.to(original_model.device)

 ...
```

```
original_model_outputs = original_model.generate(
    input_ids=input_ids,
    generation_config=GenerationConfig(max_new_tokens=200, num_beams=1)
)

 ...

 input_ids = tokenizer(prompt, return_tensors="pt").input_ids

 original_model_outputs = original_model.generate(input_ids=input_ids,
generation_config=GenerationConfig(max_new_tokens=200, num_beams=1))
original_model_text_output = tokenizer.decode(original_model_outputs[0],
skip_special_tokens=True)

input_ids_instruct = input_ids.to(instruct_model.device)

instruct_model_outputs = instruct_model.generate(
    input_ids=input_ids_instruct,
    generation_config=GenerationConfig(max_new_tokens=200, num_beams=1)
)
 instruct_model_outputs = instruct_model.generate(input_ids=input_ids,
generation_config=GenerationConfig(max_new_tokens=200, num_beams=1))
instruct_model_text_output = tokenizer.decode(instruct_model_outputs[0],
skip_special_tokens=True)

print(dash_line)
print(f'BASELINE HUMAN SUMMARY:\n{human_baseline_summary}')
print(dash_line)
print(f'ORIGINAL MODEL:\n{original_model_text_output}')
print(dash_line)
print(f'INSTRUCT MODEL:\n{instruct_model_text_output}')

rouge = evaluate.load('rouge')

dialogues = dataset['test'][0:10]['dialogue']
human_baseline_summaries = dataset['test'][0:10]['summary']

original_model_summaries = []
instruct_model_summaries = []

for _, dialogue in enumerate(dialogues):
    prompt = f"""
Summarize the following conversation.
```

```
        {dialogue}

        Summary: """
           input_ids = tokenizer(prompt, return_tensors="pt").input_ids
            Move the input tensor to the same device as the model
           input_ids = tokenizer(prompt, return_tensors="pt").input_ids.to(original_model.device)

         ...

           original_model_outputs = original_model.generate(
              input_ids=input_ids,
              generation_config=GenerationConfig(max_new_tokens=200, num_beams=1)
            )

            original_model_outputs = original_model.generate(input_ids=input_ids,
generation_config=GenerationConfig(max_new_tokens=200))
           original_model_text_output = tokenizer.decode(original_model_outputs[0],
skip_special_tokens=True)
           original_model_summaries.append(original_model_text_output)

           input_ids_instruct = input_ids.to(instruct_model.device)

           instruct_model_outputs = instruct_model.generate(
             input_ids=input_ids_instruct,
              generation_config=GenerationConfig(max_new_tokens=200, num_beams=1)
           )
            instruct_model_outputs = instruct_model.generate(input_ids=input_ids,
generation_config=GenerationConfig(max_new_tokens=200))
           instruct_model_text_output = tokenizer.decode(instruct_model_outputs[0],
skip_special_tokens=True)
           instruct_model_summaries.append(instruct_model_text_output)

        zipped_summaries = list(zip(human_baseline_summaries, original_model_summaries,
instruct_model_summaries))

        df = pd.DataFrame(zipped_summaries, columns = ['human_baseline_summaries',
'original_model_summaries', 'instruct_model_summaries'])
        df

        original_model_results = rouge.compute(
           predictions=original_model_summaries,
           references=human_baseline_summaries[0:len(original_model_summaries)],
           use_aggregator=True,
```

```
    use_stemmer=True,
)

instruct_model_results = rouge.compute(
    predictions=instruct_model_summaries,
    references=human_baseline_summaries[0:len(instruct_model_summaries)],
    use_aggregator=True,
    use_stemmer=True,
)

print('ORIGINAL MODEL:')
print(original_model_results)
print('INSTRUCT MODEL:')
print(instruct_model_results)

 results = pd.read_csv("data/dialogue-summary-training-results.csv")

 human_baseline_summaries = results['human_baseline_summaries'].values
 original_model_summaries = results['original_model_summaries'].values
 instruct_model_summaries = results['instruct_model_summaries'].values

 original_model_results = rouge.compute(
     predictions=original_model_summaries,
     references=human_baseline_summaries[0:len(original_model_summaries)],
     use_aggregator=True,
     use_stemmer=True,
 )

 instruct_model_results = rouge.compute(
     predictions=instruct_model_summaries,
     references=human_baseline_summaries[0:len(instruct_model_summaries)],
     use_aggregator=True,
     use_stemmer=True,
 )

 print('ORIGINAL MODEL:')
 print(original_model_results)
 print('INSTRUCT MODEL:')
 print(instruct_model_results)

print("Absolute percentage improvement of INSTRUCT MODEL over ORIGINAL MODEL")
```

```
        improvement = (np.array(list(instruct_model_results.values())) -
np.array(list(original_model_results.values())))
        for key, value in zip(instruct_model_results.keys(), improvement):
            print(f'{key}: {value*100:.2f}%')

        from peft import LoraConfig, get_peft_model, TaskType

        lora_config = LoraConfig(
            r=32,  Rank
            lora_alpha=32,
            target_modules=["q", "v"],
            lora_dropout=0.05,
            bias="none",
            task_type=TaskType.SEQ_2_SEQ_LM  FLAN-T5
        )

        peft_model = get_peft_model(original_model,
                                    lora_config)
        print(print_number_of_trainable_model_parameters(peft_model))

        output_dir = f'./peft-dialogue-summary-training-{str(int(time.time()))}'

        peft_training_args = TrainingArguments(
            output_dir=output_dir,
            auto_find_batch_size=True,
            learning_rate=1e-3,  Higher learning rate than full fine-tuning.
            num_train_epochs=1,
            logging_steps=1,
            max_steps=1
        )

        peft_trainer = Trainer(
            model=peft_model,
            args=peft_training_args,
            train_dataset=tokenized_datasets["train"],
        )

        peft_trainer.train()

        peft_model_path="./peft-dialogue-summary-checkpoint-local"

        peft_trainer.model.save_pretrained(peft_model_path)
        tokenizer.save_pretrained(peft_model_path)
```

```
!aws s3 cp --recursive s3://dlai-generative-ai/models/peft-dialogue-summary-checkpoint/ ./peft-dialogue-summary-checkpoint-from-s3/
```

```
!ls -al ./peft-dialogue-summary-checkpoint-from-s3/adapter_model.bin
```

```
from peft import PeftModel, PeftConfig

peft_model_base = AutoModelForSeq2SeqLM.from_pretrained("google/flan-t5-base", torch_dtype=torch.bfloat16)
tokenizer = AutoTokenizer.from_pretrained("google/flan-t5-base")

peft_model = PeftModel.from_pretrained(peft_model_base,
                                       './peft-dialogue-summary-checkpoint-from-s3/',
                                       torch_dtype=torch.bfloat16,
                                       is_trainable=False)

print(print_number_of_trainable_model_parameters(peft_model))

index = 200
dialogue = dataset['test'][index]['dialogue']
baseline_human_summary = dataset['test'][index]['summary']

prompt = f"""
Summarize the following conversation.

{dialogue}

Summary: """

input_ids = tokenizer(prompt, return_tensors="pt").input_ids

original_model_outputs = original_model.generate(input_ids=input_ids, generation_config=GenerationConfig(max_new_tokens=200, num_beams=1))
original_model_text_output = tokenizer.decode(original_model_outputs[0], skip_special_tokens=True)

instruct_model_outputs = instruct_model.generate(input_ids=input_ids, generation_config=GenerationConfig(max_new_tokens=200, num_beams=1))
instruct_model_text_output = tokenizer.decode(instruct_model_outputs[0], skip_special_tokens=True)
```

```
        peft_model_outputs = peft_model.generate(input_ids=input_ids,
generation_config=GenerationConfig(max_new_tokens=200, num_beams=1))
        peft_model_text_output = tokenizer.decode(peft_model_outputs[0], skip_special_tokens=True)

        print(dash_line)
        print(f'BASELINE HUMAN SUMMARY:\n{human_baseline_summary}')
        print(dash_line)
        print(f'ORIGINAL MODEL:\n{original_model_text_output}')
        print(dash_line)
        print(f'INSTRUCT MODEL:\n{instruct_model_text_output}')
        print(dash_line)
        print(f'PEFT MODEL: {peft_model_text_output}')

        dialogues = dataset['test'][0:10]['dialogue']
        human_baseline_summaries = dataset['test'][0:10]['summary']

        original_model_summaries = []
        instruct_model_summaries = []
        peft_model_summaries = []

        for idx, dialogue in enumerate(dialogues):
            prompt = f"""
        Summarize the following conversation.

        {dialogue}

        Summary: """

            input_ids = tokenizer(prompt, return_tensors="pt").input_ids

            human_baseline_text_output = human_baseline_summaries[idx]

            original_model_outputs = original_model.generate(input_ids=input_ids,
generation_config=GenerationConfig(max_new_tokens=200))
            original_model_text_output = tokenizer.decode(original_model_outputs[0],
skip_special_tokens=True)

            instruct_model_outputs = instruct_model.generate(input_ids=input_ids,
generation_config=GenerationConfig(max_new_tokens=200))
            instruct_model_text_output = tokenizer.decode(instruct_model_outputs[0],
skip_special_tokens=True)
```

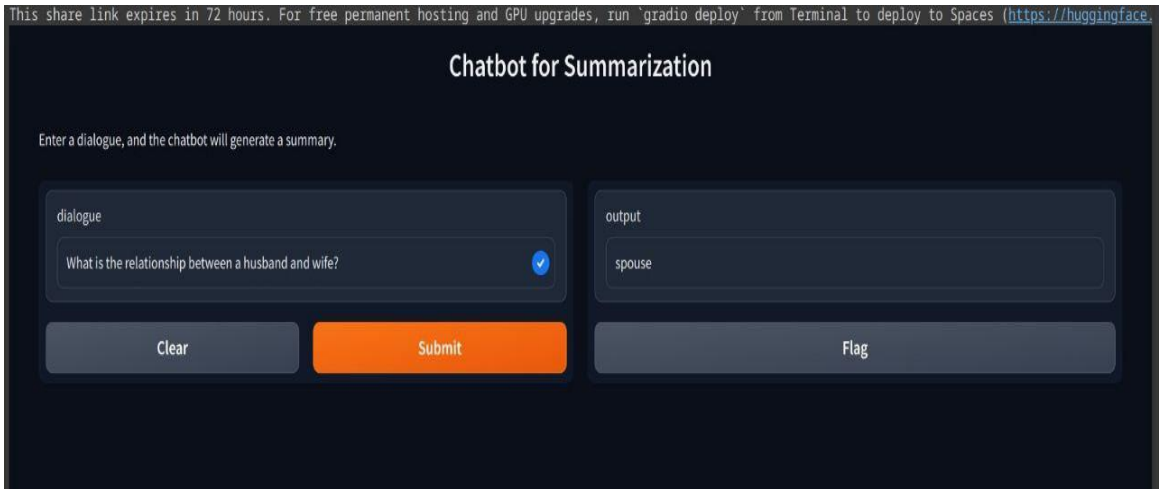
```
        peft_model_outputs = peft_model.generate(input_ids=input_ids,
generation_config=GenerationConfig(max_new_tokens=200))
        peft_model_text_output = tokenizer.decode(peft_model_outputs[0], skip_special_tokens=True)

        original_model_summaries.append(original_model_text_output)
        instruct_model_summaries.append(instruct_model_text_output)
        peft_model_summaries.append(peft_model_text_output)

    zipped_summaries = list(zip(human_baseline_summaries, original_model_summaries,
instruct_model_summaries, peft_model_summaries))

    df = pd.DataFrame(zipped_summaries, columns = ['human_baseline_summaries',
'original_model_summaries', 'instruct_model_summaries', 'peft_model_summaries'])
    df

    rouge = evaluate.load('rouge')

    original_model_results = rouge.compute(
        predictions=original_model_summaries,
        references=human_baseline_summaries[0:len(original_model_summaries)],
        use_aggregator=True,
        use_stemmer=True,
    )

    instruct_model_results = rouge.compute(
        predictions=instruct_model_summaries,
        references=human_baseline_summaries[0:len(instruct_model_summaries)],
        use_aggregator=True,
        use_stemmer=True,
    )

    peft_model_results = rouge.compute(
        predictions=peft_model_summaries,
        references=human_baseline_summaries[0:len(peft_model_summaries)],
        use_aggregator=True,
        use_stemmer=True,
    )

    print('ORIGINAL MODEL:')
    print(original_model_results)
    print('INSTRUCT MODEL:')
    print(instruct_model_results)
    print('PEFT MODEL:')
```

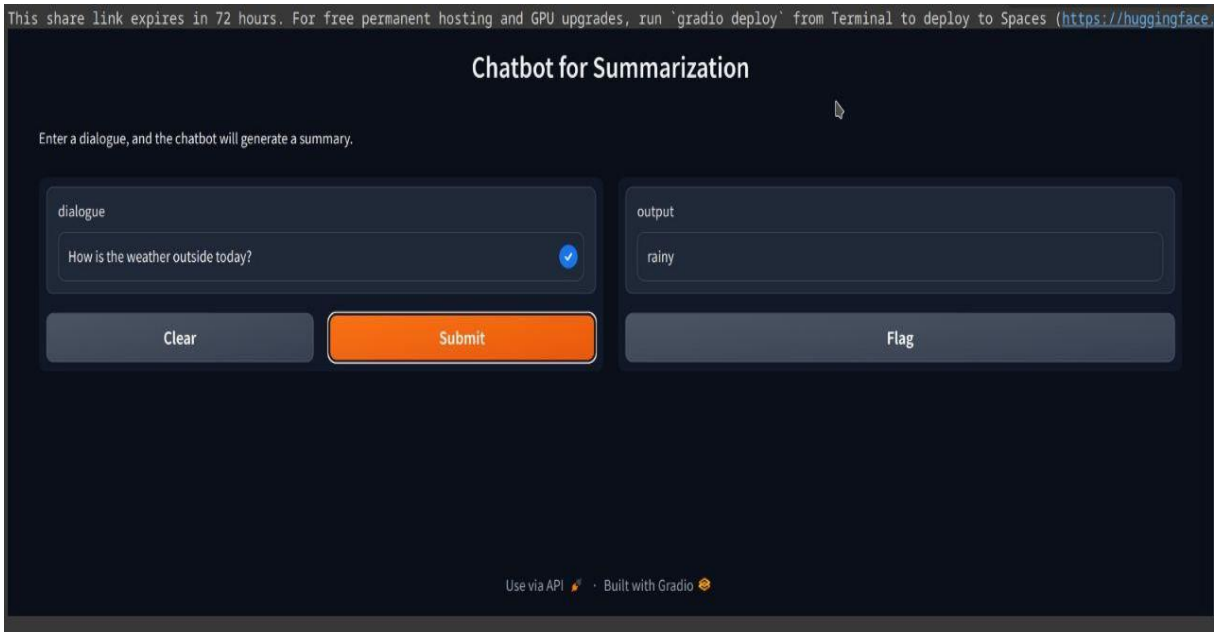
```
print(peft_model_results)

human_baseline_summaries = results['human_baseline_summaries'].values
original_model_summaries = results['original_model_summaries'].values
instruct_model_summaries = results['instruct_model_summaries'].values
peft_model_summaries     = results['peft_model_summaries'].values

original_model_results = rouge.compute(
    predictions=original_model_summaries,
    references=human_baseline_summaries[0:len(original_model_summaries)],
    use_aggregator=True,
    use_stemmer=True,
)

instruct_model_results = rouge.compute(
    predictions=instruct_model_summaries,
    references=human_baseline_summaries[0:len(instruct_model_summaries)],
    use_aggregator=True,
    use_stemmer=True,
)

peft_model_results = rouge.compute(
    predictions=peft_model_summaries,
    references=human_baseline_summaries[0:len(peft_model_summaries)],
    use_aggregator=True,
    use_stemmer=True,
)

print('ORIGINAL MODEL:')
print(original_model_results)
print('INSTRUCT MODEL:')
print(instruct_model_results)
print('PEFT MODEL:')
print(peft_model_results)

print("Absolute percentage improvement of PEFT MODEL over ORIGINAL MODEL")

improvement = (np.array(list(peft_model_results.values())) -
np.array(list(original_model_results.values())))
for key, value in zip(peft_model_results.keys(), improvement):
    print(f'{key}: {value*100:.2f}%')

print("Absolute percentage improvement of PEFT MODEL over INSTRUCT MODEL")
```

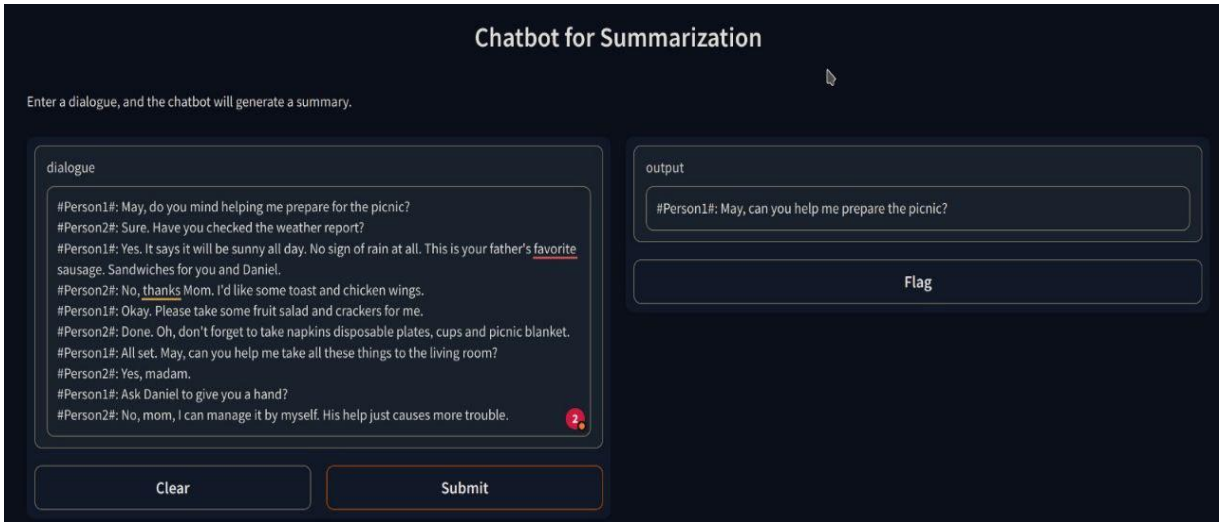
```
 improvement = (np.array(list(peft_model_results.values())) -
np.array(list(instruct_model_results.values())))
 for key, value in zip(peft_model_results.keys(), improvement):
    print(f'{key}: {value*100:.2f}%')

import locale
def getpreferredencoding(do_setlocale = True):
   return "UTF-8"
locale.getpreferredencoding = getpreferredencoding

pip install --upgrade pip

pip install gradio

pip install gradio==2.7.1

import gradio as gr

 Your existing code for loading the model and tokenizer

def generate_summary(dialogue):
   inputs = tokenizer(dialogue, return_tensors='pt')
   output = tokenizer.decode(
      model.generate(
         inputs["input_ids"],
         max_new_tokens=50,
      )[0],
      skip_special_tokens=True
   )
   return output




iface = gr.Interface(
   fn=generate_summary,
   inputs="text",
   outputs="text",
   live=False,
   title="Chatbot for Summarization",
   description="Enter a dialogue, and the chatbot will generate a summary.",
)
```

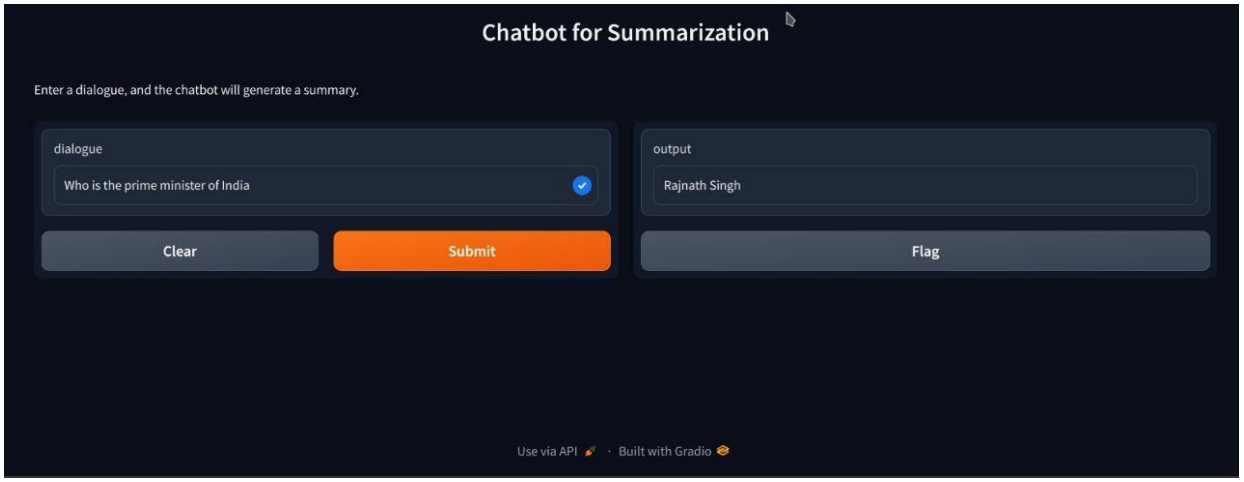
iface.launch()

Demonstration Screenshots:






Chatbot for Summarization
Enter a dialogue, and the chatbot will generate a summary.
dialogue
#Person1#: May, do you mind helping me prepare for the picnic?
#Person2#: Sure. Have you checked the weather report?
#Person1#: Yes. It says it will be sunny all day. No sign of rain at all. This is your father's favorite sausage. Sandwiches for you and Daniel.
#Person2#: No, thanks Mom. I'd like some toast and chicken wings.
#Person1#: Okay. Please take some fruit salad and crackers for me.
#Person2#: Done. Oh, don't forget to take napkins disposable plates, cups and picnic blanket.
#Person1#: All set. May, can you help me take all these things to the living room?
#Person2#: Yes, madam.
#Person1#: Ask Daniel to give you a hand?
#Person2#: No, mom, I can manage it by myself. His help just causes more trouble.
Clear
Submit
output
#Person1#: May, can you help me prepare the picnic?
Flag


Chatbot for Summarization
Enter a dialogue, and the chatbot will generate a summary.
dialogue
Who is the prime minister of India
Clear
Submit
output
Rajnath Singh
Flag
Use via API
Built with Gradio

Bibliography

i. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need.

ii. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2021). ReAct: Synergizing Reasoning and Acting in Language Models.

iii. Lester, B., Al-Rfou, R., & Constant, N. (2021). The Power of Scale for Parameter-Efficient Prompt Tuning. Google Research.

iv. Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., ... & Wei, J. (2021). Scaling Instruction-Finetuned Language Models. Google.

v. Bosma, M., & Wei, J. (2021, October 6). Introducing FLAN: More generalizable Language Models with Instruction Fine-Tuning. [Blog post]. Google Research.

vi. Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. R. (2018). GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. New York University, University of Washington, DeepMind.

vii. Lin, C. Y. (2004). ROUGE: A Package for Automatic Evaluation of Summaries. Information Sciences Institute, University of Southern California.

viii. Hu, E., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., ... & Chen, W. (2021). LORA: Low-Rank Adaptation of Large Language Models. Microsoft Corporation.

ix. Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., ... & Wei, J. (2021). Scaling Instruction-Finetuned Language Models. Google.