Detailed Report on the Audio Processing and Response Generation Model

#### 1. Introduction

This report provides an in-depth analysis of a sophisticated model designed to handle voice audio inputs, process them through various stages, and produce a coherent audio response. The model combines multiple advanced technologies, including Whisper for speech-to-text transcription, Gemini for natural language response generation, and Parler TTS for text-to-speech synthesis. The following sections describe each component of the model, its functionality, and its role in the overall workflow.

#### 2. Workflow Overview

The pipeline for processing voice audio involves several key stages:

1. Audio Conversion: Converts the input audio into a standardized format.

2. Voice Activity Detection (VAD)*: Filters out non-speech segments to retain only the voiced portions.

3. Audio Transcription: Transcribes the cleaned audio into text.

4. Response Generation: Generates a textual response based on the transcribed text.

5. Text-to-Speech (TTS): Converts the text response back into audio.

#### 3. Detailed Description

3.1. Audio Conversion

- Function: The `convert_audio` function standardizes the input audio file by converting it to a 16 kHz sampling rate and a mono channel.

- Tool Used: `ffmpeg`

- Process:

  - Command Execution: Executes a command-line instruction to transform the audio format. The `-ar` flag sets the sampling rate to 16 kHz, and the `-ac` flag sets the audio channel count to mono.

  - Purpose: Uniform audio format is essential for consistent processing across different stages. The 16 kHz sampling rate is chosen for its balance between audio quality and processing efficiency, while mono channel simplifies the audio data by removing the complexity of stereo sound.

3.2. Voice Activity Detection (VAD)

- Function: The `apply_vad` function applies Voice Activity Detection to remove silent or non-speech segments from the audio.

- Tool Used*: `webrtcvad`

- Process:

  - *Audio Frame Analysis: The audio is read and divided into frames. Each frame is evaluated to determine if it contains speech.

  - Voice Activity Detection: The `webrtcvad` library is used to identify frames that contain voiced audio. The VAD mode is set to 1, which provides a balance between sensitivity and accuracy.

  -Purpose: Removing silence improves the efficiency of transcription by focusing only on segments with actual speech. This also reduces the processing load and enhances the overall quality of the transcription.

3.3. Audio Transcription

- Function: The `transcribe_audio` function converts the VAD-processed audio into text.

- Tool Used: Whisper model

- Process:

  - Model Loading: The Whisper ASR model is loaded for transcription. The model is pre-trained to handle various accents, noise conditions, and speech patterns.

  - Transcription: The audio file is processed by Whisper to generate a text representation of the spoken content.

  - Purpose: Accurate transcription is crucial for generating meaningful responses. Whisper's advanced capabilities ensure high-quality text extraction even from noisy or challenging audio inputs.

3.4. Response Generation

- Function: The Gemini model generates a textual response based on the transcribed text.

- Tool Used: Gemini model

- Process:

  - Model Configuration: The model is initialized with an API key and started for chat interactions.

  - Response Generation: The transcribed text is sent to the Gemini model, which processes it and generates a coherent, contextually relevant response.

- Purpose: The response generation step provides a meaningful reply to the input query, leveraging advanced natural language understanding and generation capabilities.

3.5. Text-to-Speech (TTS)

- Function: The Parler TTS model converts the generated text response into audio.

- Tool Used: Parler TTS model

- Process:

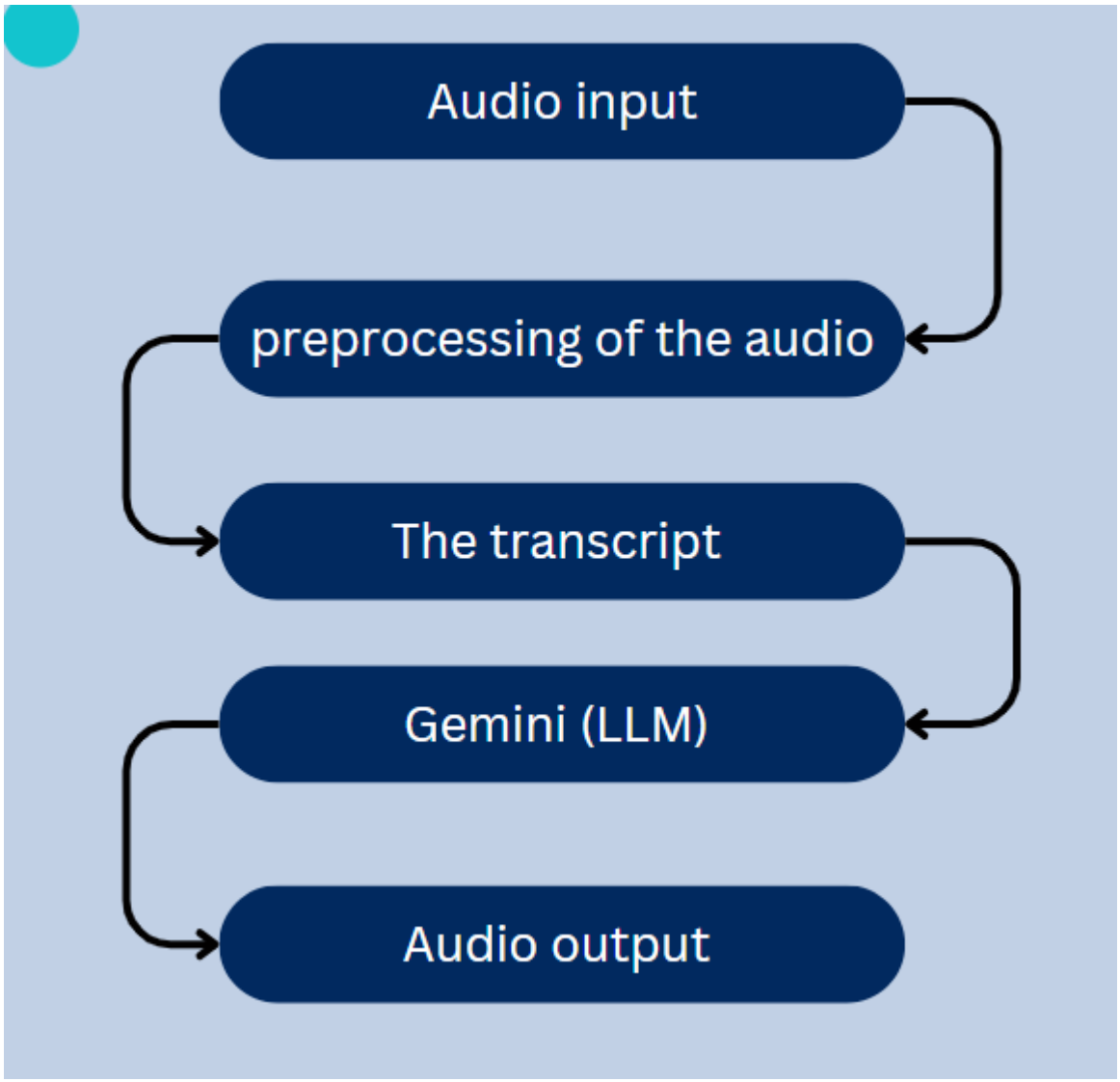
  - Model Loading: The Parler TTS model and its associated tokenizer are loaded.

  - Audio Generation: The response text is converted into audio using the TTS model, which produces speech that mimics the specified voice characteristics.

  - Purpose: Converting the text response back into audio ensures that the final output is accessible and user-friendly, providing a natural and engaging auditory experience.

#### 4. Conclusion

The model effectively integrates multiple advanced technologies to deliver a seamless end-to-end solution for processing voice audio. The pipeline ensures that spoken input is accurately transcribed, contextually relevant responses are generated, and those responses are converted into high-quality audio. Each stage of the workflow is optimized to handle specific aspects of the processing chain, resulting in a sophisticated system capable of transforming spoken input into meaningful and audible responses. This approach not only improves user interaction but also leverages cutting-edge tools to maintain high standards of accuracy and efficiency.


Audio input
preprocessing of the audio
The transcript
Gemini (LLM)
Audio output

```
!pip install git+https://github.com/huggingface/parler-tts.git

!pip install -q git+https://github.com/openai/whisper.git
!apt-get install -y ffmpeg
!pip install webrtcvad
!pip install bark
!pip install google-generativeai
```

1. `!pip install git+https://github.com/huggingface/parler-tts.git`:

This command installs a Python package directly from the Hugging Face GitHub repository called "parler-tts," which is likely related to text-to-speech (TTS) functionality.

2. `!pip install -q git+https://github.com/openai/whisper.git`:

This installs a Python package called "whisper" from the OpenAI GitHub repository. The `-q` flag makes the installation process quieter, showing fewer details in the output.

3.`!apt-get install -y ffmpeg`:

This command installs a software package called `ffmpeg`, which is used for handling multimedia files (like audio and video). The `-y` flag automatically agrees to any prompts during installation.

4. `!pip install webrtcvad`:

This installs a Python package called "webrtcvad," which provides a tool for voice activity detection (VAD), helping to identify when someone is speaking in an audio stream.

5. `!pip install bark`:

This installs a Python package called "bark." The specific functionality isn't clear from the name alone, but it could relate to various things like sound processing or AI models.

6. `!pip install google-generativeai`:

This command installs a Python package related to Google's generative AI tools, which likely includes models or tools for generating content like text, images, or code.

Here's a simple explanation of each line in this snippet:

```
import os
import whisper
import subprocess
import webrtcvad
import numpy as np
import scipy.io.wavfile as wavfile
import google.generativeai as genai
```

1. `import os`:

This line loads a tool that helps the program interact with the computer's operating system, like managing files or running commands.

2. `import whisper`:

This loads a tool that probably helps the program understand or process spoken language.

3. `import subprocess`:

This loads a tool that allows the program to run other programs or commands within the computer.

4. `import webrtcvad`:

This loads a tool that helps the program detect when someone is talking in an audio recording.

5. `import numpy as np`:

This loads a powerful math tool called NumPy and gives it the nickname `np`, which is often used to handle numbers, data, and calculations.

6. `import scipy.io.wavfile as wavfile`:

This loads a tool to work with audio files (specifically `.wav` files) and gives it the nickname `wavfile`.

7. `import google.generativeai as genai`:

This loads a set of tools from Google that helps the program create things like text or images using AI and gives it the nickname `genai`.

```
def convert_audio(input_path, output_path):
    # Convert audio to 16 kHz sampling rate and mono channel using ffmpeg
    command = [
        'ffmpeg',
        '-i', input_path,
        '-ar', '16000',    # Set sampling rate to 16 kHz
        '-ac', '1',       # Set audio channel count to mono
        output_path
    ]
    subprocess.run(command, check=True)
```

1. `def convert_audio(input_path, output_path):`

This line defines a function called `convert_audio` that takes two inputs: `input_path` (the location of the original audio file) and `output_path` (where the converted audio file will be saved).

this function takes an audio file, converts it to have a 16 kHz sampling rate and mono audio, and saves the new file in a specified location.

2. 'ffmpeg', '-i', input_path, '-ar', '16000', '-ac', '1', output_path`

These are the specific instructions for `ffmpeg`:

- `'ffmpeg'`: Calls the `ffmpeg` program.

- `'-i', input_path`: Specifies the input file (`input_path`).

- `'-ar', '16000'`: Sets the audio sampling rate to 16 kHz.

- `'-ac', '1'`: Sets the audio to have one channel (mono).

- `output_path`: Specifies where the output file should be saved.

5. `subprocess.run(command, check=True)`

This line runs the `ffmpeg` command that was just set up. The `check=True` part makes sure that if there's an error, report it

```
def apply_vad(input_path, output_path, vad_threshold=0.5):
    # Read the preprocessed audio
    sample_rate, audio_data = wavfile.read(input_path)
    vad = webrtcvad.Vad(1)  # VAD mode: 0 (Least aggressive) to 3 (Most aggressive)

    # Apply VAD to remove silence
    assert sample_rate in [8000, 16000, 32000, 48000], "Sample rate must be 8000, 16000, 32000, or 48000 Hz"
    audio_data = np.array(audio_data, dtype=np.int16)
    frame_duration = 0.02  # Duration of each frame (20 ms)
    frame_length = int(sample_rate * frame_duration)
    frames = [audio_data[i:i + frame_length] for i in range(0, len(audio_data), frame_length)]
    voiced_frames = []
    for frame in frames:
        if len(frame) == frame_length:  # Check if frame is the expected length
            is_speech = vad.is_speech(frame.tobytes(), sample_rate)
            if is_speech:
                voiced_frames.append(frame)
    voiced_audio = np.concatenate(voiced_frames)

    # Save the VAD processed audio
    wavfile.write(output_path, sample_rate, voiced_audio)
```

This function named `apply_vad` that processes an audio file to remove any silent parts and keep only the portions where someone is speaking. Here's a simple explanation of each part of the code:

1. Initialize Voice Activity Detection (VAD):

- A VAD object is created with mode 1. VAD mode can range from 0 (least aggressive) to 3 (most aggressive) in detecting speech. Mode 1 is a moderate level of aggressiveness.

2. Check Sample Rate:

- The code checks if the audio's sample rate is one of the allowed rates (8000, 16000, 32000, or 48000 Hz). If not, it will raise an error.

3. Prepare Audio Data for Processing:

- The audio data is converted to a format suitable for processing (16-bit integer).

- The audio is split into small chunks (frames) of 20 milliseconds each. This frame duration is important for accurately detecting speech.

4. Concatenate and Save Voiced Audio:

- All the frames containing speech are combined into a single audio stream.

- The processed audio, which only contains the parts with speech, is saved to the `output_path`.

this function takes an audio file, removes the silent parts, and saves the processed audio that only includes speech, making it useful for applications like speech recognition or communication systems where you want to ignore non-speech sounds.

```
input_audio_path = "/content/drive/MyDrive/internship/Recording.m4a"
preprocessed_audio_path = "/content/processed_audio2.wav"
vad_audio_path = "/content/vad_audio.wav"
```

1. `input_audio_path = "/content/drive/MyDrive/internship/Recording.m4a"`

This line specifies the location of the original audio file, named "Recording.m4a," stored in a Google Drive folder called "internship."

2. `preprocessed_audio_path = "/content/processed_audio2.wav"`

This line sets the location where the processed audio file will be saved, with the name "processed_audio2.wav."

3. `vad_audio_path = "/content/vad_audio.wav"`

```
convert_audio(input_audio_path, preprocessed_audio_path)
apply_vad(preprocessed_audio_path, vad_audio_path)
```

1. `convert_audio(input_audio_path, preprocessed_audio_path)`:

This function takes the original audio file located at `input_audio_path` (which we saw earlier is "Recording.m4a") and converts it to a new format (with a 16 kHz sampling rate and mono channel) and saves it at `preprocessed_audio_path` (which is "processed_audio.wav").

2. `apply_vad(preprocessed_audio_path, vad_audio_path)`:

This line takes the converted audio file ("processed_audio.wav") and applies Voice Activity Detection (VAD) to it, which likely filters out parts where there's no speech. The result is saved at `vad_audio_path` (which is "vad_audio.wav").

The code first converts the original audio file to a new format, then processes the converted file to detect and keep only the parts where someone is speaking.

```
def transcribe_audio(audio_path):
    model = whisper.load_model("base")
    result = model.transcribe(audio_path)
    return result["text"]

transcription = transcribe_audio(vad_audio_path)
print(transcription)
```

1. **`def transcribe_audio(audio_path):`**

This line defines a function called `transcribe_audio` that takes in the location of an audio file (`audio_path`) as input.

The function to take an audio file, convert the speech in it to text, and then it prints out what was said in the audio file.

2. `model = whisper.load_model("base")`

This line loads a pre-trained speech recognition model called "base" from the Whisper library. The model will be used to transcribe speech into text.

3.`result = model.transcribe(audio_path)`

This line uses the model to transcribe the speech from the audio file located at `audio_path`. The transcription result is stored in a variable called `result`.

4. `return result["text"]`

This line returns only the transcribed text from the result (ignoring any other information the model might have provided).

5. `transcription = transcribe_audio(vad_audio_path)`

This line calls the `transcribe_audio` function, passing in the path to the VAD-processed audio file (`vad_audio_path`), and stores the transcribed text in the `transcription` variable.

6.`print(transcription)`

Finally, this line prints the transcribed text to the screen.

```
genai.configure(api_key=api_key)

# Step 4: Initialize the Gemini model
model = genai.GenerativeModel("gemini-pro")
chat = model.start_chat(history=[])

# Step 5: Function to get a response from the Gemini model
def get_response(question):
    response = chat.send_message(question, stream=False)
    response_text = "".join([chunk.text for chunk in response])
    return response_text

# Step 6: Input query
query =transcription

# Step 7: Get response from Gemini
response_text = get_response(query)

# Step 8: Print the response
print("Response from Gemini:")
print(response_text)
```

This code snippet is for interacting with a generative AI model called "Gemini" using the Google Generative AI (genai) library. Here's what each part does:

1. `genai.configure(api_key=api_key)`

This line sets up the generative AI library by providing it with an API key (`api_key`). This key is needed to access the Gemini model.

2. `model = genai.GenerativeModel("gemini-pro")`

This line initializes the "Gemini" model, specifically a version called "gemini-pro," which is a generative AI model that can create responses based on input queries.

3. `chat = model.start_chat(history=[])`

This line starts a chat session with the Gemini model. The `history=[]` part indicates that the chat is starting fresh with no previous conversation history.

4. `def get_response(question):`

This line defines a function called `get_response` that takes a question (input query) as input and sends it to the Gemini model to get a response.

5.`response = chat.send_message(question, stream=False)`

Inside the `get_response` function, this line sends the input question to the Gemini model. The `stream=False` part indicates that the entire response should be received before processing.

6. `response_text = "".join([chunk.text for chunk in response])`

This line extracts the text from the response generated by the model and combines all the parts into a single string.

7. `return response_text`

This line returns the final response text from the `get_response` function.

8. `query = input("Enter your query: ")`

This line asks the user to type in a question or query, which is then stored in the variable `query`.

9. `response_text = get_response(query)`

This line sends the user's query to the Gemini model and stores the AI-generated response in the variable `response_text`.

10. `print("Response from Gemini:")`

This line prints a message indicating that the following output is the response from the Gemini model.

11. `print(response_text)`

Finally, this line prints out the actual response generated by the Gemini model based on the user's query.

The code sets up an AI model named Gemini, takes a question from the user, asks the model to respond, and then prints out the model's answer.

```
import torch
from parler_tts import ParlerTTSForConditionalGeneration
from transformers import AutoTokenizer
import soundfile as sf

device = "cuda:0" if torch.cuda.is_available() else "cpu"

model = ParlerTTSForConditionalGeneration.from_pretrained("parler-tts/parler-tts-large-v1").to(device)
tokenizer = AutoTokenizer.from_pretrained("parler-tts/parler-tts-large-v1")

prompt = response_text
description = "Jon's voice is monotone yet slightly fast in delivery, with a very close recording that almost has no background noise."

input_ids = tokenizer(description, return_tensors="pt").input_ids.to(device)
prompt_input_ids = tokenizer(prompt, return_tensors="pt").input_ids.to(device)

generation = model.generate(input_ids=input_ids, prompt_input_ids=prompt_input_ids)
audio_arr = generation.cpu().numpy().squeeze()
sf.write("parler_tts_out2.wav", audio_arr, model.config.sampling_rate)
```

This is  designed to generate synthetic speech from text using a Text-to-Speech (TTS) model called "ParlerTTS." :

1. Imports:

- `import torch`**: Loads the PyTorch library, which is used for running machine learning models, especially on GPUs.

- `from parler_tts import ParlerTTSForConditionalGeneration`: Loads the ParlerTTS model for generating speech.

- ‘from transformers import AutoTokenizer`: Loads a tool that converts text into a format the model can understand (tokenization).

- `import soundfile as sf`: Loads a library for saving audio files.

2. `device = "cuda:0" if torch.cuda.is_available() else "cpu"`

This line checks if a GPU is available. If it is, the code will use it (`"cuda:0"`); otherwise, it will use the CPU (`"cpu"`). This helps the model run faster if a GPU is available.

3. Model and Tokenizer Initialization:

- `model = ParlerTTSForConditionalGeneration.from_pretrained("parler-tts/parler-tts-large-v1").to(device)`**: Loads the pre-trained ParlerTTS model and sends it to the chosen device (GPU or CPU).

- `tokenizer = AutoTokenizer.from_pretrained("parler-tts/parler-tts-large-v1")`**: Loads the tokenizer that matches the ParlerTTS model.

4. Setting Up Inputs:

- `prompt = response_text`**: Uses the `response_text` (which might be from the previous snippet where the Gemini model generated a response) as the text that will be turned into speech.

- `description = "Jon's voice is monotone yet slightly fast in delivery, with a very close recording that almost has no background noise."`**: A description of the desired voice characteristics, used to guide the TTS model.

5. Tokenization:

- `input_ids = tokenizer(description, return_tensors="pt").input_ids.to(device)`: Converts the voice description into tokens that the model can understand.

- `prompt_input_ids = tokenizer(prompt, return_tensors="pt").input_ids.to(device)`: Converts the text prompt (the speech content) into tokens.

6.Generate Speech:

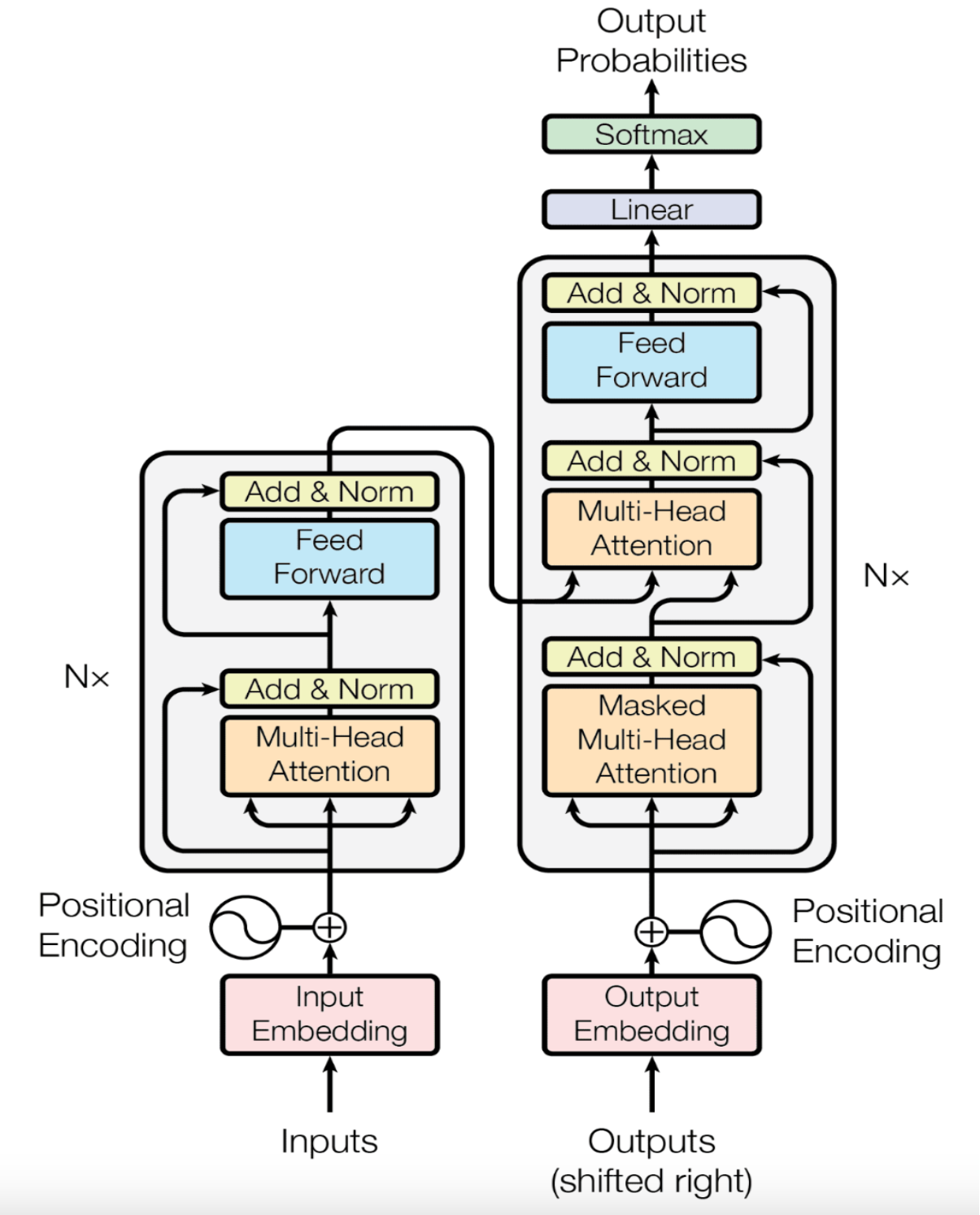
- `generation = model.generate(input_ids=input_ids, prompt_input_ids=prompt_input_ids)`: Generates the speech audio based on the voice description and text prompt.

7. Save the Audio:

-`audio_arr = generation.cpu().numpy().squeeze()`: Converts the generated audio from PyTorch format to a NumPy array and removes any extra dimensions.

- `sf.write("parler_tts_out.wav", audio_arr, model.config.sampling_rate)`: Saves the generated audio to a file named "parler_tts_out.wav" using the appropriate sampling rate from the model configuration.

This  takes a text description of a voice and a text prompt (what you want the voice to say), generates synthetic speech that matches the description, and then saves that speech to an audio file.




**The Gemini model**

**The git-hub link for video and code.**

**https://github.com/anshumanverse/Internship**