Report: Development of Our Text-to-Speech (TTS) System Using PyTorch

Objective:

The goal of our project was to create a **Text-to-Speech (TTS)** system from scratch using **PyTorch**. We aimed to develop a system that could convert text into speech by generating a waveform, all without relying on pre-trained models. This would help us understand the underlying mechanics of TTS systems and allow us to customize the process according to our needs.

Steps We Took in the Development Process:

1. Data Preparation:

To begin, we needed a dataset containing **text and corresponding audio pairs**. We used a dataset such as **LJSpeech**, which consists of thousands of such pairs. The first task was to preprocess the text. This involved converting all the text into lowercase, removing any unnecessary special characters, and tokenizing it into smaller units like words or characters. Tokenization is essential because it prepares the text to be fed into the model in a way that it can understand.

Next, we focused on processing the audio data. We used a method called **librosa** to convert the audio files into **Mel spectrograms**. These spectrograms represent the frequency content of the speech and are easier for the model to work with compared to raw audio waveforms. This preprocessing step allowed us to convert audio into a format suitable for training the model.

2. Model Architecture:

The core of our TTS system is a **Seq2Seq model**. This type of model is designed to convert sequences (in our case, text) into another sequence (in this case, a Mel spectrogram). Our model follows two main steps:

- **Encoder:** The input text is passed through an encoder that processes it and transforms it into a representation that the decoder can work with. This step is essential as it turns the text into numerical values, each representing a character or word.
- **Decoder:** The decoder then uses this processed input to generate the corresponding **Mel spectrogram**. A Mel spectrogram is a visual

representation of sound frequencies, and it is the first step in converting text into speech.

This architecture helped us map text sequences to audio features effectively, allowing the model to learn how to generate speech from text.

3. Vocoder (Mel-to-Waveform Conversion):

Once we had the Mel spectrogram from the model, we needed to convert it into an audible waveform. For this, we used a **vocoder**. The vocoder takes the Mel spectrogram and reconstructs it into a **waveform** that can be played as audio.

Instead of using complex models for this step, we opted for the **Griffin-Lim algorithm**. While not the most advanced method, it is simple and works well for basic tasks. It converts the Mel spectrogram into an audio waveform, though the quality of the output can be further improved with more advanced techniques. Normally, vocoders like **WaveGlow** or **HiFi-GAN** would offer better results, but for simplicity, we started with Griffin-Lim.

4. Training the Model:

Training the model was a critical step. We needed to pair each piece of text with the corresponding Mel spectrogram and feed it into the model. The model was trained using a loss function that measured the difference between the predicted and actual Mel spectrograms. By minimizing this loss over multiple iterations, the model learned to generate better Mel spectrograms.

Training the model was a lengthy process. Since we were working with complex audio data, it required significant computational resources and time. However, we could see the model gradually improving as it was exposed to more data.

5. Inference and Final Output:

After training, the model was ready to generate speech from new text. During inference, we input new text into the model. The model would process the text, generate the corresponding Mel spectrogram, and pass it through the vocoder to obtain the final audio waveform.

The output was then saved as a **.wav file**, which could be played back. This audio file contained the synthesized speech that was directly generated from the input text.

Challenges We Faced:

1. **Data Requirements:** We quickly realized that having a large and diverse dataset is crucial. The model's ability to generate high-quality speech is directly related to the quality and quantity of the data used for training.
2. **Model Complexity:** The architecture we chose was relatively simple, and while it worked, we could improve the system by incorporating more advanced models like **Tacotron 2** or **FastSpeech 2**. These models can handle more complex speech patterns and generate higher-quality audio.
3. **Vocoder Limitations:** The Griffin-Lim vocoder, while functional, is not the best at producing natural-sounding speech. If we had more resources, using a more advanced vocoder like **WaveGlow** or **HiFi-GAN** would significantly improve the output.

Conclusion:

Through this project, we successfully developed a basic **Text-to-Speech (TTS)** system using **PyTorch**. The system works by converting text into Mel spectrograms using a neural network, and then using a vocoder to turn these spectrograms into audible speech. While the system is functional, we recognize that it could be improved with more advanced techniques for both the model and vocoder.

This project has been an invaluable learning experience, providing insights into the complexities of speech synthesis. Going forward, we plan to refine the model and improve the audio quality by using better vocoder techniques and more advanced network architectures.

Here is the implementation

https://colab.research.google.com/drive/1iXbd0iH7MfuAJb7kvmWjGuvMuuQhD9DM?usp=sharing

the dataset used for the model is downloaded from the below link

https://www.kaggle.com/datasets/mathurinache/the-lj-speech-dataset/code