REAL TIME VOICE CLONING USING NEURAL FUSION

A PROJECT REPORT

Submitted by

AKSHAY SREE KRISHNA M **(211520104010)**

APPRAJIT VAIBHAV M **(211520104014)**

JOHN KELWIN JK **(211520104069)**

SRIRAM CM **(211520104156)**

in partial fulfillment for the award of the

degree of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

PANIMALAR INSTITUTE OF TECHNOLOGY, POONAMALLEE

ANNA UNIVERSITY: CHENNAI-600025

MAY 2024

ANNA UNIVERSITY: CHENNAI-600 025

BONAFIDE CERTIFICATE

Certified that this project report **"REAL TIME VOICE CLONING USING NEURAL FUSION**" is the bonafide work of **"AKSHAY SREE KRISHNA M (211520104010), APPRAJIT VAIBHAV M(211520104014),JOHN KELWIN JK (211520104069), SRIRAM CM (211520104156)"** who carried out the project work under my supervision.

SIGNATURE

Dr. D. LAKSHMI, M.E., PH.D.,

Professor and Head,

Department of CSE,
Panimalar Institute of Technology,
Poonamallee, Chennai-600123.

SIGNATURE

Mrs. TAMIL SELVI, M.E.

Assistant Professor,

Department of CSE,
Panimalar Institute of Technology,
Poonamallee, Chennai-600123.

Certified that the above candidates were examined in the university project work viva voice examination held on_____at Panimalar Institute of Technology,Chennai- 600123.

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ACKNOWLEDGEMENT

We wish to express our sincere thanks to all those who were involved in the completion of this project.

We would like to express our deep gratitude to our beloved **Secretary and Correspondent, Dr. P. CHINNADURAI, M.A., M.Phil., Ph.D.,** for his kind words and enthusiastic motivation.

We also express our sincere thanks and gratitude to all our dynamic **Directors Mrs.C.VIJAYA RAJESHWARI, Dr. C. SAKTHIKUMAR**, **M.E., Ph.D.,** and **Dr. SARANYASREE, SAKTHIKUMAR, B.E., M.B.A,Ph.D.**, for providing us infrastructure required to carry out this project.

We also express our appreciation and gratefulness to our respected **Principal Dr. T. JAYANTHY, M.E., Ph.D.,** for her thoughtful cooperation and encouragement.

We wish to convey our thanks and gratitude to our **Professor and Head of the Department, Dr. D. LAKSHMI, M.E., Ph.D.,** for her valuable guidance and excellent support.

Special thanks to our Supervisor **Mrs.TAMIL SELVI M.E.,Assistant Professor**, Computer Science and Engineering for his technical expertise and domain knowledge for successful completion of this project.

Last but not the least we place a deep sense of gratitude to our family members and our friends who have been constant source of inspiration during the preparation of this project work.

TABLE OF CONTENTS

ABSTRACT

A neural network-based system for text-to-speech (TTS) synthesis is outlined, demonstrating its capacity to generate speech audio in the voices of various speakers, including those not observed during training. This project comprises three independently trained components. A speaker encoder network, which undergoes training on a speaker verification task utilizing a dataset of noisy speech from thousands of speakers without transcripts. This network generates a fixed-dimensional embedding vector based on only a few seconds of reference speech from a target speaker.A sequence-to-sequence synthesis network, based on Tacotron 2, produces a mel spectrogram from input text, conditioned on the speaker embedding derived from the speaker encoder.An auto-regressive WaveNet-based vocoder network converts the mel spectrogram into time-domain waveform samples.The model effectively transfers knowledge of speaker variability learned by the discriminatively-trained speaker encoder to the multispeaker TTS task, enabling the synthesis of natural speech from speakers not encountered during training. Emphasis is placed on the importance of training the speaker encoder on a diverse and extensive speaker set to achieve optimal generalization performance. Furthermore, it is demonstrated that randomly sampled speaker embeddings can be employed to synthesize speech in the voices of novel speakers dissimilar to those encountered during training, highlighting the model's ability to learn a high-quality speaker representation.

LIST OF FIGURES

LIST OF ABBREVIATION

S.NO	ABBREVIATION	EXPANSION
1.	TTS	Text To Speech
2.	EFTS2	Efficient Text-To-Speech 2
3.	MT-KD	Multi-Teacher Knowledge Distillation
4.	MFCCs	Mel-Frequency cepstral coefficients
10	DTF	Discrete Fourier transform
11.	VC	Voice Conversion

CHAPTER 1

1.1INTRODUCTION:

The goal of this work is to build a TTS system which can generate natural speech for a variety of speakers in a data efficient manner. We specifically address a zero-shot learning setting, where a few seconds of untranscribed reference audio from a target speaker is used to synthesize new speech in that speaker's voice, without updating any model parameters. Such systems have accessibility applications, such as restoring the ability to communicate naturally to users who have lost their voice and are therefore unable to provide many new training examples. They could also enable new applications, such as transferring a voice across languages for more natural speech-to-speech translation, or generating realistic speech from text in low resource settings. However, it is also important to note the potential for misuse of this technology, for example impersonating someone's voice without their consent. In order to address safety concerns consistent with principles such as [1], we verify that voices generated by the proposed model can easily be distinguished from real voices. Synthesizing natural speech requires training on a large number of high quality speech-transcript pairs, and supporting many speakers usually uses tens of minutes of training data per speaker [8]. Recording a large amount of high quality data for many speakers is impractical. Our approach is to decouple speaker modeling from speech synthesis by independently training a speaker-discriminative embedding network that captures the space of speaker characteristics and training a high quality TTS

1.2OBJECTIVE:

To develop a real-time voice cloning system using neural fusion that can synthesize speech sounds in various speakers' voices, including those not seen during training. The system aims to achieve high-quality voice cloning with zero-shot adaptation to novel speakers, efficient and versatile real-time processing

capabilities, and the ability to generate distinct embeddings for voice identification . The system also aims to dynamically adapt to different noise environments and improve prosody in speech synthesis . Additionally, the project focuses on inference and zero-shot speaker adaptation, allowing for the synthesis of speech with corresponding speaker characteristics from audio clips of novel speakers outside the training set . The system also allows users to record personal speech segments for the generation of distinct embeddings, enabling user identification through voice . Overall, the project aims to showcase promising capabilities in multispeaker TTS synthesis, offering flexibility in training requirements and demonstrating reasonable performance even for unseen speakers.

CHAPTER 2

LITERATURE SURVEY

Paper 1

Title: Efficiency TTS 2: variational end to end text to speech synthesis and voice conversation

Authors: chefeng miao, qingying zhu, minchuan chen, jun ma, shaojun wang, jing xiao

Abstract: It introduces EfficientTTS 2 (EFTS2), an end-to-end text-to-speech model that utilizes a bijective flow structure for high-quality results and incorporates bidirectional prior/posterior training to reduce training-inference mismatch. EFTS2-VC, a voice conversion model based on EFTS2, maintains content information while altering speaker characteristics. Experimental results demonstrate strong performance in both single-speaker and multi-speaker TTS tasks. The study details the model architecture, training setup, and evaluation metrics, highlighting the model's efficiency, quality, and disentanglement capability. EFTS2 offers controllable diversity in latent variables and speech rhythms, with well-learned The paper introduces EfficientTTS 2 (EFTS2), an end-to-end text-to-speech model that utilizes a bijective flow structure for high-quality results and incorporates bidirectional prior/posterior training to reduce training- inference mismatch. EFTS2-VC, a voice conversion model based on EFTS2, maintains content information while altering speaker characteristics. Experimental results demonstrate strong performance in both single-speaker and multi-speaker TTS tasks. The study details the model architecture, training setup, and evaluation metrics, highlighting the model's efficiency, quality, and disentanglement capability. EFTS2 offers controllable diversity in latent variables and speech rhythms, with well-learned attention matrices. It is fully differentiable, enabling end-to-end training and outperforming baseline models in various aspects. The model's design choices can be applied to other TTS frameworks, showcasing its potential as an advanced TTS system. attention matrices. It is fully differentiable, enabling end-to-end training and outperforming baseline models in various aspects. The model's design choices can be applied to other TTS frameworks, showcasing its potential as an advanced TTS system.

Paper 2

Title: Expressive tts training with frame and style reconstruction loss

Authors: rui liu, berrak sisman, guanglai gao, haizhoo li

Abstract:

The paper introduces a novel training strategy for Tacotron-based Text-to-Speech (TTS) systems, Tacotron-PL, which incorporates frame and style reconstruction loss to enhance speech expressiveness without the need for prosody annotations. Experimental results demonstrate that Tacotron-PL outperforms existing models in terms of spectral, F0, and duration modeling, improving speech quality and expressiveness. By utilizing deep style features for perceptual loss evaluation and training with frame and style reconstruction loss, the proposed approach shows promising results for enhancing speech synthesis quality and expressiveness. Overall, the integration of frame and style reconstruction loss, along with deep style features, showcases the potential for advancing speech synthesis technology towards more expressive and high-quality outputs.

Paper3

Title: Decoding knowledge transfer for neural text to speech training

Authors: rui liu, berrak sisman, guanglai gao, haizhoo li

Abstract: It introduces a novel decoding knowledge transfer strategy, multi-teacher knowledge distillation (MT-KD), to combat exposure bias in neural text-to-speech (TTS) systems. By transferring knowledge from two teacher models to a student model, MT-KD enhances TTS performance in terms of naturalness, robustness, and expressiveness. Comparative evaluations demonstrate the superiority of MT-KD over other methods, showcasing its effectiveness in both in-domain and out-of-domain scenarios. The study underscores the benefits of knowledge distillation over adversarial learning and data augmentation in mitigating exposure bias, with MT-KD offering improved synthesis quality. Future directions include leveraging pre-trained models for multi-pass decoding to further enhance TTS system

Paper 4

Title: A Semi-Automatic Method for Transcription Error Correction for Indian Language TTS Systems

Authors: Swetha Tanamala, Jeena J Prakash and Hema A Murthy Department of Computer Science and Engineering, IIT Madras, India swethat, jeenaj, hema

Abstract: Database for building text-to-speech (TTS) synthesis systems consist of text sentences and the corresponding spoken waveform. Errors in tran- scription is an important factor that leads to poor synthesis quality. Mismatches between recorded speech and the corresponding text transcriptions occur mainly due to addition, deletion and substitution of words or phrases in the text or speech data. Presently, such mismatches are identified and corrected manually after listening to individual speech waveforms. In this work, this process of transcription error cor- rection is made semi-automatic by flagging off the error units at phone level, based on the log-likelihood values of each phone after forced Viterbi alignment. A user interface is developed that highlights the errors. The errors only need to be manually corrected. TTS systems are built for two languages with the complete dataset and with the dataset after removing the data with wrong units. Result of listening test conducted showed that there is a significant improvement in synthesis quality.

Paper 5

Title: Neural Fusion for Voice Cloning

Authors: Bo Chen , Student Member, IEEE, Chenpeng Du , and Kai Yu

Abstract: Voice cloning is a technique to build text-to-speech applications for individuals.When only very limited training data is available, it is challenging to preserve both high speech quality and high speaker similarity. We propose a neural fusion architecture to incorporate a unit concatenation method into a parametric text-to-speech model to address this issue. Unlike the hybrid unit concatenation system, the proposed fusion architecture is still an end-to-end neural network model. It consists of a text encoder, an acoustic decoder, and a phoneme-level reference

encoder. The reference encoder extracts phoneme-level embeddings corresponding to the cloning audio segments, and the text encoder infers phoneme level embeddings from the input text. One of the two embeddings is then selected and sent to the decoder. We use auto-regressive distribution modeling and decoder refinement after the selection stage to overcome the concatenation discontinuity problem. Experimental results show that the neural fusion system significantly improves the speaker similarity using the selected units with the highest probability

Paper 6

Title: TTS-BY-TTS: TTS-DRIVEN DATA AUGMENTATION FOR FAST AND HIGH-QUALITY SPEECH SYNTHESIS

Authors: Min-Jae Hwang, Ryuichi Yamamoto, Eunwoo Song, and Jae-Min Kim

Abstract: A text-to-speech (TTS)-driven data augmentation method for improving the quality of a non-autoregressive (AR) TTS system. Recently proposed non-AR models, such as FastSpeech 2, have successfully achieved a fast speech synthesis system. However, their quality is not satisfactory, especially when the amount of training data is insufficient. To address this problem, we propose an effective data augmentation method using a well designed AR TTS system. In this method, large-scale synthetic corpora including text-waveform pairs with phoneme duration are generated by the AR TTS system, and then used to train the target non- AR model. Perceptual listening test results showed that the proposed method significantly improved the quality of the non-AR TTS system. In particular, we augmented five hours of a training database to 179 hours of a synthetic one. Using these databases, our TTS system consisting of a FastSpeech 2 acoustic model with a Parallel WaveGAN vocoder achieved a mean opinion score of 3.74, which is 40% higher than that achieved by the conventional method.

Paper 7

Title: SPEECH BERT EMBEDDING FOR IMPROVING PROSODY IN NEURAL TTS

Authors: Liping Chen, Yan Deng, Xi Wang, Frank K. Soong, Lei He

Abstract: It presents a speech BERT model to extract embedded prosody information in speech segments for improving the prosody of synthesized speech in neural text-to-speech (TTS). As a pre-trained model, it can learn prosody attributes from a large amount of speech data, which can utilize more data than the original training data used by the target TTS. The embedding is extracted from the previous segment of a fixed length in the proposed BERT. The extracted embedding is then used together with the mel-spectrogram to predict the following segment in the TTS decoder. Experimental results obtained by the Transformer TTS show that the proposed BERT can extract fine-grained, segment-level prosody, which is complementary to utterance-level prosody to improve the final prosody of the TTS speech. The objective distortions measured on a single speaker TTS are reduced between the generated speech and original recordings. Subjective listening tests also show that the proposed approach is favorably preferred over the TTS without the BERT prosody embedding module, for both in-domain and out-of-domain applications. For Microsoft professional, single/multiple speakers and the LJ Speaker in the public database, subjective preference is similarly confirmed with the new BERT prosody embedding.

Paper 8

Title : Depressive Tendency Recognition by Fusing Speech and text features

Authors: Yimin He, Xiaoyong Lu*, Jingyi Yuan, Tao Pan, and Yafan Wang

Abstract: Depression will be accompanied by long-term depression, loss of interest, excessive guilt, and other states, seriously affecting people's physical and mental health, causing certain harm to the individual family and society. Depressed people speak slowly, single tone, and the pause time is longer. At the same time, the expression of emotion is often accompanied by many negative words. Therefore, the combination of speech and text information using Gated Recurrent Neural Network

for depression prediction, this fusion method from the signal layer to the language layer to analyze the data. It is more comprehensive than only use a single speech feature or text feature model. At the same time, the comparative analysis of different speech types, three kinds of emotional stimulus corpus, and gender was carried out. The multimodal system is more effective than a single modality, and speech type, emotional stimulus, and gender have certain effects on depressive recognition.

Paper 9:

Title: NAUTILUS: A Versatile Voice Cloning System

Authors: Hieu-Thi Luong , Member, IEEE, and Junichi Yamagishi , Senior Member, IEEE

Abstract: It introduce a novel speech synthesis system, called NAUTILUS, that can generate speech with a target voice either from a text input or a reference utterance of an arbitrary source speaker. By using a multi-speaker speech corpus to train all requisite encoders and decoders in the initial training stage, our system can clone unseen voices using untranscribed speech of target speakers on the basis of the backpropagation algorithm. Moreover, depending on the data circumstance of the target speaker, the cloning strategy can be adjusted to take advantage of additional data and modify the behaviors of text-to-speech (TTS) and/or voice conversion (VC) systems to accommodate the situation. We test the performance of the proposed framework by using deep convolution layers to model the encoders, decoders and WaveNet vocoder. Evaluations show that it achieves comparable quality with state-of-the-art TTS and VC systems when cloning with just five minutes of untranscribed speech. Moreover, it is demonstrated that the proposed framework has the ability to switch between TTS and VC with high speaker consistency, which will be useful for many applications.

Paper 10:

Title: VOICELDM: TEXT-TO-SPEECH WITH ENVIRONMENTAL CONTEXT

Authors: Yeonghyeon Lee, Inmo Yeon, Juhan Nam, Joon Son Chung

Abstract: It presents VoiceLDM, a model designed to produce audio that accurately follows two distinct natural language text prompts: the description prompt and the content prompt. The former provides information about the overall environmental context of the audio, while the latter conveys the linguistic content. To achieve this, we adopt a text-to-audio (TTA) model based on latent diffusion models and extend its functionality to incorporate an additional content prompt as a conditional input. By utilizing pretrained contrastive language- audio pretraining (CLAP) and Whisper, VoiceLDM is trained on large amounts of real-world audio without manual annotations or transcriptions. Additionally, we employ dual classifierfree guidance to further enhance the controllability of VoiceLDM. Experimental results demonstrate that VoiceLDM is capable of generating plausible audio that aligns well with both input conditions, even surpassing the speech intelligibility of the ground truth audio on the AudioCaps test set. Furthermore, we explore the text-to speech (TTS) and zero-shot text-to-audio capabilities of VoiceLDM and show that it achieves competitive results.

Paper 11:

Title: Direct Text to Speech Translation System Using Acoustic Units

Authors: Victoria Mingote , Pablo Gimeno , Luis Vicente , Sameer Khurana , Antoine Laurent , and Jarod Duret

Abstract: A direct text to speech translation system using discrete acoustic units. This framework employs text in different source languages as input to generate speech in the target language without the need for text transcriptions in this language. Motivated by the success of acoustic units in previous works for direct speech to speech translation systems, we use the same pipeline to extract the acoustic units using a speech encoder combined with a clustering algorithm. Once units are obtained, an encoder-decoder architecture is trained to predict them. Then a vocoder generates speech from units. Our approach for direct text to speech translation was

tested on the new CVSS corpus with two different text mBART models employed as initialisation. The systems presented report competitive performance for most of the language pairs evaluated. Besides, results show a remarkable improvement when initialising our proposed architecture with a model pre-trained with more languages.

Paper 12:

Title: Speech Synthesis With Mixed Emotions

Authors: Kun Zhou, Berrak Sisman, Rajib Rana , Member, Bjorn W. Schuller ,and Haizhou Li

Abstract: Emotional speech synthesis aims to synthesize human voices with various emotional effects. The current studies are mostly focused on imitating an averaged style belonging to a specific emotion type. In this paper, we seek to generate speech with a mixture of emotions at run-time. We propose a novel formulation that measures the relative difference between the speech samples of different emotions. We then incorporate our formulation into a sequence-to-sequence emotional text-to-speech framework. During the training, the framework does not only explicitly characterize emotion styles but also explores the ordinal nature of emotions by quantifying the differences with other emotions. At run-time, we control the model to produce the desired emotion mixture by manually defining an emotion attribute vector. The objective and subjective evaluations have validated the effectiveness of the proposed framework. To our best knowledge, this research is the first study on modelling, synthesizing, and evaluating mixed emotions in speech.

CHAPTER 3

SYSTEM ANALYSIS

3.1 EXISTING SYSTEM:

The existing system is a neural fusion architecture for small data voice cloning tasks that incorporates a unit concatenation method into a parametric text-to-speech model. This architecture is designed on the phoneme-level prosody conditioned FastSpeech2 and significantly improves speaker similarity. The existing system requires a pre-trained model and takes significant time to train the voice cloning system.

3.2 PROPOSED SYSTEM:

In the Proposed system with this application, neural fusion architecture for small data voice cloning tasks incorporates a unit concatenation method into a parametric text-to-speech model. This architecture is designed on the phoneme-level prosody conditioned FastSpeech2 and significantly improves speaker similarity.

Advantages:

1. The proposed neural fusion architecture for small data voice cloning tasks significantly improves speaker similarity.
2. The incorporation of unit embeddings in the architecture enhances the speaker similarity in synthetic speech.
3. The refinement procedure in the architecture can mitigate concatenation problems, leading to improved speech quality.
4. Experimental results demonstrate significant improvements in speaker similarity while maintaining speech naturalness.

3.3 MODULE DESCRIPTION:

1. Encoder
2. Synthesizer
3. Vocoder

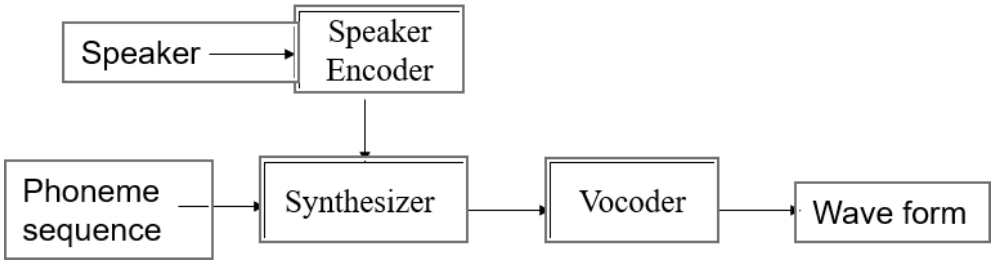

Figure 3.1 voice clone block diagram

DESCRIPTION:

The Voice Cloning aims to develop a system capable of replicating a person's voice using neural fusion technology. Neural fusion involves the integration of various neural network architectures to enhance the accuracy and fidelity of voice cloning. This system analysis provides an overview of the project's objectives, requirements, and the architecture involved**.** It also involves several key components. Initially, the audio data undergoes preprocessing, including noise reduction and feature extraction. These features are then represented in suitable formats such as spectrograms or Mel-frequency cepstral coefficients (MFCCs) for input to the neural networks. Neural fusion techniques are employed to combine multiple architectures like RNNs, CNNs, and GANs, leveraging their respective strengths. The trained models are deployed into an inference engine for real-time voice synthesis.

1.Synthesizer:

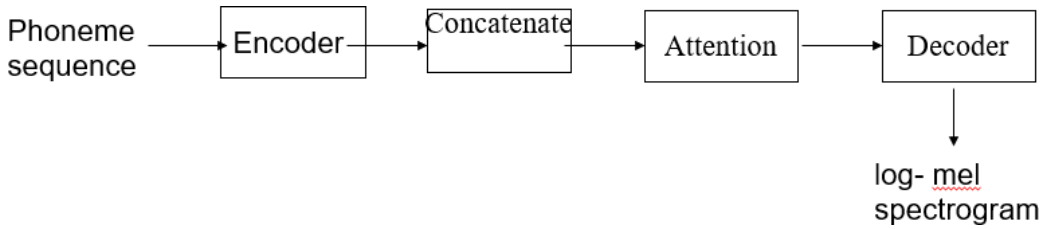

Figure:3.2 Synthesizer workflow

DESCRIPTION:

The synthesizer serves as the essential engine that converts text input into natural-sounding speech. It employs sophisticated techniques, often leveraging deep learning and neural networks, to mimic human speech patterns accurately. This involves considering elements such as intonation, rhythm, pitch, and timbre to generate speech that sounds convincingly human-like. A high-quality

synthesizer offers features like customization options for adjusting parameters such as pitch and speaking rate to achieve specific voice characteristics. Moreover, it may support multilingual capabilities and the ability to convey different emotions in the synthesized speech, enhancing its expressiveness. With real-time processing capabilities, the synthesizer enables interactive applications where generated speech responds promptly to user inputs. Ultimately, the synthesizer plays a pivotal role in voice cloning projects, powering applications ranging from virtual assistants to accessibility tools and audiobook narration, by transforming text into lifelike speech.

2.Vocoder:

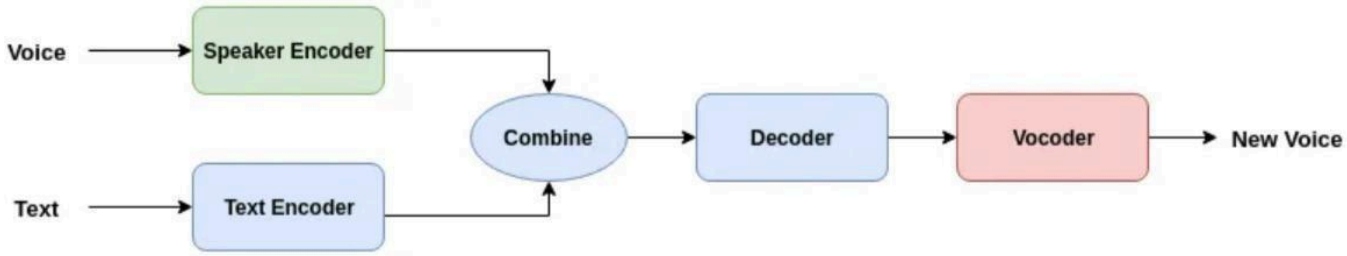

Figure 3.3 Vocoder's workflow

DESCRIPTION:

A vocoder plays a pivotal role in the synthesis of speech. A vocoder, short for voice encoder, is an essential component that processes and modulates the characteristics of a human voice. It functions by analyzing the spectral information of the input voice signal and then imposing those characteristics onto a carrier signal, typically generated synthetically. This process enables the replication of the nuances, intonations, and timbre of a particular voice onto a synthesized speech output. By utilizing advanced algorithms and techniques, a vocoder can effectively capture the subtleties of a speaker's voice, including pitch variations, resonance, and cadence. The resulting synthesized speech closely mirrors the original voice, making it a valuable tool for applications such as voice cloning, speech synthesis, and voice transformation. In a voice cloning project, the vocoder serves as a crucial element in recreating a lifelike and natural-sounding replication of a target voice, contributing significantly to the authenticity and quality of the synthesized speech output.

3.4 TECHNIQUE:

Neural Fusion is a cutting-edge technique utilized within voice cloning projects, revolutionizing the field with its sophisticated amalgamation of neural network architectures. It seamlessly integrates multiple neural networks, synergistically combining their strengths to enhance the fidelity and naturalness of cloned voices.

How does Neural Fusion work?

Neural Fusion in voice cloning operates by integrating the outputs of multiple neural networks in a seamless and complementary manner. Initially, distinct neural networks are trained on different aspects of speech synthesis, such as phonetic features, prosody, and emotional expressiveness. Each network specializes in extracting specific features from input speech data, like pitch, rhythm, and cadence. The fusion mechanism intelligently combines these outputs, leveraging the strengths of each network while compensating for any weaknesses. This fusion process is often dynamic, adapting in real-time based on input characteristics and desired output. Through iterative training and fine-tuning using large datasets, Neural Fusion optimizes network performance and refines the fusion mechanism to achieve high fidelity in voice cloning. The resulting synthetic voice mirrors the original speaker's characteristics with remarkable realism, capturing subtle nuances of pronunciation, intonation, and emotion. Overall, Neural Fusion represents a sophisticated approach to synthesizing human-like speech, offering compelling and natural-sounding synthetic voices for various applications.

Since the audio data is small in voice cloning tasks, parametric methods are usually adopted. The synthesized audio usually achieves good quality and similarity with the speaker embeddings. However, the similarity relies on the database to train the multi-speaker model and the method to extract the speaker embeddings. The similarity might drop if the speaker’s identity is distinctive to the training corpus. Based on these facts, this paper introduces a neural fusion architecture to incorporate a unit concatenation method in a parametric text-to-speech system.

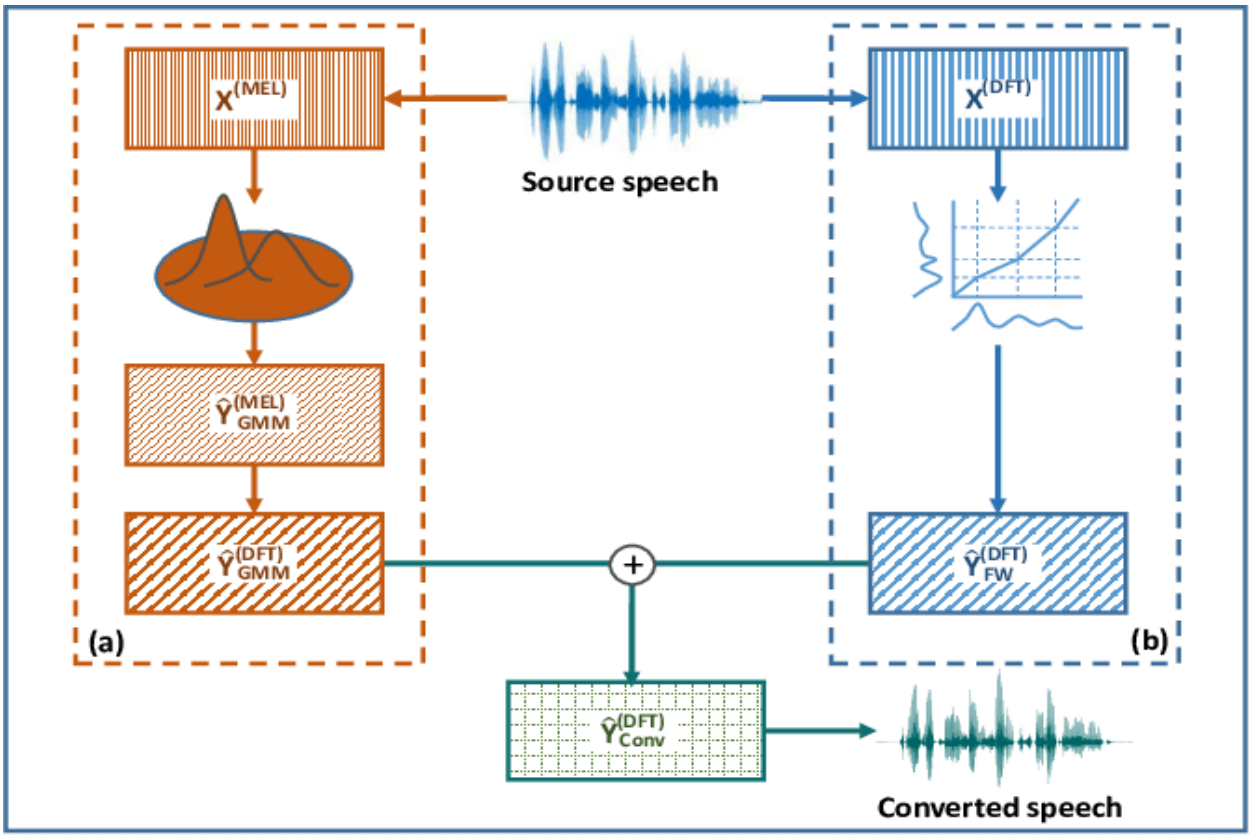

Figure:3.4: voice conversion system fusion. (a) is the conversion process of GMM-based VC system, (b) is the conversion process of FW-based VC system

Voice conversion system fusion is a technique that combines the strengths of different voice conversion (VC) approaches to achieve better overall performance. The concept is based on the idea that individual VC methods have their own advantages and disadvantages.

For instance, Gaussian Mixture Model (GMM) based systems excel at capturing the speaker's identity and transforming the voice to sound like the target speaker. However, they might not always generate the highest quality speech. On the other hand, Frequency Warping (FW) based systems are known for producing high-quality converted speech. But, their ability to accurately capture speaker identity might be limited.

This is where voice conversion system fusion comes in. By combining these two approaches, researchers aim to leverage the best of both worlds. They can achieve this by using a framework that integrates GMM and FW systems. Here's a breakdown of a possible fusion process based on the provided figure:

Feature Extraction: Both the source (original speaker) and target (desired speaker) speech are analyzed to extract Mel-Cepstral Coefficients (MCCs) and spectrum features. MCCs provide information about the speaker's vocal tract characteristics, while spectrum features capture

the overall frequency content of the speech.

Alignment: Dynamic Time Warping (DTW) is applied to align the MCCs sequences of the source and target speech. This ensures that corresponding frames from each speaker's speech are matched, facilitating the conversion process.

Model Training/Dictionary Construction: The aligned features are then used for training or building the conversion models specific to each approach. GMM-based methods utilize both the aligned MCCs and spectrum for model training. This model learns the mapping between the source and target speaker's voice characteristics. On the other hand, the FW-based method focuses solely on the aligned spectra for constructing a dictionary. This dictionary essentially stores information on how to modify the source speech spectrum to achieve the target speaker's characteristics.

Fusion Strategy (not shown in figure): This crucial step involves combining the outputs from the GMM and FW systems. The specific details of this fusion strategy would depend on the research work, but it could involve weighting the contributions of each system or combining their outputs in a specific way.

By combining the speaker identity transformation capabilities of GMM with the high-quality speech generation of FW, voice conversion system fusion strives to achieve a more natural and accurate conversion of the source speech to the target speaker's voice.

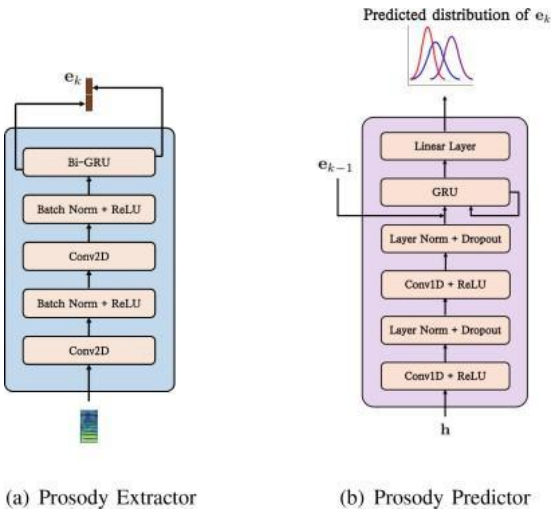

Figure: 3.5 Prosody

The roles of Prosody Extractor and Prosody Predictor in voice cloning can be understood as a two-act play: analyzing the target speaker and then using that analysis to create a natural-sounding voice.

Act 1: Prosody Extractor - The Keen Observer

Imagine Prosody Extractor as a detective meticulously studying the target speaker's speech patterns. Its job is to extract all the clues about how the speaker uses prosody, which includes:

Pitch: How high or low the voice goes (think excitement raising the pitch or questions ending with a higher pitch).
Rhythm: The pacing and emphasis of the speech (like faster speech for excitement or pauses for dramatic effect).
Intonation: The way the pitch changes throughout the speech (imagine the sing-song tone of a question).
Tools of the Trade:

Prosody Extractor uses sophisticated signal processing techniques to analyze the target speaker's audio. It might extract features like:

Fundamental frequency (F0): This captures the variations in pitch that correspond to how high or low the voice sounds.
Energy contours: These reveal changes in volume, helping to identify emphasis and stress patterns used by the speaker.
Silence markers: These pinpoint pauses and breaks in speech, which are also part of prosody.
By gathering this information, Prosody Extractor builds a profile of the target speaker' s prosodic fingerprint.

Act 2: Prosody Predictor - The Voice Architect

Now comes Prosody Predictor, who takes the baton from the detective. It receives the prosodic features extracted by Prosody Extractor and uses them as a blueprint.

Imagine Prosody Predictor as an architect using the speaker's prosodic fingerprint to design a new voice for unseen text. Here's what it does:

Machine Learning Magic: Prosody Predictor often relies on machine learning models trained on a massive dataset of speech with corresponding prosodic annotations. This training allows the model to learn the connection between text and how it's likely to be spoken with specific variations in pitch, rhythm, and intonation.
Predicting the Target Speaker's Delivery: By analyzing the new text and the extracted prosodic features, Prosody Predictor essentially predicts how the target speaker would deliver that specific message. It adjusts pitch, rhythm, and intonation patterns to mimic the target speaker's natural speaking style.

The Grand Finale: A Natural-Sounding Voice Clone

By combining the work of Prosody Extractor and Prosody Predictor,the resulting voice clone goes beyond just mimicking the voice timbre (the basic sound) of the target speaker. It also

captures the way they speak, making the cloned voice sound more natural, expressive, and even convey emotions through prosodic variations. This enhances the overall quality and believability of the voice clone.

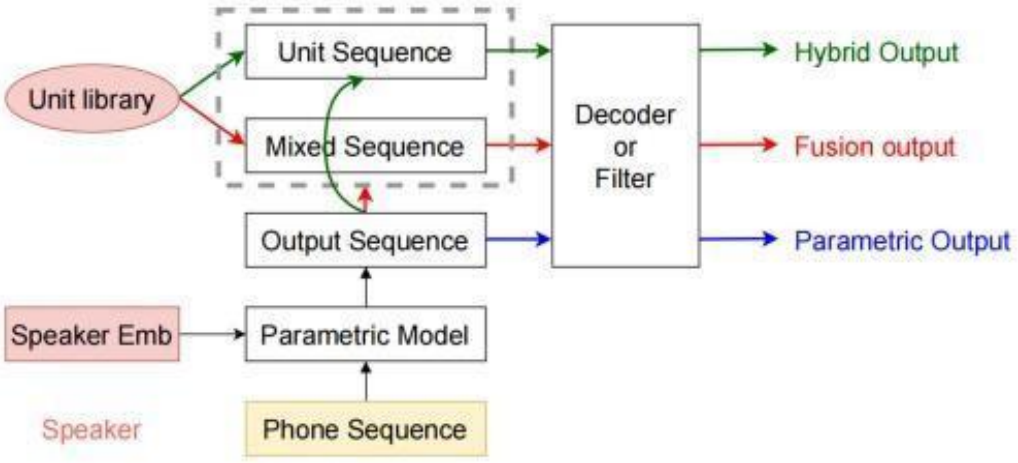

Figure:3.6 : Architecture of the fusion system compared to the hybrid concatenation system and the parametric system

Voice conversion (VC) systems can be built using different architectures, each with its strengths and weaknesses. Here's a comparison of three common approaches: fusion, hybrid concatenation, and parametric.

Fusion systems act like a team of specialists. They combine multiple VC approaches, like using one system for capturing speaker identity (GMM) and another for high-quality speech (FW). This can result in better overall performance by leveraging the complementary strengths of each approach. However, designing an effective fusion strategy to combine their outputs is crucial and can be computationally demanding.

Hybrid Concatenation System:

Hybrid concatenation systems are like a skilled impersonator with a vast library of voice clips. They analyze the source speaker's speech and then meticulously select snippets from a target speaker's database that closely match the source's characteristics. These snippets are then stitched together to form the converted speech. This approach can achieve natural-sounding conversions with proper prosody (speaking style), but requires a large, high-quality collection of the target speaker's voice recordings.

Parametric System:

Parametric systems are like a voice-altering machine. They rely on statistical models trained on a large amount of data from both speakers. The model learns the mapping between

their speech features and can directly convert the source features into the target speaker's features. This method is efficient and works in real-time, but achieving natural-sounding conversions can be challenging compared to other approaches.

The choice of architecture depends on the desired outcome. Fusion systems offer high-quality conversions but require careful design. Hybrid concatenation excels in natural prosody with a good target speaker database. Parametric systems are efficient but might struggle with naturalness.

It combines the strengths of two or more individual VC approaches, each tackling a specific aspect of the conversion.

Strengths:

Leverages Expertise: By combining approaches (e.g., GMM for speaker identity, FW for speech quality), a fusion system can potentially achieve better overall performance compared to a single method.
Customization Potential: The choice of VC approaches within the fusion framework allows for customization based on the desired outcome.

Challenges:

Fusion Strategy: Designing an effective way to combine the outputs from different systems is crucial for success. This strategy might involve weighting their contributions or merging them in a specific way, requiring careful research and experimentation.
Computational Cost: Combining multiple VC methods can be computationally expensive, especially for real-time applications.

However, if real-time conversion or computational efficiency is crucial, a parametric system could be a viable option, even if it comes with some trade-offs in naturalness

CHAPTER 4

SYSTEM SPECIFICATION

4.1 GENERAL:

These are the requirements for doing the project. Without using these tools and software's we can't do the project. So we have two requirements to do the project. They are

1. Hardware Requirements.
2. Software Requirements.

4.2 HARDWARE REQUIREMENTS:

The hardware requirements may serve as the basis for a contract for the implementation of the system and should therefore be a complete and consistent specification of the whole system. They are used by software engineers as the starting point for the system design. It shows what the system does and not how it should be implemented.

PROCESSOR	:	Intel Core i5 @ 2.5GHz
RAM	:	4GB RAM AND ABOVE
GPU	:	2GB OR above
Audio	:	Microphone
Display	:	720p Monitor or above

4.3SOFTWARE REQUIREMENTS:

The software requirements document is the specification of the system. It should include both a definition and a specification of requirements. It is a set of what the system should do rather than how it should do it. The software requirements provide a basis for creating the software requirements specification.

It is useful in estimating cost, planning team activities, performing tasks and tracking the team's and tracking the team's progress throughout the development activity.

Operating System	: Windows 10 & 11
Language	: PYTHON
IDE	: MICROSOFT 'S VISUAL STUDIO CODE
Libraries	: Tkinder(GUI)
Framework	: PyTorch
Package Manager	: PIP
Dependencies	: FFmpe

CHAPTER 5

SYSTEM DESIGN

5.1 SYSTEM ARCHITECHTURE:

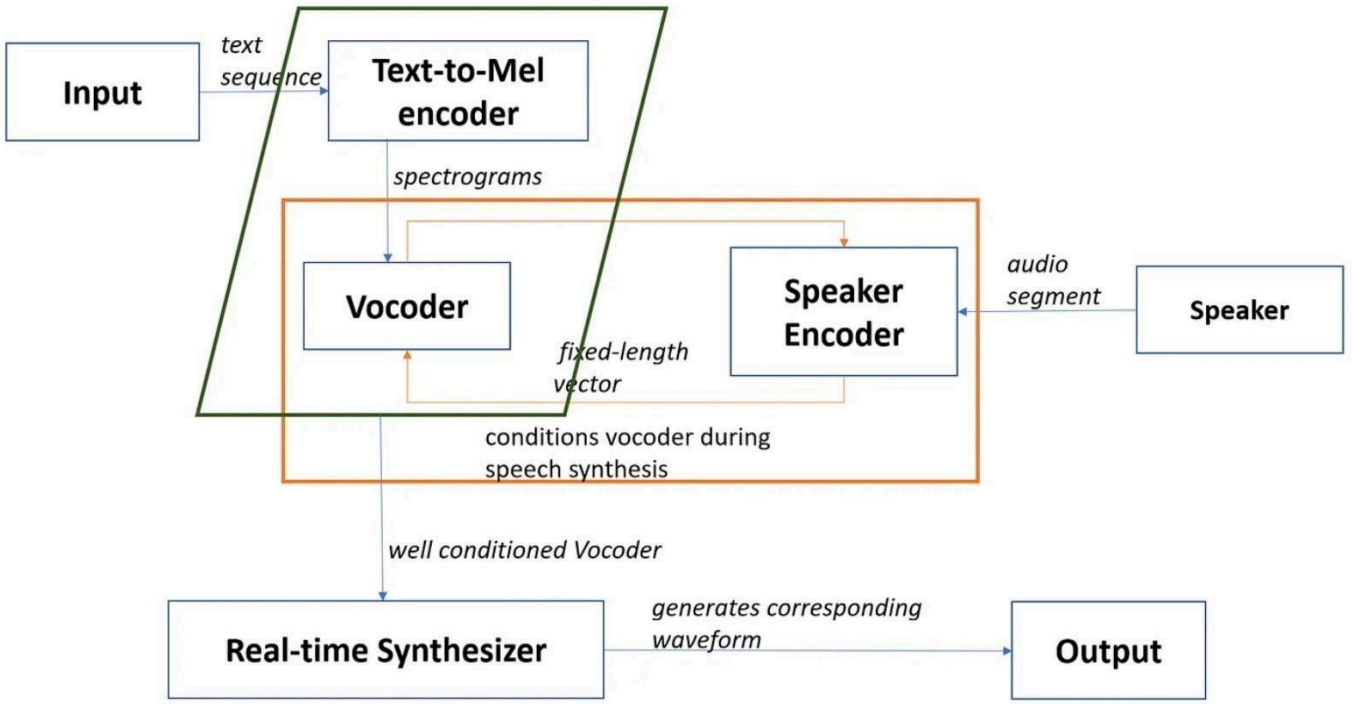

Figure 5.1 System Architecture

The proposed system architecture is a neural fusion system that combines unit concatenation and parametric systems for voice cloning tasks . This architecture consists of a text encoder, an acoustic decoder, and a phoneme-level reference encoder to extract phoneme-level embeddings from audio segments and input text . The fusion system selects unit embeddings with the highest probability and incorporates them into the decoder to generate synthetic speech . Additionally, the system uses auto-regressive distribution modeling and decoder refinement to address concatenation discontinuity issues.

A neural fusion voice clone system leverages the strengths of multiple voice conversion (VC) approaches to achieve high-quality and speaker-specific voice cloning. Here's a breakdown of its typical architecture:

5.1.1 **Preprocessing Stage**:

Data Acquisition: This involves collecting high-quality audio recordings from

both the source speaker (whose voice you want to modify) and the target speaker (whose voice you want to clone).

Data Preprocessing: The audio recordings are then preprocessed to ensure consistency and prepare them for further processing. This might involve tasks like:

Noise reduction to remove unwanted background sounds.
Format conversion to ensure all recordings have a uniform format (e.g., sample rate, bit depth).
Segmentation into smaller units (e.g., phonemes, frames) for easier processing by the neural networks.

Feature Extraction: Relevant features are extracted from the preprocessed audio data. These features capture information about the speaker's voice characteristics, such as:

Mel-Cepstral Coefficients (MCCs): Represent the spectral envelope of the speech signal, which contributes to the voice timbre (perceived quality).
Spectrum features: Capture the overall frequency content of the speech.

5.1.2. Model Training Stage:

Individual VC System Training:

The system likely utilizes two or more separate neural network models, each specializing in a different aspect of voice conversion. Common choices include:
GMM-based model: Focuses on capturing the speaker's identity and transforming the source speaker's features to resemble the target speaker's identity.

FW-based model: Concentrates on improving the speech quality of the converted voice, often utilizing a dictionary approach.
Each model is trained on its respective data using the extracted features from the source and target speaker recordings.

5.1.3. Fusion Stage (Core of Neural Fusion):

Combining Outputs: This crucial step involves merging the outputs from the individual VC systems. The specific strategy for fusion can vary depending on the research work:

Weighted Sum: The outputs might be combined using weights that determine the relative contribution of each system.

Feature-Level Concatenation: The extracted features from each system might be directly concatenated (joined together) before feeding them into another neural network for refinement.

Hybrid Approaches: More sophisticated fusion strategies might involve combining elements from different approaches.

5.1.4. Post-processing Stage:

Refinement and Synthesis: The fused output from the previous stage might undergo further processing for refinement and speech synthesis. This could involve tasks such as:

Removing artifacts or inconsistencies introduced during the fusion process. Generating a smooth and natural-sounding voice using a vocoder or other audio synthesis techniques.

5.1.5. Evaluation:

The final converted voice is evaluated for quality and speaker similarity. This might involve human listening tests or objective metrics designed to assess naturalness, speaker identity match, and audio fidelity.

Benefits of Neural Fusion:

Improved Overall Performance: By combining the strengths of different approaches, neural fusion aims to achieve better voice conversion quality compared to using a single method.
Speaker Identity and Speech Quality: GMM focuses on capturing speaker identity, while FW improves speech quality. Fusion leverages both aspects.

Challenges of Neural Fusion:

Designing Effective Fusion Strategy: Finding the optimal way to combine outputs from different systems is crucial and requires careful research and experimentation.
Computational Complexity: Training and running multiple neural networks can be computationally expensive, especially for real-time applications.

Overall, the neural fusion architecture offers a promising approach for creating high-quality voice clones by combining the strengths of different voice conversion techniques.

CHAPTER 6

SOFTWARE IMPLEMENTATIONS

6.1 GENERAL:

This chapter is about the software language and the tools used in the development of the project. The platform used here is Python. The primary languages are Python, along with libraries like pytorch and ffmpeg. FEATURES OF JAVA

6.1.1 THE PYTHON

Python is a dynamic, high-level programming language created by Guido van Rossum and first released in 1991. It emphasizes code readability and simplicity, with a syntax that allows programmers to express concepts in fewer lines of code compared to languages like Java. Python is versatile, supporting multiple programming paradigms including procedural, object-oriented, and functional programming.

Python's design philosophy emphasizes ease of use and readability, which has contributed to its popularity among beginners and experienced developers alike. It features a comprehensive standard library and a vast ecosystem of third-party packages that extend its functionality for various purposes, from web development to scientific computing and artificial intelligence.

Python code is typically interpreted, meaning it is executed line by line by the Python interpreter. However, it can also be compiled to bytecode for improved performance using tools like PyPy or Cython. Python's cross-platform compatibility allows code to run on different operating systems without modification.

Python's strengths lie in its simplicity, readability, and versatility, making it suitable for a wide range of applications, from scripting and automation to web development, data analysis, machine learning, and more. Its community-driven development model and open-source nature contribute to its ongoing evolution and widespread adoption in both industry and academia.

6.1.2 OBJECTIVES OF PYTHON

To see places of python in Action in our daily life, explore python.org.

WHY DEEP LEARNING DEVELOPERS CHOOSE PYTHON:

Deep learners often select Python as their preferred programming language due to several reasons:

- **Ease of Learning and Use**: Python's simple and clean syntax makes it easier for beginners to understand and write code compared to languages like Java. This accessibility lowers the barrier to entry for individuals interested in deep learning.

- **Rich Ecosystem of Libraries:** Python boasts a vast ecosystem of libraries and frameworks tailored for deep learning tasks. Libraries like TensorFlow, PyTorch, and Keras provide high-level abstractions for building and training neural networks, making complex tasks more manageable.

- **Community Support:** Python has a vibrant and active community of developers who contribute to the development of libraries, provide support through forums and online communities, and share resources such as tutorials and documentation. This support network can be invaluable for deep learners seeking assistance and guidance.

- **Flexibility and Versatility:** Python's versatility allows deep learners to use it for various tasks beyond deep learning, such as data analysis, web development, automation, and more. This flexibility enables individuals to leverage their Python skills across different domains and projects.

- **Integration with Other Technologies:** Python integrates seamlessly with other technologies commonly used in the deep learning ecosystem, such as NumPy for numerical computations, pandas for data manipulation, and matplotlib for data visualization. This interoperability streamlines the development workflow and facilitates experimentation and prototyping.

- **Performance and Efficiency:** While Python is generally slower than lower-level languages like C or C++, its performance is often sufficient for deep learning tasks, especially when combined with optimized libraries and frameworks. Additionally, Python's ease of use and rapid development cycle can outweigh minor performance considerations for many deep learning practitioners.

Overall, the combination of Python's simplicity, extensive libraries, community support,

versatility, and performance characteristics makes it an attractive choice for deep learners looking to explore and apply deep learning techniques effectively.

6.1.3 Evolution of voice cloning:

1. The evolution of voice cloning represents a significant milestone in the field of artificial intelligence and speech synthesis. Initially, voice cloning techniques relied on concatenative synthesis, which involved stitching together pre-recorded speech segments to generate new utterances. While effective to some extent, this method often resulted in unnatural-sounding voices and limited flexibility.

2. However, advancements in machine learning, particularly in the field of deep learning, have revolutionized voice cloning. Deep learning-based approaches, such as neural network-based text-to-speech (TTS) models, have enabled more natural and expressive speech synthesis. These models learn to generate speech from text inputs by capturing the nuances of human speech patterns and intonations, resulting in more realistic and lifelike voices.

3. Moreover, the advent of generative models, such as generative adversarial networks (GANs) and variational autoencoders (VAEs), has further improved voice cloning capabilities. These models can learn to synthesize speech from limited training data, allowing for the creation of personalized voices with minimal recordings.

4. Recent breakthroughs in neural vocoder architectures, such as WaveNet and WaveGlow, have also contributed to the advancement of voice cloning technology. These models produce high-quality speech waveforms directly from spectrograms generated by TTS models, resulting in clearer and more natural-sounding voices.

5. Furthermore, the availability of large-scale datasets and powerful computational resources has accelerated progress in voice cloning research. Techniques like transfer learning and fine-tuning have enabled researchers to adapt pre-trained TTS models to new voices or languages with relative ease.

6. Despite these advancements, challenges remain, including addressing issues related to voice privacy, authentication, and ethical concerns surrounding the misuse of voice cloning technology for fraudulent purposes. Nevertheless, the evolution of voice cloning holds immense potential for applications in speech synthesis, virtual assistants, accessibility tools, entertainment, and beyond, shaping the future of human-computer interaction.

6.1.4 Benefits of python

1. **Readability**: Python's syntax is designed to prioritize human readability, making it easier to understand and maintain code. Its use of indentation for block delimiters enforces consistent formatting, reducing the likelihood of errors and enhancing code clarity. This readability is particularly advantageous for collaboration among team members and for code reviews.
2. **Versatility**: Python's versatility stems from its support for multiple programming paradigms, including procedural, object-oriented, and functional programming. This flexibility allows developers to choose the most appropriate approach for their specific needs, whether they're building web applications, data analysis tools, scientific simulations, or machine learning models.

3. **Rich ecosystem**: Python boasts a vast ecosystem of libraries and frameworks covering virtually every aspect of software development. From popular libraries like NumPy, pandas, and Matplotlib for data manipulation and visualization, to frameworks like Django and Flask for web development, and TensorFlow and PyTorch for deep learning, Python offers a comprehensive toolkit that accelerates development and reduces the need to reinvent the wheel.

4. **Community support**: Python enjoys widespread adoption and has a large and active community of developers, enthusiasts, and experts. This community provides a wealth of resources, including documentation, tutorials, forums, and open-source contributions. Whether seeking assistance with a specific problem, learning new techniques, or contributing to open-source projects, developers can benefit from the knowledge-sharing and collaboration fostered by the Python community.

5. **Cross-platform compatibility**: Python's cross-platform compatibility allows code written in Python to run seamlessly on various operating systems,

including Windows, macOS, and Linux. This portability eliminates the need for separate codebases for different platforms, simplifying development and deployment processes. Additionally, Python's compatibility with other languages and technologies enables seamless integration with existing systems, extending its utility and versatility in diverse environments.

6.2 PYTORCH

- PyTorch is a powerful open-source machine learning library developed by Facebook's AI Research lab (FAIR). It's renowned for its flexibility, ease of use, and dynamic computational graph construction, making it a popular choice among researchers and practitioners in the field of deep learning. Here's a brief overview:

- PyTorch provides a dynamic computational graph, which means that the graph is built on-the-fly as operations are executed, allowing for more flexibility in model construction and debugging compared to static graph frameworks like TensorFlow.

- Its intuitive interface and Pythonic syntax make it easy to learn and use, enabling rapid prototyping and experimentation. This accessibility has contributed to its widespread adoption in both academia and industry.

- PyTorch offers a rich ecosystem of modules and utilities for building and training neural networks. Its modular design allows users to construct complex models by composing smaller building blocks, facilitating code reuse and modular development.

- One of PyTorch's standout features is its support for automatic differentiation, which enables efficient computation of gradients for optimization algorithms like stochastic gradient descent (SGD). This feature simplifies the implementation of custom loss functions and network architectures.

- PyTorch seamlessly integrates with popular Python libraries like NumPy, pandas, and scikit-learn, enhancing its capabilities for data preprocessing, manipulation, and analysis.

- Furthermore, PyTorch has a thriving community of developers who contribute to

its development, provide support through forums and online communities, and share resources such as tutorials, documentation, and pre-trained models.

- In summary, PyTorch is a versatile and powerful library for deep learning, offering dynamic graph construction, ease of use, flexibility, and a vibrant community ecosystem. Its intuitive interface and rich feature set make it an excellent choice for researchers, educators, and practitioners alike.

6.3 FFMPEG

FFmpeg is a cross-platform open-source multimedia framework that provides a comprehensive suite of tools and libraries for handling multimedia data. It enables users to encode, decode, transcode, mux, demux, stream, filter, and play various audio and video formats with ease. Here's an overview:

At its core, FFmpeg consists of a collection of libraries and command-line tools that allow users to manipulate multimedia files in virtually any format. These tools include ffmpeg for converting and transcoding, ffplay for playback, ffprobe for analyzing multimedia streams, and many others.

One of FFmpeg's key strengths is its support for a wide range of audio and video codecs, containers, and protocols. It can handle popular formats like MP3, AAC, H.264, and MPEG-4, as well as less common formats and proprietary codecs.

FFmpeg's modular architecture and extensive documentation make it highly customizable and adaptable to a variety of use cases. Developers can leverage its libraries to build custom multimedia applications or integrate multimedia functionality into existing projects.

Additionally, FFmpeg's command-line interface makes it accessible to users of all skill levels, allowing them to perform complex multimedia tasks with simple commands. This accessibility has contributed to FFmpeg's popularity among multimedia professionals, hobbyists, and developers alike.

Furthermore, FFmpeg is actively maintained and developed by a dedicated community of contributors, ensuring that it remains up-to-date with the latest audio and video technologies and standards.

In summary, FFmpeg is a versatile and powerful multimedia framework that provides essential tools and libraries for handling audio and video processing tasks. Its wide format support, modular architecture, ease of use, and active community make it an invaluable resource for anyone working with multimedia content.

6.4 python Conclusion

In conclusion, Python stands out as a powerful and versatile programming language, offering numerous benefits that cater to the needs of developers across various domains. Its emphasis on readability, combined with support for multiple programming paradigms, makes it accessible to beginners while providing advanced capabilities for experienced programmers. The extensive ecosystem of libraries and frameworks accelerates development and enables the creation of complex applications with ease. Moreover, Python's vibrant community fosters collaboration, knowledge-sharing, and continuous improvement, further enhancing its appeal. With its cross-platform compatibility and seamless integration capabilities, Python emerges as a top choice for building innovative solutions across a wide range of industries and disciplines. Its enduring popularity and widespread adoption underscore its status as a cornerstone of modern software development.

CHAPTER 7

SOFTWARE TESTING

7.1 Functional Testing

Text-to-Speech Synthesis: This aspect of functional testing evaluates how accurately the system converts input text into spoken words. Test cases are designed to cover a wide range of linguistic features, including vocabulary, grammar, punctuation, and language-specific nuances.

Voice Generation: Functional testing verifies the system's ability to generate voices that closely match the desired characteristics, such as pitch, tone, accent, and emotion. This involves testing different voice styles and ensuring consistency and fidelity across various inputs.

Voice Conversion: Voice conversion testing assesses the system's capability to transform one voice into another while preserving the linguistic content and maintaining naturalness. Test cases may involve converting voices between different genders, ages, or accents to evaluate the accuracy and effectiveness of the conversion process.

7.2 Quality Assessment:

Perceptual Quality: Quality assessment testing involves subjective evaluation by human listeners to assess the perceived quality of synthesized voices. Listeners provide feedback on factors such as clarity, naturalness, fluency, and overall satisfaction, helping to identify areas for improvement.

Objective Metrics: In addition to subjective evaluation, objective metrics such as Mean Opinion Score (MOS), Signal-to-Noise Ratio (SNR), and Mel

Cepstral Distortion (MCD) may be used to quantitatively measure the quality of synthesized speech. These metrics provide standardized criteria for assessing the fidelity and intelligibility of the generated voices.

7.3 Robustness Testing:

Speaking Variability: Robustness testing evaluates how well the system performs under variations in speaking rate, volume, and rhythm. Test cases may include rapid speech, whispered speech, and speech with exaggerated intonation to assess the system's ability to handle different speaking styles.

Accents and Dialects: Testing with diverse accents and dialects helps identify potential biases or limitations in the system's voice modeling and adaptation capabilities. By exposing the system to a wide range of linguistic variations, developers can ensure that it can accurately synthesize voices from different cultural and linguistic backgrounds.

7.4 Security Testing:

Authentication Mechanisms: Security testing verifies the effectiveness of authentication mechanisms implemented in the voice cloning system to prevent unauthorized access. This may include testing password protection, multi-factor authentication, and biometric authentication methods to ensure secure user authentication.

Privacy Controls: Testing privacy controls assesses the system's ability to protect sensitive user data, such as voice recordings and personal information, from unauthorized access or disclosure. This involves evaluating data encryption, access controls, and data anonymization techniques to safeguard

user privacy.

7.5 Compatibility Testing:

Cross-Platform Compatibility: Compatibility testing verifies that the voice cloning system operates consistently across different platforms, including desktop computers, mobile devices, and web browsers. Test cases may involve testing compatibility with various operating systems (e.g., Windows, macOS, Linux), browser versions, and screen resolutions to ensure a seamless user experience.

7.6 Third-Party Integration: Testing third-party integration ensures that the voice cloning system interoperates smoothly with other software applications, APIs, and services. This includes testing compatibility with speech recognition systems, virtual assistants, and communication platforms to facilitate seamless integration and interoperability.

7.7 Performance Testing:

Speed and Responsiveness: Performance testing assesses the system's speed and responsiveness under typical and peak loads. Test cases may involve measuring response times for voice synthesis requests, playback latency, and system resource utilization to identify performance bottlenecks and optimize system performance.

Scalability: Testing scalability evaluates how well the voice cloning system scales to accommodate increasing workloads and user demand. This includes testing performance under varying concurrency levels, data volumes, and user traffic to ensure that the system can handle growing demand without

degradation in performance.

7.8 User Acceptance Testing (UAT):

User Feedback: UAT involves soliciting feedback from end-users to evaluate their satisfaction with the voice cloning system. Test cases may include usability testing, user surveys, and focus groups to gather insights into user preferences, pain points, and overall satisfaction with the system.

Usability Testin: t Usability testing assesses the system's ease of use, intuitiveness, and accessibility for end-users. This involves observing users as they interact with the system, identifying usability issues, and collecting feedback on user interface design, navigation, and functionality to improve user experience.

By conducting thorough testing using these methodologies, developers can ensure that voice cloning systems meet stringent quality standards, deliver superior performance, and provide a seamless and satisfying user experience.

CHAPTER 8

FUTURE ENHANCEMENT

Future enhancements in real-time voice cloning using neural fusion offer exciting prospects for advancing this technology's capabilities and applications. Key areas of improvement include:

1.Enhanced Realism: Refining neural fusion algorithms to better capture human speech nuances, improving the naturalness and expressiveness of synthesized voices.

2.Personalization: Tailoring synthesized voices to individual users through adaptive learning, providing more personalized interactions in various applications.

3.Real-Time Performance: Optimizing system performance and reducing latency to enable seamless, instantaneous voice synthesis in real-time applications.

4.Multimodal Integration: Integrating voice synthesis with other modalities like facial expressions and text inputs for more immersive user experiences in virtual and augmented reality environments.

5.Ethical Considerations: Developing robust privacy safeguards and transparency measures to address ethical concerns and ensure responsible deployment of voice cloning technology.

By addressing these areas, future enhancements can significantly enhance voice cloning technology, paving the way for more lifelike and engaging interactions across a wide range of domains.

CHAPTER 9

CONCLUSION:

In conclusion, real-time voice cloning with neural fusion represents a significant advancement in human-computer interaction technology. By leveraging neural fusion algorithms, this technology offers the potential to create highly realistic and personalized synthesized voices in real-time applications. With ongoing research and development, we can anticipate further improvements in naturalness, performance, and ethical considerations, paving the way for more immersive and engaging user experiences across diverse domains.

REFERENCES

[1] Zhou, Berrak Sisman, Rajib Rana,Bjom W.Schuller, Haizhoo li, “speech synthesis with mixed emotions.” *IEEE TRANSACTIONS ON AFFECTIVE COMPUTING, VOL. 14, NO. 4, OCTOBER-DECEMBER 2023*

[2] Rui Liu, Berrak Sisman, Guanglai Gao, Haizhoo Li, “decoding knowledge transfer for neural text to speech training.” *IEEE/ACM TRANSACTIONS ON AUDIO, SPEECH, AND LANGUAGE PROCESSING, VOL. 30, 2022*

[3] Rui Liu, Berrak Sisman, Guanglai Gao, Haizhoo Li, “expressive tts training with frame and style reconstruction loss” *IEEE/ACM TRANSACTIONS ON AUDIO, SPEECH, AND LANGUAGE PROCESSING, VOL. 29, 2021*

[4] Sashi Novitasari, Sakriani Sakti, Satoshi Nakamura, “a machine speech chain approach for dynamically adaptive lombard tts in static and dynamic noise environments” *IEEE/ACM TRANSACTIONS ON AUDIO, SPEECH, AND LANGUAGE PROCESSING, VOL. 30, 2022*

[5] Chefeng Miao, Qingying Zhu, Minchuan Chen, Jun Ma, Shaojun Wang, Jing Xiao, “efficiency TTS 2: variational end to end text to speech synthesis and voice conversation” *IEEE/ACM Transactions on Audio, Speech and Language Processing.*

[6] Yimin He, Xiaoyong Lu*, Jingyi Yuan, Tao Pan, and Yafan Wang, “Depressive Tendency Recognition by Fusing Speech and Text Features: A Comparative Analysis,” Northwest Normal University Lanzhou, Gansu, China

[7] Liping Chen, Yan Deng, Xi Wang, Frank K. Soong, Lei He, “SPEECH BERT EMBEDDING FOR IMPROVING PROSODY IN NEURAL TTS”

[8] Min-Jae Hwang1, Ryuichi Yamamoto2, Eunwoo Song3 and Jae-Min Kim3, “TTS-BY-TTS: TTS-DRIVEN DATA AUGMENTATION FOR FAST AND HIGH-QUALITY SPEECH SYNTHESIS”

[9] Bo Chen, Chenpeng Du and Kai Yu, “Neural Fusion for Voice Cloning” *IEEE/ACM TRANSACTIONS ON AUDIO, SPEECH, AND LANGUAGE PROCESSING, VOL. 30, 2022*

[10] Swetha Tanamala, Jeena J Prakash and Hema A, “A Semi-Automatic Method for Transcription Error Correction for Indian Language TTS Systems”, *2017 Twenty-third National Conference on Communications (NCC)*

[11] Victoria Mingote , Pablo Gimeno , Luis Vicente , Sameer Khurana , Antoine Laurent , and Jarod Duret “ Direct Text to Speech Translation System Using Acoustic Units”

[12] Yeonghyeon Lee, Inmo Yeon, Juhan Nam, Joon Son Chung “VOICELDM: TEXT-TO-SPEECH WITH ENVIRONMENTAL CONTEXT

APPENDIX A

SAMPLE SOURCE CODE

demo_toolbox.py :

```
import argparse
import os
from pathlib import Path
from toolbox import Toolbox
from utils.argutils import print_args
from utils.default_models import ensure_default_models


if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description="Runs the toolbox.",
        formatter_class=argparse.ArgumentDefaultsHelpFormatter
    )

    parser.add_argument("-d", "--datasets_root", type=Path, help= \
        "Path to the directory containing your datasets. See toolbox/__init__.py for a list of "
        "supported datasets.", default=None)
    parser.add_argument("-m", "--models_dir", type=Path, default="saved_models",
                        help="Directory containing all saved models")
    parser.add_argument("--cpu", action="store_true", help=\
        "If True, all inference will be done on CPU")
    parser.add_argument("--seed", type=int, default=None, help=\
        "Optional random number seed value to make toolbox deterministic.")
    args = parser.parse_args()
    arg_dict = vars(args)
    print_args(args, parser)

    # Hide GPUs from Pytorch to force CPU processing
    if arg_dict.pop("cpu"):
        os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
```

```
    # Remind the user to download pretrained models if needed
    ensure_default_models(args.models_dir)

    # Launch the toolbox
    Toolbox(**arg_dict)
```

argulits.py :

```
from pathlib import Path
import numpy as np
import argparse

_type_priorities = [    # In decreasing order
    Path,
    str,
    int,
    float,
    bool,
]

def _priority(o):
    p = next((i for i, t in enumerate(_type_priorities) if type(o) is t), None)
    if p is not None:
        return p
    p = next((i for i, t in enumerate(_type_priorities) if isinstance(o, t)), None)
    if p is not None:
        return p
    return len(_type_priorities)

def print_args(args: argparse.Namespace, parser=None):
    args = vars(args)
    if parser is None:
        priorities = list(map(_priority, args.values()))
    else:
        all_params = [a.dest for g in parser._action_groups for a in g._group_actions ]
```

```
        priority = lambda p: all_params.index(p) if p in all_params else len(all_params)
        priorities = list(map(priority, args.keys()))

    pad = max(map(len, args.keys())) + 3
    indices = np.lexsort((list(args.keys()), priorities))
    items = list(args.items())

    print("Arguments:")
    for i in indices:
        param, value = items[i]
        print("    {0}:{1}{2}".format(param, ' ' * (pad - len(param)), value))
    print("")
```

logmmse.py

```
# The MIT License (MIT)
#
# Copyright (c) 2015 braindead
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in all
# copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
```

```
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE
# SOFTWARE.
#
#
# This code was extracted from the logmmse package (https://pypi.org/project/logmmse/) and I
# simply modified the interface to meet my needs.


import numpy as np
import math
from scipy.special import expn
from collections import namedtuple

NoiseProfile = namedtuple("NoiseProfile", "sampling_rate window_size len1 len2 win n_fft
noise_mu2")


def profile_noise(noise, sampling_rate, window_size=0):
    """
    Creates a profile of the noise in a given waveform.

    :param noise: a waveform containing noise ONLY, as a numpy array of floats or ints.
    :param sampling_rate: the sampling rate of the audio
    :param window_size: the size of the window the logmmse algorithm operates on. A default
value
    will be picked if left as 0.
    :return: a NoiseProfile object
    """
    noise, dtype = to_float(noise)
    noise += np.finfo(np.float64).eps

    if window_size == 0:
        window_size = int(math.floor(0.02 * sampling_rate))
```

```
    if window_size % 2 == 1:
        window_size = window_size + 1

    perc = 50
    len1 = int(math.floor(window_size * perc / 100))
    len2 = int(window_size - len1)

    win = np.hanning(window_size)
    win = win * len2 / np.sum(win)
    n_fft = 2 * window_size

    noise_mean = np.zeros(n_fft)
    n_frames = len(noise) // window_size
    for j in range(0, window_size * n_frames, window_size):
        noise_mean += np.absolute(np.fft.fft(win * noise[j:j + window_size], n_fft, axis=0))
    noise_mu2 = (noise_mean / n_frames) ** 2

    return NoiseProfile(sampling_rate, window_size, len1, len2, win, n_fft, noise_mu2)


def denoise(wav, noise_profile: NoiseProfile, eta=0.15):
    """
    Cleans the noise from a speech waveform given a noise profile. The waveform must have the
    same sampling rate as the one used to create the noise profile.

    :param wav: a speech waveform as a numpy array of floats or ints.
    :param noise_profile: a NoiseProfile object that was created from a similar (or a segment of
    the same) waveform.
    :param eta: voice threshold for noise update. While the voice activation detection value is
    below this threshold, the noise profile will be continuously updated throughout the audio.
    Set to 0 to disable updating the noise profile.
    :return: the clean wav as a numpy array of floats or ints of the same length.
    """
    wav, dtype = to_float(wav)
    wav += np.finfo(np.float64).eps
    p = noise_profile
```

```
nframes = int(math.floor(len(wav) / p.len2) - math.floor(p.window_size / p.len2))
x_final = np.zeros(nframes * p.len2)

aa = 0.98
mu = 0.98
ksi_min = 10 ** (-25 / 10)

x_old = np.zeros(p.len1)
xk_prev = np.zeros(p.len1)
noise_mu2 = p.noise_mu2
for k in range(0, nframes * p.len2, p.len2):
    insign = p.win * wav[k:k + p.window_size]

    spec = np.fft.fft(insign, p.n_fft, axis=0)
    sig = np.absolute(spec)
    sig2 = sig ** 2

    gammak = np.minimum(sig2 / noise_mu2, 40)

    if xk_prev.all() == 0:
        ksi = aa + (1 - aa) * np.maximum(gammak - 1, 0)
    else:
        ksi = aa * xk_prev / noise_mu2 + (1 - aa) * np.maximum(gammak - 1, 0)
        ksi = np.maximum(ksi_min, ksi)

    log_sigma_k = gammak * ksi/(1 + ksi) - np.log(1 + ksi)
    vad_decision = np.sum(log_sigma_k) / p.window_size
    if vad_decision < eta:
        noise_mu2 = mu * noise_mu2 + (1 - mu) * sig2

    a = ksi / (1 + ksi)
    vk = a * gammak
    ei_vk = 0.5 * expn(1, np.maximum(vk, 1e-8))
    hw = a * np.exp(ei_vk)
    sig = sig * hw
    xk_prev = sig ** 2
    xi_w = np.fft.ifft(hw * spec, p.n_fft, axis=0)
```

```
        xi_w = np.real(xi_w)

        x_final[k:k + p.len2] = x_old + xi_w[0:p.len1]
        x_old = xi_w[p.len1:p.window_size]

    output = from_float(x_final, dtype)
    output = np.pad(output, (0, len(wav) - len(output)), mode="constant")
    return output


## Alternative VAD algorithm to webrctvad. It has the advantage of not requiring to install that
## darn package and it also works for any sampling rate. Maybe I'll eventually use it instead of
## webrctvad
# def vad(wav, sampling_rate, eta=0.15, window_size=0):
#     """
#     TODO: fix doc
#     Creates a profile of the noise in a given waveform.
#
#     :param wav: a waveform containing noise ONLY, as a numpy array of floats or ints.
#     :param sampling_rate: the sampling rate of the audio
#     :param window_size: the size of the window the logmmse algorithm operates on. A default value
#     will be picked if left as 0.
#     :param eta: voice threshold for noise update. While the voice activation detection value is
#     below this threshold, the noise profile will be continuously updated throughout the audio.
#     Set to 0 to disable updating the noise profile.
#     """
#     wav, dtype = to_float(wav)
#     wav += np.finfo(np.float64).eps
#
#     if window_size == 0:
#         window_size = int(math.floor(0.02 * sampling_rate))
#
#     if window_size % 2 == 1:
#         window_size = window_size + 1
#
#     perc = 50
```

```
#     len1 = int(math.floor(window_size * perc / 100))
#     len2 = int(window_size - len1)
#
#     win = np.hanning(window_size)
#     win = win * len2 / np.sum(win)
#     n_fft = 2 * window_size
#
#     wav_mean = np.zeros(n_fft)
#     n_frames = len(wav) // window_size
#     for j in range(0, window_size * n_frames, window_size):
#         wav_mean += np.absolute(np.fft.fft(win * wav[j:j + window_size], n_fft, axis=0))
#     noise_mu2 = (wav_mean / n_frames) ** 2
#
#     wav, dtype = to_float(wav)
#     wav += np.finfo(np.float64).eps
#
#     nframes = int(math.floor(len(wav) / len2) - math.floor(window_size / len2))
#     vad = np.zeros(nframes * len2, dtype=np.bool)
#
#     aa = 0.98
#     mu = 0.98
#     ksi_min = 10 ** (-25 / 10)
#
#     xk_prev = np.zeros(len1)
#     noise_mu2 = noise_mu2
#     for k in range(0, nframes * len2, len2):
#         insign = win * wav[k:k + window_size]
#
#         spec = np.fft.fft(insign, n_fft, axis=0)
#         sig = np.absolute(spec)
#         sig2 = sig ** 2
#
#         gammak = np.minimum(sig2 / noise_mu2, 40)
#
#         if xk_prev.all() == 0:
#             ksi = aa + (1 - aa) * np.maximum(gammak - 1, 0)
#         else:
```

```
#             ksi = aa * xk_prev / noise_mu2 + (1 - aa) * np.maximum(gammak - 1, 0)
#             ksi = np.maximum(ksi_min, ksi)
#
#         log_sigma_k = gammak * ksi / (1 + ksi) - np.log(1 + ksi)
#         vad_decision = np.sum(log_sigma_k) / window_size
#         if vad_decision < eta:
#             noise_mu2 = mu * noise_mu2 + (1 - mu) * sig2
#         print(vad_decision)
#
#         a = ksi / (1 + ksi)
#         vk = a * gammak
#         ei_vk = 0.5 * expn(1, np.maximum(vk, 1e-8))
#         hw = a * np.exp(ei_vk)
#         sig = sig * hw
#         xk_prev = sig ** 2
#
#         vad[k:k + len2] = vad_decision >= eta
#
#     vad = np.pad(vad, (0, len(wav) - len(vad)), mode="constant")
#     return vad


def to_float(_input):
    if _input.dtype == np.float64:
        return _input, _input.dtype
    elif _input.dtype == np.float32:
        return _input.astype(np.float64), _input.dtype
    elif _input.dtype == np.uint8:
        return (_input - 128) / 128., _input.dtype
    elif _input.dtype == np.int16:
        return _input / 32768., _input.dtype
    elif _input.dtype == np.int32:
        return _input / 2147483648., _input.dtype
    raise ValueError('Unsupported wave file format')
def from_float(_input, dtype):
    if dtype == np.float64:
        return _input, np.float64
```

```
    elif dtype == np.float32:
        return _input.astype(np.float32)
    elif dtype == np.uint8:
        return ((_input * 128) + 128).astype(np.uint8)
    elif dtype == np.int16:
        return (_input * 32768).astype(np.int16)
    elif dtype == np.int32:
        print(_input)
        return (_input * 2147483648).astype(np.int32)
    raise ValueError('Unsupported wave file format
```

Encoder_preprocessor.py

```
from encoder.preprocess import preprocess_librispeech, preprocess_voxceleb1,
preprocess_voxceleb2
from utils.argutils import print_args
from pathlib import Path
import argparse



if __name__ == "__main__":
    class MyFormatter(argparse.ArgumentDefaultsHelpFormatter,
argparse.RawDescriptionHelpFormatter):
        pass


    parser = argparse.ArgumentParser(
        description="Preprocesses audio files from datasets, encodes them as mel spectrograms and
"
                    "writes them to the disk. This will allow you to train the encoder. The "
                    "datasets required are at least one of VoxCeleb1, VoxCeleb2 and LibriSpeech. "
                    "Ideally, you should have all three. You should extract them as they are "
                    "after having downloaded them and put them in a same directory, e.g.:\n"
                    "-[datasets_root]\n"
                    "  -LibriSpeech\n"
                    "    -train-other-500\n"
                    "  -VoxCeleb1\n"
                    "    -wav\n"
```

```
            "    -vox1_meta.csv\n"
            "  -VoxCeleb2\n"
            "    -dev",
    formatter_class=MyFormatter
)
parser.add_argument("datasets_root", type=Path, help=\
    "Path to the directory containing your LibriSpeech/TTS and VoxCeleb datasets.")
parser.add_argument("-o", "--out_dir", type=Path, default=argparse.SUPPRESS, help=\
    "Path to the output directory that will contain the mel spectrograms. If left out, "
    "defaults to <datasets_root>/SV2TTS/encoder/")
parser.add_argument("-d", "--datasets", type=str,
                    default="librispeech_other,voxceleb1,voxceleb2", help=\
    "Comma-separated list of the name of the datasets you want to preprocess. Only the train "
    "set of these datasets will be used. Possible names: librispeech_other, voxceleb1, "
    "voxceleb2.")
parser.add_argument("-s", "--skip_existing", action="store_true", help=\
    "Whether to skip existing output files with the same name. Useful if this script was "
    "interrupted.")
parser.add_argument("--no_trim", action="store_true", help=\
    "Preprocess audio without trimming silences (not recommended).")
args = parser.parse_args()

# Verify webrtcvad is available
if not args.no_trim:
    try:
        import webrtcvad
    except:
        raise ModuleNotFoundError("Package 'webrtcvad' not found. This package enables "
            "noise removal and is recommended. Please install and try again. If installation fails, "
            "use --no_trim to disable this error message.")
del args.no_trim

# Process the arguments
args.datasets = args.datasets.split(",")
if not hasattr(args, "out_dir"):
    args.out_dir = args.datasets_root.joinpath("SV2TTS", "encoder")
assert args.datasets_root.exists()
```

```
    args.out_dir.mkdir(exist_ok=True, parents=True)

    # Preprocess the datasets
    print_args(args, parser)
    preprocess_func = {
        "librispeech_other": preprocess_librispeech,
        "voxceleb1": preprocess_voxceleb1,
        "voxceleb2": preprocess_voxceleb2,
    }
    args = vars(args)
    for dataset in args.pop("datasets"):
        print("Preprocessing %s" % dataset)
        preprocess_func[dataset](**args)
```

encoder_train.py

```
from utils.argutils import print_args
from encoder.train import train
from pathlib import Path
import argparse


if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        description="Trains the speaker encoder. You must have run encoder_preprocess.py first.",
        formatter_class=argparse.ArgumentDefaultsHelpFormatter
    )

    parser.add_argument("run_id", type=str, help= \
        "Name for this model. By default, training outputs will be stored to
saved_models/<run_id>/. If a model state "
        "from the same run ID was previously saved, the training will restart from there. Pass -f to
overwrite saved "
        "states and restart from scratch.")
    parser.add_argument("clean_data_root", type=Path, help= \
        "Path to the output directory of encoder_preprocess.py. If you left the default "
        "output directory when preprocessing, it should be <datasets_root>/SV2TTS/encoder/.")
```

```
    parser.add_argument("-m", "--models_dir", type=Path, default="saved_models", help=\
        "Path to the root directory that contains all models. A directory <run_name> will be created
under this root."
        "It will contain the saved model weights, as well as backups of those weights and plots
generated during "
        "training.")
    parser.add_argument("-v", "--vis_every", type=int, default=10, help= \
        "Number of steps between updates of the loss and the plots.")
    parser.add_argument("-u", "--umap_every", type=int, default=100, help= \
        "Number of steps between updates of the umap projection. Set to 0 to never update the "
        "projections.")
    parser.add_argument("-s", "--save_every", type=int, default=500, help= \
        "Number of steps between updates of the model on the disk. Set to 0 to never save the "
        "model.")
    parser.add_argument("-b", "--backup_every", type=int, default=7500, help= \
        "Number of steps between backups of the model. Set to 0 to never make backups of the "
        "model.")
    parser.add_argument("-f", "--force_restart", action="store_true", help= \
        "Do not load any saved model.")
    parser.add_argument("--visdom_server", type=str, default="http://localhost")
    parser.add_argument("--no_visdom", action="store_true", help= \
        "Disable visdom.")
    args = parser.parse_args()

    # Run the training
    print_args(args, parser)
    train(**vars(args))
```

audio.py

```
import math
import numpy as np
import librosa
import vocoder.hparams as hp
from scipy.signal import lfilter
import soundfile as sf
```

```
def label_2_float(x, bits) :
    return 2 * x / (2**bits - 1.) - 1.


def float_2_label(x, bits) :
    assert abs(x).max() <= 1.0
    x = (x + 1.) * (2**bits - 1) / 2
    return x.clip(0, 2**bits - 1)


def load_wav(path) :
    return librosa.load(str(path), sr=hp.sample_rate)[0]


def save_wav(x, path) :
    sf.write(path, x.astype(np.float32), hp.sample_rate)


def split_signal(x) :
    unsigned = x + 2**15
    coarse = unsigned // 256
    fine = unsigned % 256
    return coarse, fine


def combine_signal(coarse, fine) :
    return coarse * 256 + fine - 2**15


def encode_16bits(x) :
    return np.clip(x * 2**15, -2**15, 2**15 - 1).astype(np.int16)


mel_basis = None
```

```
def linear_to_mel(spectrogram):
    global mel_basis
    if mel_basis is None:
        mel_basis = build_mel_basis()
    return np.dot(mel_basis, spectrogram)


def build_mel_basis():
    return librosa.filters.mel(hp.sample_rate, hp.n_fft, n_mels=hp.num_mels, fmin=hp.fmin)


def normalize(S):
    return np.clip((S - hp.min_level_db) / -hp.min_level_db, 0, 1)


def denormalize(S):
    return (np.clip(S, 0, 1) * -hp.min_level_db) + hp.min_level_db


def amp_to_db(x):
    return 20 * np.log10(np.maximum(1e-5, x))


def db_to_amp(x):
    return np.power(10.0, x * 0.05)


def spectrogram(y):
    D = stft(y)
    S = amp_to_db(np.abs(D)) - hp.ref_level_db
    return normalize(S)


def melspectrogram(y):
    D = stft(y)
    S = amp_to_db(linear_to_mel(np.abs(D)))
```

```
    return normalize(S)


def stft(y):
    return librosa.stft(y=y, n_fft=hp.n_fft, hop_length=hp.hop_length, win_length=hp.win_length)


def pre_emphasis(x):
    return lfilter([1, -hp.preemphasis], [1], x)


def de_emphasis(x):
    return lfilter([1], [1, -hp.preemphasis], x)


def encode_mu_law(x, mu) :
    mu = mu - 1
    fx = np.sign(x) * np.log(1 + mu * np.abs(x)) / np.log(1 + mu)
    return np.floor((fx + 1) / 2 * mu + 0.5)


def decode_mu_law(y, mu, from_labels=True) :
    if from_labels:
        y = label_2_float(y, math.log2(mu))
    mu = mu - 1
    x = np.sign(y) / mu * ((1 + mu) ** np.abs(y) - 1)
    return x
```

display.py

```
import time
import numpy as np
import sys


def progbar(i, n, size=16):
    done = (i * size) // n
```

```
    bar = ''
    for i in range(size):
        bar += '▓' if i <= done else '░'
    return bar


def stream(message) :
    try:
        sys.stdout.write("\r{%s}" % message)
    except:
        #Remove non-ASCII characters from message
        message = ''.join(i for i in message if ord(i)<128)
        sys.stdout.write("\r{%s}" % message)


def simple_table(item_tuples) :

    border_pattern = '+---------------------------------------'
    whitespace = '                                        '

    headings, cells, = [], []

    for item in item_tuples :

        heading, cell = str(item[0]), str(item[1])

        pad_head = True if len(heading) < len(cell) else False

        pad = abs(len(heading) - len(cell))
        pad = whitespace[:pad]

        pad_left = pad[:len(pad)//2]
        pad_right = pad[len(pad)//2:]

        if pad_head :
            heading = pad_left + heading + pad_right
        else :
```

```
            cell = pad_left + cell + pad_right

        headings += [heading]
        cells += [cell]

    border, head, body = '', '', ''

    for i in range(len(item_tuples)) :

        temp_head = f'| {headings[i]} '
        temp_body = f'| {cells[i]} '

        border += border_pattern[:len(temp_head)]
        head += temp_head
        body += temp_body

        if i == len(item_tuples) - 1 :
            head += '|'
            body += '|'
            border += '+'

    print(border)
    print(head)
    print(border)
    print(body)
    print(border)
    print(' ')


def time_since(started) :
    elapsed = time.time() - started
    m = int(elapsed // 60)
    s = int(elapsed % 60)
    if m >= 60 :
        h = int(m // 60)
        m = m % 60
        return f'{h}h {m}m {s}s'
```

```
    else :
        return f'{m}m {s}s'


def save_attention(attn, path):
    import matplotlib.pyplot as plt

    fig = plt.figure(figsize=(12, 6))
    plt.imshow(attn.T, interpolation='nearest', aspect='auto')
    fig.savefig(f'{path}.png', bbox_inches='tight')
    plt.close(fig)


def save_spectrogram(M, path, length=None):
    import matplotlib.pyplot as plt

    M = np.flip(M, axis=0)
    if length : M = M[:, :length]
    fig = plt.figure(figsize=(12, 6))
    plt.imshow(M, interpolation='nearest', aspect='auto')
    fig.savefig(f'{path}.png', bbox_inches='tight')
    plt.close(fig)


def plot(array):
    import matplotlib.pyplot as plt

    fig = plt.figure(figsize=(30, 5))
    ax = fig.add_subplot(111)
    ax.xaxis.label.set_color('grey')
    ax.yaxis.label.set_color('grey')
    ax.xaxis.label.set_fontsize(23)
    ax.yaxis.label.set_fontsize(23)
    ax.tick_params(axis='x', colors='grey', labelsize=23)
    ax.tick_params(axis='y', colors='grey', labelsize=23)
    plt.plot(array)
```

```
def plot_spec(M):
    import matplotlib.pyplot as plt

    M = np.flip(M, axis=0)
    plt.figure(figsize=(18,4))
    plt.imshow(M, interpolation='nearest', aspect='auto')
    plt.show()
```

distribution.py

```
import numpy as np
import torch
import torch.nn.functional as F


def log_sum_exp(x):
    """ numerically stable log_sum_exp implementation that prevents overflow """
    # TF ordering
    axis = len(x.size()) - 1
    m, _ = torch.max(x, dim=axis)
    m2, _ = torch.max(x, dim=axis, keepdim=True)
    return m + torch.log(torch.sum(torch.exp(x - m2), dim=axis))


# It is adapted from
https://github.com/r9y9/wavenet_vocoder/blob/master/wavenet_vocoder/mixture.py
def discretized_mix_logistic_loss(y_hat, y, num_classes=65536,
                                  log_scale_min=None, reduce=True):
    if log_scale_min is None:
        log_scale_min = float(np.log(1e-14))
    y_hat = y_hat.permute(0,2,1)
    assert y_hat.dim() == 3
    assert y_hat.size(1) % 3 == 0
    nr_mix = y_hat.size(1) // 3

    # (B x T x C)
```

```
y_hat = y_hat.transpose(1, 2)

# unpack parameters. (B, T, num_mixtures) x 3
logit_probs = y_hat[:, :, :nr_mix]
means = y_hat[:, :, nr_mix:2 * nr_mix]
log_scales = torch.clamp(y_hat[:, :, 2 * nr_mix:3 * nr_mix], min=log_scale_min)

# B x T x 1 -> B x T x num_mixtures
y = y.expand_as(means)

centered_y = y - means
inv_stdv = torch.exp(-log_scales)
plus_in = inv_stdv * (centered_y + 1. / (num_classes - 1))
cdf_plus = torch.sigmoid(plus_in)
min_in = inv_stdv * (centered_y - 1. / (num_classes - 1))
cdf_min = torch.sigmoid(min_in)

# log probability for edge case of 0 (before scaling)
# equivalent: torch.log(F.sigmoid(plus_in))
log_cdf_plus = plus_in - F.softplus(plus_in)

# log probability for edge case of 255 (before scaling)
# equivalent: (1 - F.sigmoid(min_in)).log()
log_one_minus_cdf_min = -F.softplus(min_in)

# probability for all other cases
cdf_delta = cdf_plus - cdf_min

mid_in = inv_stdv * centered_y
# log probability in the center of the bin, to be used in extreme cases
# (not actually used in our code)
log_pdf_mid = mid_in - log_scales - 2. * F.softplus(mid_in)

# tf equivalent
"""
log_probs = tf.where(x < -0.999, log_cdf_plus,
                     tf.where(x > 0.999, log_one_minus_cdf_min,
```

```
                        tf.where(cdf_delta > 1e-5,
                                 tf.log(tf.maximum(cdf_delta, 1e-12)),
                                 log_pdf_mid - np.log(127.5))))
    """
    # TODO: cdf_delta <= 1e-5 actually can happen. How can we choose the value
    # for num_classes=65536 case? 1e-7? not sure..
    inner_inner_cond = (cdf_delta > 1e-5).float()

    inner_inner_out = inner_inner_cond * \
        torch.log(torch.clamp(cdf_delta, min=1e-12)) + \
        (1. - inner_inner_cond) * (log_pdf_mid - np.log((num_classes - 1) / 2))
    inner_cond = (y > 0.999).float()
    inner_out = inner_cond * log_one_minus_cdf_min + (1. - inner_cond) * inner_inner_out
    cond = (y < -0.999).float()
    log_probs = cond * log_cdf_plus + (1. - cond) * inner_out

    log_probs = log_probs + F.log_softmax(logit_probs, -1)

    if reduce:
        return -torch.mean(log_sum_exp(log_probs))
    else:
        return -log_sum_exp(log_probs).unsqueeze(-1)


def sample_from_discretized_mix_logistic(y, log_scale_min=None):
    """
    Sample from discretized mixture of logistic distributions
    Args:
        y (Tensor): B x C x T
        log_scale_min (float): Log scale minimum value
    Returns:
        Tensor: sample in range of [-1, 1].
    """
    if log_scale_min is None:
        log_scale_min = float(np.log(1e-14))
    assert y.size(1) % 3 == 0
    nr_mix = y.size(1) // 3
```

```
    # B x T x C
    y = y.transpose(1, 2)
    logit_probs = y[:, :, :nr_mix]

    # sample mixture indicator from softmax
    temp = logit_probs.data.new(logit_probs.size()).uniform_(1e-5, 1.0 - 1e-5)
    temp = logit_probs.data - torch.log(- torch.log(temp))
    _, argmax = temp.max(dim=-1)

    # (B, T) -> (B, T, nr_mix)
    one_hot = to_one_hot(argmax, nr_mix)
    # select logistic parameters
    means = torch.sum(y[:, :, nr_mix:2 * nr_mix] * one_hot, dim=-1)
    log_scales = torch.clamp(torch.sum(
        y[:, :, 2 * nr_mix:3 * nr_mix] * one_hot, dim=-1), min=log_scale_min)
    # sample from logistic & clip to interval
    # we don't actually round to the nearest 8bit value when sampling
    u = means.data.new(means.size()).uniform_(1e-5, 1.0 - 1e-5)
    x = means + torch.exp(log_scales) * (torch.log(u) - torch.log(1. - u))

    x = torch.clamp(torch.clamp(x, min=-1.), max=1.)

    return x


def to_one_hot(tensor, n, fill_with=1.):
    # we perform one hot encore with respect to the last axis
    one_hot = torch.FloatTensor(tensor.size() + (n,)).zero_()
    if tensor.is_cuda:
        one_hot = one_hot.cuda()
    one_hot.scatter_(len(tensor.size()), tensor.unsqueeze(-1), fill_with)
    return one_hot
```

genwavernn.py

```
from vocoder.models.fatchord_version import  WaveRNN
from vocoder.audio import *


def gen_testset(model: WaveRNN, test_set, samples, batched, target, overlap, save_path):
    k = model.get_step() // 1000

    for i, (m, x) in enumerate(test_set, 1):
        if i > samples:
            break

        print('\n| Generating: %i/%i' % (i, samples))

        x = x[0].numpy()

        bits = 16 if hp.voc_mode == 'MOL' else hp.bits

        if hp.mu_law and hp.voc_mode != 'MOL' :
            x = decode_mu_law(x, 2**bits, from_labels=True)
        else :
            x = label_2_float(x, bits)

        save_wav(x, save_path.joinpath("%dk_steps_%d_target.wav" % (k, i)))

        batch_str = "gen_batched_target%d_overlap%d" % (target, overlap) if batched else \
            "gen_not_batched"
        save_str = save_path.joinpath("%dk_steps_%d_%s.wav" % (k, i, batch_str))

        wav = model.generate(m, batched, target, overlap, hp.mu_law)
        save_wav(wav, save_str)
```

hparam.py

```
from synthesizer.hparams import hparams as _syn_hp


# Audio settings------------------------------------------------------------------------
# Match the values of the synthesizer
sample_rate = _syn_hp.sample_rate
n_fft = _syn_hp.n_fft
num_mels = _syn_hp.num_mels
hop_length = _syn_hp.hop_size
win_length = _syn_hp.win_size
fmin = _syn_hp.fmin
min_level_db = _syn_hp.min_level_db
ref_level_db = _syn_hp.ref_level_db
mel_max_abs_value = _syn_hp.max_abs_value
preemphasis = _syn_hp.preemphasis
apply_preemphasis = _syn_hp.preemphasize

bits = 9                        # bit depth of signal
mu_law = True                       # Recommended to suppress noise if using raw bits in
hp.voc_mode
                                # below


# WAVERNN / VOCODER --------------------------------------------------------------------------
voc_mode = 'RAW'                    # either 'RAW' (softmax on raw bits) or 'MOL' (sample from
# mixture of logistics)
voc_upsample_factors = (5, 5, 8)    # NB - this needs to correctly factorise hop_length
voc_rnn_dims = 512
voc_fc_dims = 512
voc_compute_dims = 128
voc_res_out_dims = 128
voc_res_blocks = 10

# Training
voc_batch_size = 100
```

```
voc_lr = 1e-4
voc_gen_at_checkpoint = 5          # number of samples to generate at each checkpoint
voc_pad = 2                        # this will pad the input so that the resnet can 'see' wider
                                   # than input length
voc_seq_len = hop_length * 5       # must be a multiple of hop_length

# Generating / Synthesizing
voc_gen_batched = True             # very fast (realtime+) single utterance batched generation
voc_target = 8000                  # target number of samples to be generated in each batch entry
voc_overlap = 400                  # number of samples for crossfading between batches
```

interference.py

```
from vocoder.models.fatchord_version import WaveRNN
from vocoder import hparams as hp
import torch


_model = None   # type: WaveRNN

def load_model(weights_fpath, verbose=True):
    global _model, _device

    if verbose:
        print("Building Wave-RNN")
    _model = WaveRNN(
        rnn_dims=hp.voc_rnn_dims,
        fc_dims=hp.voc_fc_dims,
        bits=hp.bits,
        pad=hp.voc_pad,
        upsample_factors=hp.voc_upsample_factors,
        feat_dims=hp.num_mels,
        compute_dims=hp.voc_compute_dims,
        res_out_dims=hp.voc_res_out_dims,
        res_blocks=hp.voc_res_blocks,
        hop_length=hp.hop_length,
```

```
        sample_rate=hp.sample_rate,
        mode=hp.voc_mode
    )

    if torch.cuda.is_available():
        _model = _model.cuda()
        _device = torch.device('cuda')
    else:
        _device = torch.device('cpu')

    if verbose:
        print("Loading model weights at %s" % weights_fpath)
    checkpoint = torch.load(weights_fpath, _device)
    _model.load_state_dict(checkpoint['model_state'])
    _model.eval()


def is_loaded():
    return _model is not None


def infer_waveform(mel, normalize=True,  batched=True, target=8000, overlap=800,
                   progress_callback=None):
    """
    Infers the waveform of a mel spectrogram output by the synthesizer (the format must match
    that of the synthesizer!)

    :param normalize:
    :param batched:
    :param target:
    :param overlap:
    :return:
    """
    if _model is None:
        raise Exception("Please load Wave-RNN in memory before using it")

    if normalize:
```

```
        mel = mel / hp.mel_max_abs_value
    mel = torch.from_numpy(mel[None, ...])
    wav = _model.generate(mel, batched, target, overlap, hp.mu_law, progress_callback)
    return wav
```

train.py

```
import time
from pathlib import Path

import numpy as np
import torch
import torch.nn.functional as F
from torch import optim
from torch.utils.data import DataLoader

import vocoder.hparams as hp
from vocoder.display import stream, simple_table
from vocoder.distribution import discretized_mix_logistic_loss
from vocoder.gen_wavernn import gen_testset
from vocoder.models.fatchord_version import WaveRNN
from vocoder.vocoder_dataset import VocoderDataset, collate_vocoder


def train(run_id: str, syn_dir: Path, voc_dir: Path, models_dir: Path, ground_truth: bool,
save_every: int,
          backup_every: int, force_restart: bool):
    # Check to make sure the hop length is correctly factorised
    assert np.cumprod(hp.voc_upsample_factors)[-1] == hp.hop_length

    # Instantiate the model
    print("Initializing the model...")
    model = WaveRNN(
        rnn_dims=hp.voc_rnn_dims,
        fc_dims=hp.voc_fc_dims,
        bits=hp.bits,
        pad=hp.voc_pad,
```

```
    upsample_factors=hp.voc_upsample_factors,
    feat_dims=hp.num_mels,
    compute_dims=hp.voc_compute_dims,
    res_out_dims=hp.voc_res_out_dims,
    res_blocks=hp.voc_res_blocks,
    hop_length=hp.hop_length,
    sample_rate=hp.sample_rate,
    mode=hp.voc_mode
)

if torch.cuda.is_available():
    model = model.cuda()

# Initialize the optimizer
optimizer = optim.Adam(model.parameters())
for p in optimizer.param_groups:
    p["lr"] = hp.voc_lr
loss_func = F.cross_entropy if model.mode == "RAW" else discretized_mix_logistic_loss

# Load the weights
model_dir = models_dir / run_id
model_dir.mkdir(exist_ok=True)
weights_fpath = model_dir / "vocoder.pt"
if force_restart or not weights_fpath.exists():
    print("\nStarting the training of WaveRNN from scratch\n")
    model.save(weights_fpath, optimizer)
else:
    print("\nLoading weights at %s" % weights_fpath)
    model.load(weights_fpath, optimizer)
    print("WaveRNN weights loaded from step %d" % model.step)

# Initialize the dataset
metadata_fpath = syn_dir.joinpath("train.txt") if ground_truth else \
    voc_dir.joinpath("synthesized.txt")
mel_dir = syn_dir.joinpath("mels") if ground_truth else voc_dir.joinpath("mels_gta")
wav_dir = syn_dir.joinpath("audio")
dataset = VocoderDataset(metadata_fpath, mel_dir, wav_dir)
```

```
    test_loader = DataLoader(dataset, batch_size=1, shuffle=True)

    # Begin the training
    simple_table([('Batch size', hp.voc_batch_size),
                  ('LR', hp.voc_lr),
                  ('Sequence Len', hp.voc_seq_len)])

    for epoch in range(1, 350):
        data_loader = DataLoader(dataset, hp.voc_batch_size, shuffle=True, num_workers=2,
collate_fn=collate_vocoder)
        start = time.time()
        running_loss = 0.

        for i, (x, y, m) in enumerate(data_loader, 1):
            if torch.cuda.is_available():
                x, m, y = x.cuda(), m.cuda(), y.cuda()

            # Forward pass
            y_hat = model(x, m)
            if model.mode == 'RAW':
                y_hat = y_hat.transpose(1, 2).unsqueeze(-1)
            elif model.mode == 'MOL':
                y = y.float()
            y = y.unsqueeze(-1)

            # Backward pass
            loss = loss_func(y_hat, y)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            speed = i / (time.time() - start)
            avg_loss = running_loss / i

            step = model.get_step()
            k = step // 1000
```

```
            if backup_every != 0 and step % backup_every == 0 :
                model.checkpoint(model_dir, optimizer)

            if save_every != 0 and step % save_every == 0 :
                model.save(weights_fpath, optimizer)

            msg = f"| Epoch: {epoch} ({i}/{len(data_loader)}) | " \
                f"Loss: {avg_loss:.4f} | {speed:.1f} " \
                f"steps/s | Step: {k}k | "
            stream(msg)


        gen_testset(model, test_loader, hp.voc_gen_at_checkpoint, hp.voc_gen_batched,
                    hp.voc_target, hp.voc_overlap, model_dir)
        print("")
```

vocoder_dataset.py:

```
from torch.utils.data import Dataset
from pathlib import Path
from vocoder import audio
import vocoder.hparams as hp
import numpy as np
import torch


class VocoderDataset(Dataset):
    def __init__(self, metadata_fpath: Path, mel_dir: Path, wav_dir: Path):
        print("Using inputs from:\n\t%s\n\t%s\n\t%s" % (metadata_fpath, mel_dir, wav_dir))

        with metadata_fpath.open("r") as metadata_file:
            metadata = [line.split("|") for line in metadata_file]

        gta_fnames = [x[1] for x in metadata if int(x[4])]
        gta_fpaths = [mel_dir.joinpath(fname) for fname in gta_fnames]
        wav_fnames = [x[0] for x in metadata if int(x[4])]
```

```
    wav_fpaths = [wav_dir.joinpath(fname) for fname in wav_fnames]
    self.samples_fpaths = list(zip(gta_fpaths, wav_fpaths))

    print("Found %d samples" % len(self.samples_fpaths))

def __getitem__(self, index):
    mel_path, wav_path = self.samples_fpaths[index]

    # Load the mel spectrogram and adjust its range to [-1, 1]
    mel = np.load(mel_path).T.astype(np.float32) / hp.mel_max_abs_value

    # Load the wav
    wav = np.load(wav_path)
    if hp.apply_preemphasis:
        wav = audio.pre_emphasis(wav)
    wav = np.clip(wav, -1, 1)

    # Fix for missing padding   # TODO: settle on whether this is any useful
    r_pad =  (len(wav) // hp.hop_length + 1) * hp.hop_length - len(wav)
    wav = np.pad(wav, (0, r_pad), mode='constant')
    assert len(wav) >= mel.shape[1] * hp.hop_length
    wav = wav[:mel.shape[1] * hp.hop_length]
    assert len(wav) % hp.hop_length == 0

    # Quantize the wav
    if hp.voc_mode == 'RAW':
        if hp.mu_law:
            quant = audio.encode_mu_law(wav, mu=2 ** hp.bits)
        else:
            quant = audio.float_2_label(wav, bits=hp.bits)
    elif hp.voc_mode == 'MOL':
        quant = audio.float_2_label(wav, bits=16)

    return mel.astype(np.float32), quant.astype(np.int64)

def __len__(self):
    return len(self.samples_fpaths)
```

```
def collate_vocoder(batch):
    mel_win = hp.voc_seq_len // hp.hop_length + 2 * hp.voc_pad
    max_offsets = [x[0].shape[-1] -2 - (mel_win + 2 * hp.voc_pad) for x in batch]
    mel_offsets = [np.random.randint(0, offset) for offset in max_offsets]
    sig_offsets = [(offset + hp.voc_pad) * hp.hop_length for offset in mel_offsets]

    mels = [x[0][:, mel_offsets[i]:mel_offsets[i] + mel_win] for i, x in enumerate(batch)]

    labels = [x[1][sig_offsets[i]:sig_offsets[i] + hp.voc_seq_len + 1] for i, x in enumerate(batch)]

    mels = np.stack(mels).astype(np.float32)
    labels = np.stack(labels).astype(np.int64)

    mels = torch.tensor(mels)
    labels = torch.tensor(labels).long()

    x = labels[:, :hp.voc_seq_len]
    y = labels[:, 1:]

    bits = 16 if hp.voc_mode == 'MOL' else hp.bits

    x = audio.label_2_float(x.float(), bits)

    if hp.voc_mode == 'MOL' :
        y = audio.label_2_float(y.float(), bits)

    return x, y, mels
```

APPENDIX B

SCREENSHOTS

Toolbox

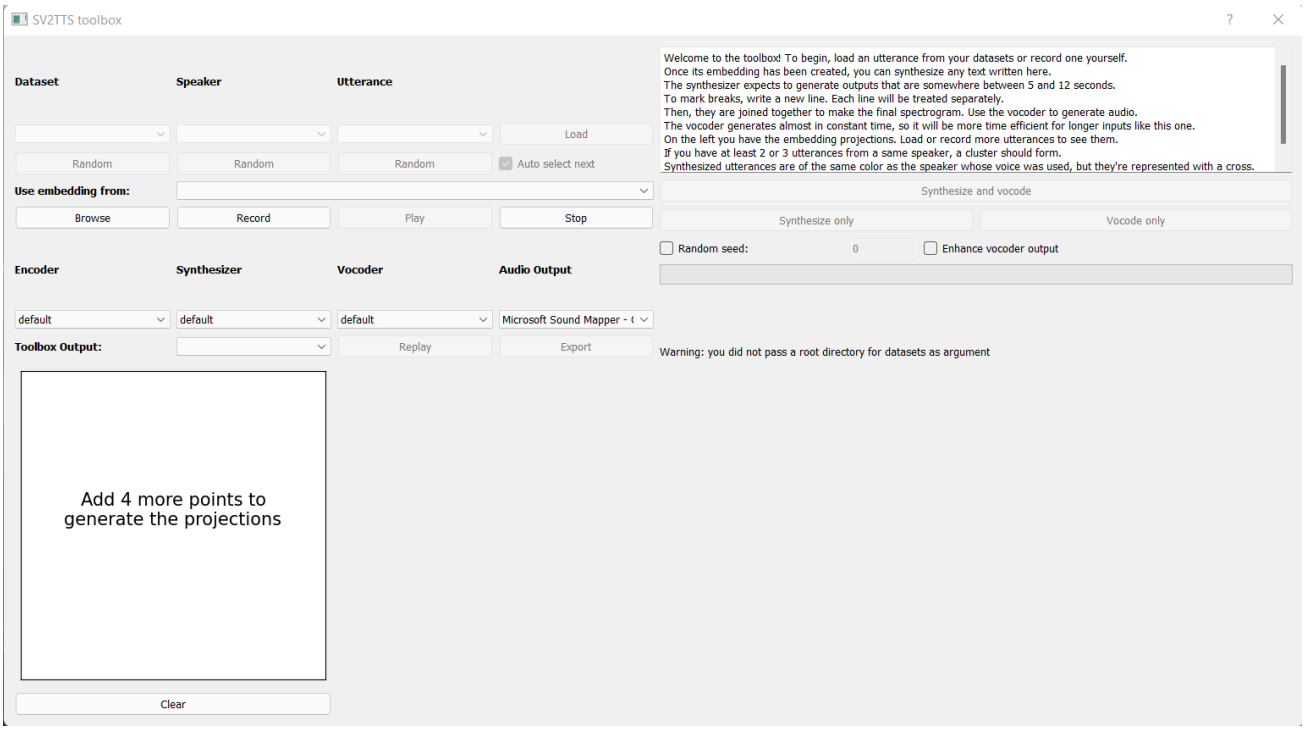

Figure B.1 Tool box

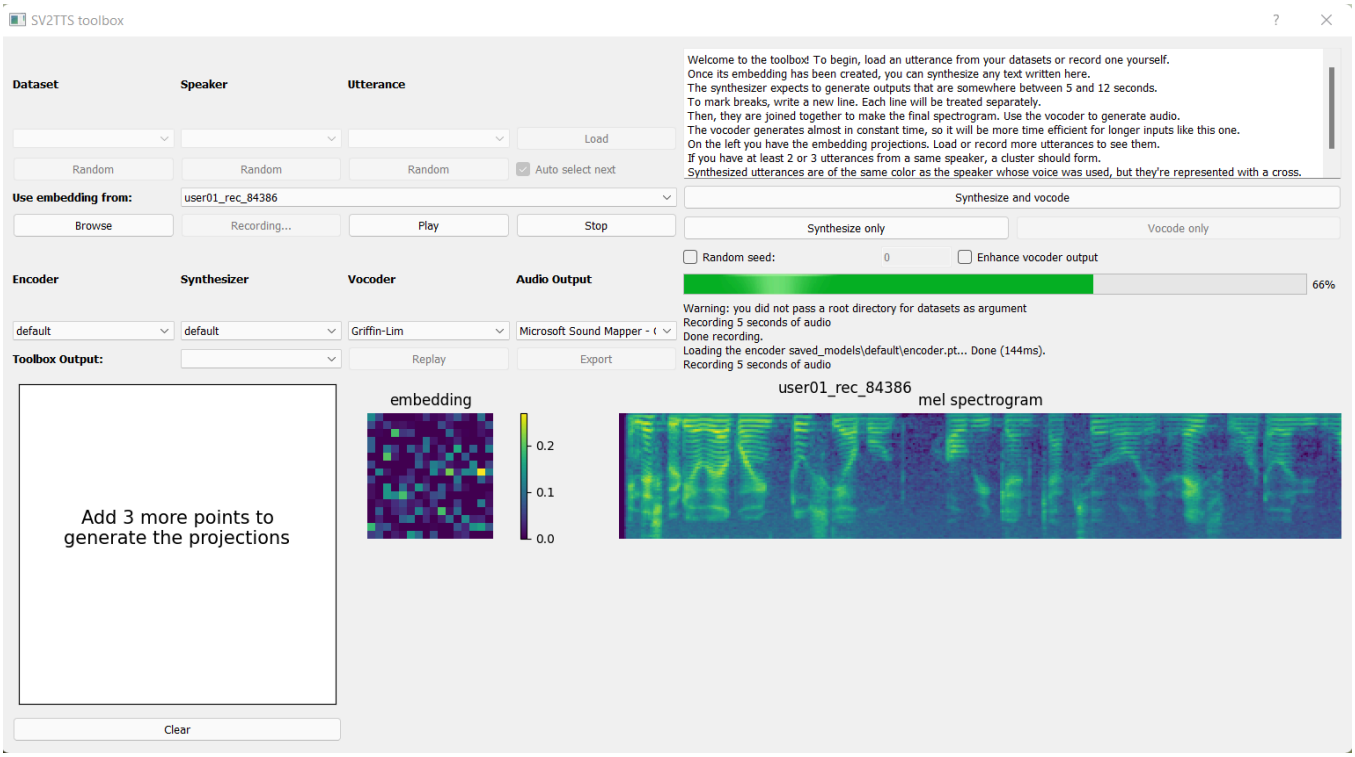

Figure B.2 tool box while processing

Mel Spectrogram

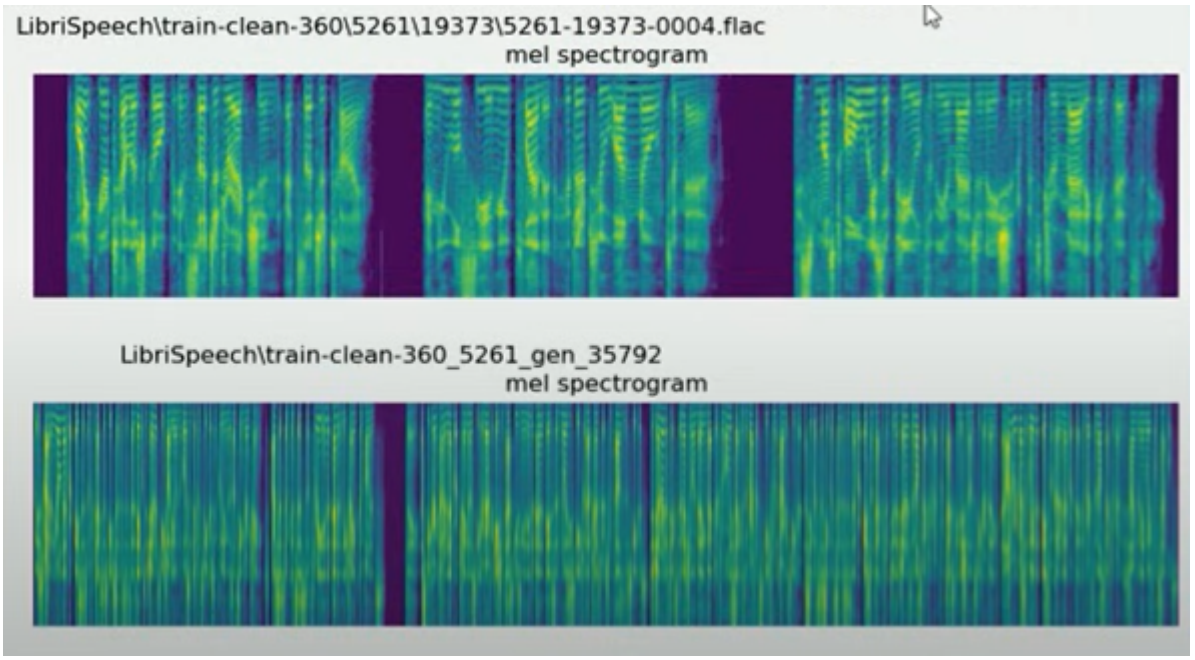

Figure B.3 Mel Spectrogram

BASE PAPER