BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION Informatics in English

DIPLOMA THESIS

Applications of Deep Learning in Voice Conversion

Supervisor
Dr. Molnar Arthur-Joszef, University Lecturer

Author
Bretan Cezar-Alexandru

2023

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA Informatică Engleză

LUCRARE DE LICENŢĂ

Aplicaţii ale Instruirii Profunde în Conversia Vocală

Conducător ştiinţific
Lect. Univ. Molnar Arthur-Joszef

Absolvent
Bretan Cezar-Alexandru

2023

ABSTRACT

With this paper, we first aim to make a literature review on the domain of Voice Conversion, the technique through which characteristics of a speech signal like speaker identity are manipulated, to undergo experiments in stride to improve relevant qualitative aspects of the existing approaches and to provide a practical application that leverages our yielded experimental results.

First, we stated the importance of Voice Conversion, and made an overview of the approaches that have been proposed in the domain around the speech signal analysis theory that is relevant in this space, and highlighted Deep Learning's impact on the literature's progress in terms of applicability and performance.

Then, we made a series of experiments that employ one of the state-of-the-art approaches as a starting point, and other components that originate from outside the literature. A significant dataset expansion has allowed us to bring a practical angle to our final proposed conversion pipeline, that we've integrated into a REST API.

Finally, we provided and documented a native Android client for communicating with the conversion API, to form an application meant for Voice Conversion of speeches from one source speaker to one of three target speakers, and other conversion-related tasks.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

Contents

1 Introduction **1**

2 Voice Conversion - Utility & State of the Art **4**
2.1 Feature Extraction from Speech Waveforms 5
2.1.1 Parametric vocoder utilities 5
2.1.2 Automatic Speech Recognition (ASR)-based features 6
2.1.3 Log-Mel-Spectrograms 7
2.2 Approaches employed in achieving VC 8
2.2.1 Leveraging parametric vocoder-based features 8
2.2.2 Combining ASR-based and vocoder-based features 11
2.2.3 Leveraging Log-Mel-Spectrograms 12
2.2.4 Comparison of Frameworks 13

3 Proposed Approach **18**
3.1 Analysis of Considered Approach 18
3.2 Experiments 20
3.3 Objective Results 25
3.4 Integrated Pipeline & Implementation 26

4 Application **28**
4.1 Analysis & Design 28
4.1.1 Use Cases & Functionalities 28
4.1.2 High-level Structure 29
4.1.3 Flow of Functionalities 31
4.2 Implementation 35
4.2.1 Technologies used 35
4.2.2 Concrete Implementation Aspects 37
4.3 Usage 42

5 Conclusions **45**

Bibliography **46**

Chapter 1

Introduction

Voice conversion (VC) is the technique through which certain characteristics of a speech signal are being modified, without changing the linguistic content. Speaker identity modification (illustrated in Figure 1.1) is the most common aspect around which the domain's literature has created discussion, though manipulation of speech prosody, accent and emotional expression have also been researched.

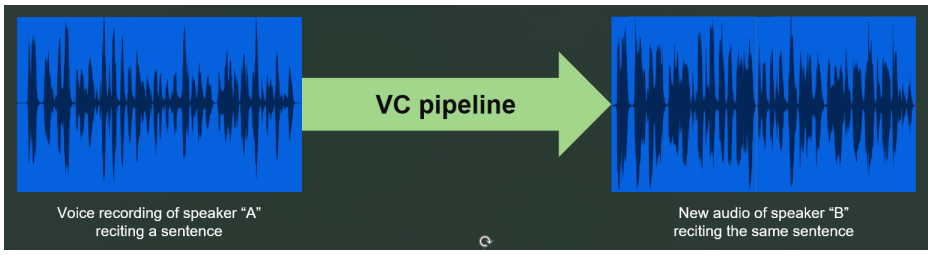

Figure 1.1: Simple illustration of speaker identity conversion.

In the development of new state-of-the-art approaches to VC, since it belongs to the more general technical field of speech synthesis, alongside Text-to-Speech (TTS), the main goal is to improve two aspects of the resulting speech content: naturalness and/or similarity to the target speech. The reduction of required training data and the improvement of conversion speed can be also considered as secondary goals.

Our personal motivation for researching this topic stems from general interest regarding audio processing, the Deep Learning technology and the practical implications of Voice Conversion (that we'll detail in Chapter 2).

According to the chart below (Figure 1.2), there is a continually increasing research interest in the domain of Voice Conversion.

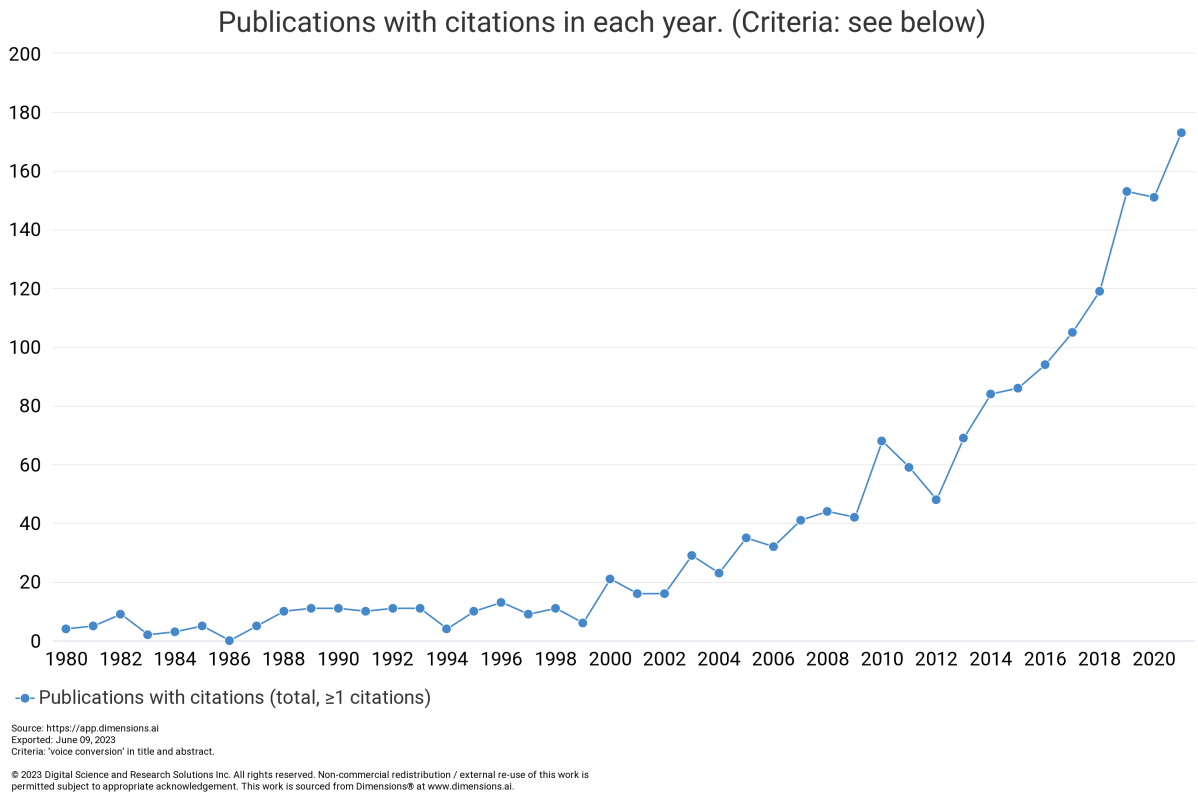

Figure 1.2: The number of scientific works regarding Voice Conversion published each year, from 1980 to 2021, as charted by dimensions.ai [HPH18]

While there are related scientific works dating back to the 1980s, we believe there are three main pivotal points (whose importance will be discussed in Chapter 2) in the domain's research timeline that elevated the problem's practicability:

- 1999 - Introduction of the STRAIGHT speech analysis tool [KMKd99];
- early 2010s - Advent of Deep Learning techniques;
- late 2010s - Apparition of Generative networks and the implementation of frequently used modules within modern Deep Learning toolkits (TensorFlow, PyTorch).

Our paper is organized as follows. In Chapter 2, we make an overview of the technology's utility and state-of-the-art, structured mainly on top of the theory regarding speech waveform analysis that is relevant to VC. In Chapter 3, we present details on the experiments we've done by using one of the state-of-the-art approaches as a starting point, the obtained results and a pipeline that will be embedded into a REST API. In Chapter 4, we describe, from a Software Engineering perspective, an Android client application that was designed to communicate with the API we proposed. Finally, Chapter 5 summarizes the paper.

Overall, this paper presents the following original contributions:

- **An overview of the state-of-the-art in Voice Conversion, from the perspectives of speech waveform analysis and Deep Learning's rapid evolution;**
- **An attempt at enabling conversion from an open set of target speakers, based on some previous work from the original authors of the approach we chose [KTK21] as a starting point for experimentation;**
- **An expansion of the CMU Arctic dataset with a large amount of data;**
- **Training of three speaker-dependent HiFiGAN vocoders for waveform synthesis, as alternatives to ParallelWaveGAN;**
- **Training and evaluation of a StyleGAN-based Log-Mel-Spectrogram enhancer for restoring information within converted features;**
- **A Django REST API that implements a full VC pipeline meant for conversion of one source speaker to three target speakers;**
- **A native Android client application built using a modern toolkit, meant for basic Voice Conversion and VC-related tasks such as:**
  - Speech Recording;
  - File storage, management and sharing;
  - Audio playback;
  - Voice conversion to a fixed number of speakers.

Chapter 2

Voice Conversion - Utility & State of the Art

The VC technology has its usages in areas such as: personalized speech synthesis, speaker-identity modification, emotional expression alteration, accent changing [FBGO09], normalization of impaired speech [THA+17], dubbing video content, and other forms of entertainment.

Currently, the Text-To-Speech (TTS)-based applications that are being used in some of these areas are yielding speeches that are impressively similar to the intended target, but still not natural enough to be indistinguishable. Future TTS technologies like Microsoft's VALL-E promise a significant improvement to speech naturalness, though still, TTS-based voice cloning can have limited to no control over the finer characteristics of the desired speech such as word cadence, emotion, and loudness.

TTS technology may be ahead of VC in both the speaker similarity and speech naturalness aspects, but we don't have reasons to believe that strides cannot be made such that the latter will yield results that are on par with TTS. If the two technologies will be brought into a position of competition sometime in the future, the finer speech characteristics such as the ones mentioned above will become the key comparison points between them.

If we were to consider a concrete demanding task of generating a speech resembling target speaker "X", that should start with a fast word cadence and emotion-neutral, but from a certain point on, the speaker should slow down, sound more confused or quieter, one could expand a TTS-based tool to take additional speech characteristic settings that are entered manually for certain segments of the text prompt given by the end user.

Or, alternatively, one could adopt a VC-based tool that takes a source speech of any person or voice actor that recites the sentences that "X" would say, and this source speech can contain some target prosodic elements that can be reproduced by the source speaker, such as the rhythm and the loudness. Therefore, the remaining job of the VC tool would be to convert the other prosodic elements that may not be possible to be reproduced by the source, namely the timbre, the pitch and the accent.

In our opinion, the second alternative seems more flexible, less convoluted, and may be more convenient for certain end users. Additionally, as we're about to see, some technologies and aspects that originate in the TTS field of research have been re-used in the VC field, yielding remarkable results.

2.1 Feature Extraction from Speech Waveforms

In any VC pipeline there must be a phase where features that may provide relevant information regarding the characteristics of a human voice are being extracted from the training datasets, consisting of raw PCM waveform vectors. These datasets may be parallel (each formed of multiple speakers saying the same phrases), or non-parallel (each formed of multiple speakers saying various phrases).

2.1.1 Parametric vocoder utilities

One of the first tools that was used in this field for this purpose was the STRAIGHT (Speech Transformation and Representation using Adaptive Interpolation of weiGHTed spectrum) vocoder [KMKd99], first introduced in 1999, that uses mathematically-based algorithms to decompose a speech signal into three key feature vectors (as visualized in Figure 2.1):

- F0 (Fundamental frequency) - a 1D vector of numbers representing an approximated pitch, measured in Hz, that can be thought of as the pulses the human vocal cords emit.
- MSP (Magnitude Spectra-based features) - a sequence of 1D vectors representing log-power-frequency spectral envelopes for each frame of the speech sample, which can be thought of as how the human vocal tract responds to the pulses. In some approaches, these are then pre-processed to Mel-cepstrums by applying a natural log scale, an Inverse Discrete Fourier Transform, and a Frequency Transform to a Mel-cepstrum of a certain order (typically 24th or 27th order), and referred to as MCEPs (or MCCs).

- AP (Aperiodicity) - a sequence of 1D vectors representing the time-frequency periodic component of the noise that causes interference with the spectral features.

The speech signal is divided into several frames, usually 5ms long, and one value from each of the three feature vectors corresponds to each frame. Additionally, such a tool is also capable of re-composing these back into the original PCM waveform with minimal artifacts, though the quality of the resulting synthesized speech may vary in scenarios where the features have been manipulated.

In some approaches, an additional 1D vector of binary values representing whether a speech frame is voiced or unvoiced is used as a feature, often referred to as U/V.

In 2016, the WORLD vocoder [MYO16], an alternative to STRAIGHT that claimed to have on-par feature fidelity, for less computational cost, was proposed.

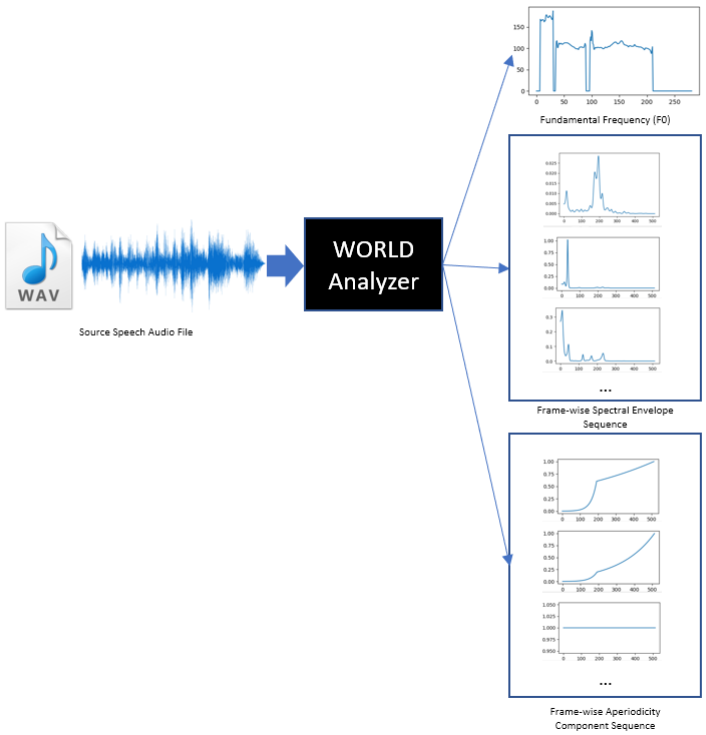

Figure 2.1: Visualization of the WORLD analyzer.

2.1.2 Automatic Speech Recognition (ASR)-based features

Phonetic Posteriorgrams (PPGs) are a phonetically-aware intermediary representation of an input speech waveform.

We consider the Short-Time Fourier Transform of a waveform, with a typical window size of 25ms, and a typical hop size of 5ms. This representation's idea is to have a 2D matrix, one axis representing time, the other axis representing the index of a phonetic class, and the values within the matrix must represent the posterior probability (a float value between 0 and 1) of each speech window belonging to each of the phonetic classes. These phonetic classes may represent linguistic senones or phonemes (as visualized in Figure 2.2).

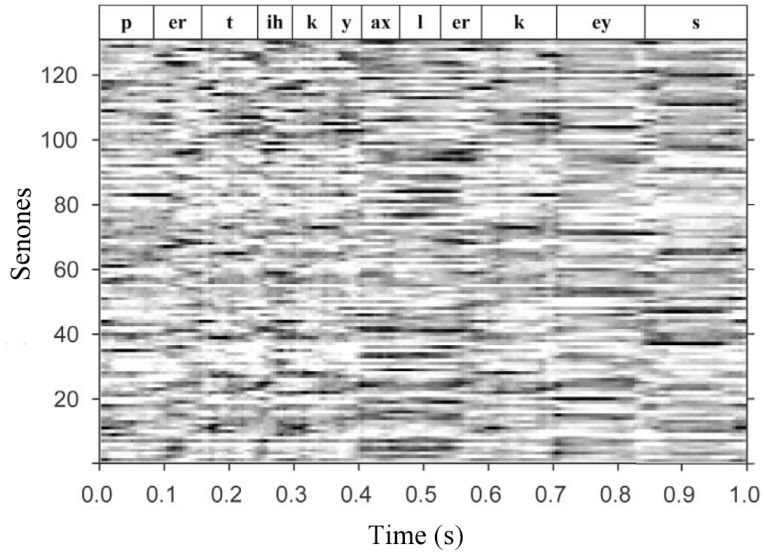

Figure 2.2: Figure excerpt from [SLW+16]. "PPG representation of the spoken phrase "particular case". The number of senones is 131. Darker shade implies a higher posterior probability."

Such a representation is obtained by feeding the Mel-Frequency Cepstral Coefficients (MFCCs) of a waveform, a widely used audio feature in the domain of Speech Recognition, through a DNN-based ASR model. In the domain's literature, the training for such a model is often done using the Kaldi ASR toolkit [PGB+11].

2.1.3 Log-Mel-Spectrograms

More recently, literature has begun working with a feature that is much easier to obtain from a speech waveform than the two aforementioned ones. Log-Mel-Spectrograms can be seen as "images" of an audio waveform that represent all of the component frequencies, in way that is relevant to human perception (as visualized in Figure 2.3). Through that perspective, such a feature may prove effective in a Fully Convolutional Neural Network (FCNN)-based pipeline.

These are obtained from a raw PCM waveform by applying the following in order: Short-Time Fourier Transform (typical parameters: window size = 64ms, hop size = 16ms), Mel Scaling (typically to 80 Mel bands) and a (natural or decimal) log scale.

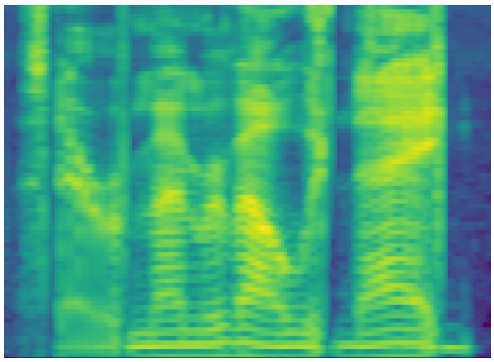

Figure 2.3: Visualization of a Log-Mel-Spectrogram.

This type of spectrogram is used in TTS pipelines such as the ones derived from NVIDIA's NeMo toolkit [HMK+], which consist of a Mel-spectrogram generator (i.e. Tacotron2), that takes a text prompt as an input, and a GAN-based neural vocoder (i.e. MelGAN [KKdB+19], HiFiGAN [KKB20], ParallelWaveGAN [YSK19]), that takes Mel-spectrograms as input, and outputs a PCM waveform.

2.2 Approaches employed in achieving VC

2.2.1 Leveraging parametric vocoder-based features

The first Voice Conversion methods that produced promising results once applied were actually based on statistical modeling rather than Deep Learning. These used the STRAIGHT analyzer for feature extraction, and its matching synthesizer for reconstructing the speech.

Toda et al. [TBT07] proposed the use of Gaussian Mixture Models (GMMs) to model the MCEP trajectories and variances using a maximum likelihood estimation technique.

Helander et al. [HSVG12] presented a Dynamic Partial Least Squares (DKPLS)-based regression technique, where a kernel function is being modeled to minimize the sum of squared errors between the feature matrix extracted from the target

speech, and the one resulted from applying the kernel function to the feature matrix extracted from the source speech. This feature matrix is a concatenation of the MCEP, AP and U/V features from three consecutive speech frames.

Wu et al. [WVK+13] introduced two statistical techniques based on Spectrogram Factorization and Deconvolution, respectively. Their main idea is to construct two dictionaries out of multiple-frame MSP exemplars from the available parallel source and target speeches. Then, an activation matrix is statistically modeled such that the source dictionary can be factorized or convolved with it in an effort to match the target dictionary. A Mel-scale filter is applied to the source spectrums before constructing the source dictionary to reduce computational cost.

The advancement of Deep Learning-based techniques has swiftly lead to a paradigm shift in Voice Conversion research, by altering the way problems are being formulated, and opened the possibility of leveraging non-parallel data. The following approaches use the WORLD analyzer for feature extraction.

Sisman et al. [SZS+18] proposed the first usage of the WaveNet vocoder as a replacement for the traditional parametric vocoder-synthesizer utilities. It is a fully convolutional neural network, introduced by DeepMind in 2016 [vdODZ+16], purposed for generating raw audio waveforms, that is able to synthesize speech that sounds noticeably more natural than their parametric counterparts. This framework specifies three training phases.

Firstly, the three features extracted from a parallel dataset containing utterances from the source and target speaker are aligned using a Dynamic Time Warping (DTW) algorithm. The re-aligned MCEPs are used to train 2 DNNs simultaneously that make up a Generative Adversarial Network (GAN) architecture: a Discriminator, to distinguish natural and generated samples, and a Generator, trained to deceive the discriminator. The Discriminator predicts a special loss that is propagated back to the Generator, named adversarial loss.

Secondly, a WaveNet vocoder is trained on a non-parallel dataset, so it will act like a speaker-neutral synthesizer, and after finishing, its state will be "frozen".

Lastly, using the GAN trained during the first stage, the parallel source features will be converted, and these will be used for further training the WaveNet vocoder against the original target speech data after "unfreezing" it. This can be thought of as an adaptation process.

Kameoka et al. [KTK+18] introduced a FCNN-based approach, named Conv-S2S-VC, that brings a series of trends from the domain of Deep Learning that mate-

rialized due to the advent of the Transformer architectures.

The frame-wise feature vector is obtained by stacking the MCEPs, F0, 1-dimensional coded AP and U/V binary. At both training and inference time, a number of r non-overlapping frames are being predicted at once, a trick originally employed by the Tacotron vocoder [WSRS+17] and an addition of the positional encodings [VSP+17] is performed.

On the topic of Transformer-specific aspects, the overall architecture is of the Encoder-Attention-Decoder type, Scaled Dot-Product Attention is used for computing time-warped versions of the feature sequences, and learnable class embeddings are used for enabling Many-to-Many or Any-to-Many Voice Conversion, more on these capabilities at Section 2.2.4.

On the architecture level, the decoder is meant to make predictions based on the information given by the target speech, while the additional reconstructor module placed after the decoder is meant for ensuring the converted feature sequence has enough information from the target speech. As a synthesizer utility, the authors recommend using a neural vocoder compatible with the four stacked features (like WaveNet).

The training process for the feature converter is single-staged, since there is a single loss that is being propagated across all modules.

Acknowledging that S2S models have the shortcoming of requiring large parallel datasets for training, the same authors have introduced a series of CycleGAN-based [ZPIE17] VC methods.

CycleGAN was originally conceived with computer vision tasks like image-to-image characteristics translation in mind, with its two main distinctions from an ordinary GAN is the employment of an additional Discriminator for predicting the inverse mapping adversarial loss, and the forward and inverse losses are used for computing a cycle-consistent loss that is given to the Generator as feedback.

Within the first two revisions of CycleGAN-VC [KK17] [KKTH19], Kaneko et al. have focused on developing a MCEP-to-MCEP mapper that does not require source-target parallel data. The motivation lies in the fact that in some real world scenarios, obtaining a significant amount of parallel data may prove difficult and impractical.

The first revision [KK17] added Gated Linear Units as activation functions to the Generator and Discriminator, and the second revision [KKTH19] introduced two additional discriminators and some architectural enhancements.

2.2.2 Combining ASR-based and vocoder-based features

It is believed that ASR-based features provide a speaker-neutral representation of the linguistic content of a speech waveform, which, at first glance, raises the possibility of a to-MCEP mapping mechanism to prove effective. The following mentioned approaches train their ASR models on a large set of non-parallel data.

Sun et al. [SLW+16] introduced the usage of PPGs in VC for the first time in a many-to-one conversion capable approach, more on this capability at Section 2.2.4.

In the training phase, it first trains a speaker-independent ASR model on a standard ASR data corpus, that maps MFCCs to PPGs, then it trains a Deep Bidirectional Long Short-Term Memory (DBLSTM) network for PPG-to-MCEP mapping on a set of speeches from the target speaker X.

In the conversion phase, speeches from the N source speakers are analyzed by STRAIGHT to obtain the F0 and AP features, and their MFCCs are fed through the ASR model, the yielded PPGs are fed through the DBLSTM model, and the converted MCEPs along with the F0 and AP features are given to the STRAIGHT synthesizer in order to be reconstructed into a waveform.

Zhang et al. [ZSR+18] presented a framework that makes use of a (pre-)trained ASR model that takes the MFCC features of a speech waveform as an input, and outputs a sequence of One-hot phoneme label (1HPL) vectors, that indicates what phoneme each speech frame is most likely a part of.

The training phase consists of three stages: average model training, average model adaptation, and Error Reduction Network training. In the first phase, a large non-parallel dataset is used to train a network for 1HPL-to-MCEP mapping, that acts as an average model. After training is finished, this model is ”frozen”.

In the second phase, this average model is being ”unfrozen”, and then training is continued on a smaller dataset that contains speeches from the target speaker.

Finally, in the third phase, a parallel dataset consisting of phrases from the source speaker and the target speaker is considered, and this will be used for training a different DBLSTM neural network, that the authors call an Error Reduction Network (ERN). As the name may suggest, its purpose is to bring the MCEPs that were outputted by the adapted average model closer to the ground truth target MCEPs.

Since two parallel speeches may differ in duration, the two MCEP vectors may differ in length. To rectify this issue, a frame-wise alignment of two vectors obtained from parallel data will be performed using a Dynamic Time Warping (DTW) algorithm. Using the frame-wise alignment information between the two MCEP vectors, a modification of the One-hot phoneme label vectors obtained from the source

speech is performed so that the labels are paired correctly with the re-aligned source MCEPs. These are fed through the adapted 1HPL-to-MCEP adapted model, and the outputs, along with the target MCEPs, to train the ERN.

The conversion pipeline is similar to the one from the previous approach, except for the ERN, that is inserted right after the 1HPL-to-MCEP model, and before the STRAIGHT synthesizer.

Later on, the same authors have expanded their work to make use of PPGs instead of One-hot phoneme labels in a new framework proposal, DeepConversion [ZSZL20]. Like [SLW+16], it uses an PPG-to-MCEP model, but here, after training it on a larger ASR corpus, it is further adapted on a dataset of limited parallel data. The training methodology for the ERN remains the same. As an optional addition to the pipeline, it proposes the use of an adapted WaveNet vocoder, similarly to the way it is trained and leveraged by [SZS+18].

2.2.3 Leveraging Log-Mel-Spectrograms

Contrastive Predictive Coding (CPC) [OLV18] is a self-supervised representation learning technique used that proved strong results in the domains of speech, text, images and reinforcement learning. Its main characteristics are the use of a Noise-Contrastive Estimation (NCE)-based [GH10] contrastive loss, and an autoregressive model for predicting future segments within the representation at run-time.

Van Niekerk et al. [vNNK20] have made use of CPC for Voice Conversion for extracting phone-like representations from Log-Mel-Spectrograms, thus offering an alternative to ASR-based features. They have chosen a convolutional model for the encoder, that extracts the downsampled continuous features, that are then discretized, using a trainable codebook, via vector quantization. For reconstructing the waveforms, the authors proposed the training of a separate RNN-based vocoder. The VC capabilities aren't really being described in detail in this paper, but it seems like the encoder and the quantization are tasked to extract a speaker-neutral representation of any speech, and the decoder is tasked to handle the conversion and the reconstruction simultaneously; though whether a single decoder is capable of producing speeches across multiple target speakers remains unclear.

The employment of a mechanism that performs a source-to-target Mel-spectrogram mapping, along with a GAN-based neural vocoder seems, to us at least, like the most viable concept to approaching VC.

Following the advancement GAN-based neural vocoders that leverage the potential of Mel-Spectrograms in speech synthesis, Kameoka et al. have introduced a refresh [KTK21] to their series of frameworks [TKKH18] [KTK+18] [KHT+20], that make use of RNN, FCNN and Transformers respectively, but all of them incorporate an Attention predictor mechanism. The refresh brings the Log-Mel-Spectrograms as input and output features, which will be passed to a ParallelWaveGAN [YSK19] vocoder for waveform synthesis, and slight modifications to the inner architectures of the RNN and FCNN-based versions so that all three variants have the same general structure: PreNet-Encoder-Attention-PostDecoder-PostNet. Out of the three, the CNN-based variant seemed to perform the best.

Additionally, they introduce two less complex model variants for Conv-S2S-VC2 and Transformer-VC2 that are more optimal for Voice Conversion via audio streaming due to the reduced inference time, which leads to less latency.

Another refresh linked to the progress in speech synthesis brought by the GAN-based neural vocoders are the next two revisions of CycleGAN-VC [KKTH20] [KKTH21], where Kaneko et al. have made observations regarding the lossy nature of MCEP-to-MCEP conversion as seen in the harmonic structure of the resulting speeches, and have instead focused on the development of a source-to-target Log-Mel-Spectrogram mapper that still doesn't require parallel data.

The third revision [KKTH20] introducing a new CNN block named Time-Frequency Adaptive Normalization (TFAN) purposed to help preserve the time-frequency structure that tends to be discarded by the previous two revisions [KK17] [KKTH19], and the "Mask" revision [KKTH21] incorporating a Mel-Spectrogram manipulation step called Filling In Frames (FIF), where a random sequence of frames are masked via an element-wise product with an all-ones mask, and leveraging an inverse converter to fill in the masked frames.

2.2.4 Comparison of Frameworks

From a theoretical point of view, the main aspects that differentiate the frameworks given by the domain's literature may be the following:

- The technology behind the mapping model;
- The features and representations used in the training process;
- The runtime pipeline architecture;
- The synthesizer used for waveform generation from the converted features.

In Figure 2.4, we've compiled a table with the approaches discussed in 2.2.1, 2.2.2 and 2.2.3, compared by these four aspects and ordered chronologically to better observe their evolution.

Approach Title	AI super set	Mapping model	Used features	Architecture (Runtime pipeline)	Signal synthesizer
GMM [TBT07] (2007)	ML	Gaussian Mixture Model	{F0, AP}, MCEP	Analyzer - Mapper - Synthesizer	STRAIGHT synth (parametric vocoder)
DKPLS [HSVG12] (2012)	ML	Dynamic Kernel Partial Least Squares Regression	MCEP, AP, U/V	Analyzer - Mapper - Synthesizer	STRAIGHT synth (parametric vocoder)
Spectrogram Factorization [WVK+13] (2013)	ML	Non-negative matrix factorization	{F0, AP}, MSP (raw spectrums), MMSP, MCEP just for alignment of source-target	Analyzer - Source dictionary - Activation Matrix - Target dictionary - Synthesizer	STRAIGHT synth (parametric vocoder)
PPG for M2O VC [SLW+16] (2016)	DL	DBLSTM	MFCC for ASR training & inference, PPG + MCEP for DBLSTM training & inference, {F0, AP}	Analyzer - ASR - Mapper - Synthesizer	STRAIGHT synth (parametric vocoder)
Error Reduction Network [ZSR+18] (2018)	DL	DBLSTM	MFCC for ASR training & inference, One-hot phoneme labels + MCEP for DBLSTM training & inference, {F0, AP}	Analyzer - ASR - Mapper - Synthesizer	STRAIGHT synth (parametric vocoder)
GAN + WaveNet [SZS+1] (2018)	DL	GAN	{F0, AP}, MCEP	Analyzer - Mapper - Synthesizer	WaveNet synth (neural vocoder)
Conv-S2S-VC [KTK+18] (2018)	DL	FC-CNN + Attention	Stacked: MCEP, F0, coded AP, U/V	Analyzer - Encoder - Attention - Decoder - Reconstructor - Synthesizer	Any, WaveNet synth recommended (neural vocoder)
CycleGAN-VC2 [KKTH19] (2019)	DL	CycleGAN	{F0, AP}, MCEP	Analyzer - Mapper (Downsample - ResBlocks - Upsample) - Synthesizer	WORLD synth (parametric vocoder)
DeepConversion [ZSZL20] (2020)	DL	DBLSTM	MFCC for ASR training & inference, PPG + MCEP for DBLSTM training & inference, {F0, AP, (U/V if WaveNet)}	Analyzer - ASR - Mapper - Synthesizer	STRAIGHT synth (parametric) or WaveNet (neural)
VQ-CPC [vNNK20] (2020)	DL	FC-CNN + GRU-RNN	Log-Mel-Spectrograms	Encoder - Bottleneck + Quantizer - Decoder	RNN-based synth (neural vocoder)
CycleGAN-VC3 [KKTH20] (2020)	DL	CycleGAN	Log-Mel-Spectrograms	Mapper (Downsample - ResBlocks - Upsample) - Synthesizer	MelGAN (neural vocoder)
Conv-S2S-VC2 [KTK21] (2021)	DL	FC-CNN + Attention	Log-Mel-Spectrograms	Encoder - Attention - Decoder - Synthesizer	Parallel WaveGAN (neural vocoder)
MaskCycleGAN-VC [KKTH21] (2021)	DL	CycleGAN	Log-Mel-Spectrograms	Mapper (Downsample - ResBlocks - Upsample) - Synthesizer	MelGAN (neural vocoder)

Figure 2.4: Table comparing the theoretical aspects of the literature's frameworks.

As we alluded to earlier, neural vocoders have seen an increased adoption in the field since their appearance due to their yielded speeches sounding more natural to humans.

Moreover, vocoders of the GAN-based variety has determined a shift away from the usage of features that were originally introduced by traditional parametric speech analyzers, and towards the usage Log-Mel-Spectrograms, which don't require a specialized analyzer tool for extraction and are easier to work with.

From a practical perspective, any VC framework can be associated to a conversion class, based on the number of source speakers it is designed to take input data from, and the number of target speakers it is designed to convert the input speech to. The conversion classes mentioned in the literature are the following:

- One-to-One - When a conversion pipeline is designed to convert speech from a single speaker X, to speech sounding like a single speaker Y.
- Many-to-One - When given a set of N source speakers, and a target speaker X, the pipeline is designed to convert any speech from one of the speakers from the given set to target speaker X.
- Many-to-Many - When given a set of N speakers, the pipeline is designed for converting across any pair of speakers from the given set.
- Any-to-One or Any-to-Many - When given a set of $N >= 1$ speakers, the conversion pipeline is designed for converting speech from a speaker that may or may not be included in the given set, to one of the N speakers.

Another utilitarian aspect to consider regarding a VC framework is the amount of data required for training and whether it needs to be parallel or not. For example, obtaining a moderate or large amount of data that is parallel to the available target speaker data may be impractical for some use cases.

Finally, how well a framework performs given the required data for training is indicated by how natural the yielded speeches sound and how similar to the desired target speaker they are. Both of these are fairly subjective metrics, and most papers quantify these by carrying out a Mean Opinion Score (MOS) test, where a group of listeners is tasked with giving a score on a scale of 1 to 5 regarding the quality and/or the target similarity of the results. Some papers also conduct ABX preference tests ("does pipeline A or pipeline B provide a more natural/similar speech resembling target speaker X?"), to show how well their proposed approach performs against an already existing baseline approach.

One objective method for measuring the difference between a converted speech and the ground truth that was adopted by most research papers in the VC domain is Mel-Cepstral Distortion (MCD).

$$MCD[dB] = \frac{10}{\ln 10} \cdot \sqrt{2 \cdot \sum_{d=1}^{D} (m_t^d - m_o^d)^2}$$

where D represents the dimension of the MCEP, m_t^d denotes the d^{th} dimension of the target MCEP, and m_o^d denotes the d^{th} dimension of the output MCEP.

In Figure 2.5, we've compiled a table with the frameworks discussed in 2.2.1, 2.2.2 and 2.2.3, compared by what conversion class each is capable of, what and how much data each requires for training, and the objective and subjective measurements conducted by each proposing paper.

By observing the evolution of the output speech quality over time, it can be observed that the frameworks capable exclusively of one-to-one VC have grown to no longer require parallel data and some of the more recent ones that are capable of Any-to-Many have grown to perform on par with their less versatile counterparts.

However, when it comes to how well MCD can quantify speaker similarity, it is really hard to make out a correlation between its measured values and the subjective performance metrics across the listed frameworks, since not all papers conduct MOS-based similarity tests, and, counter-intuitively, it seems like the metric got worse over time. A possible cause for this could be that since MCD is computed based only on the MCEPs, it may be biased towards the frameworks that focus more on direct mapping of MSP-based features, like GMM [TBT07] and GAN + Adaptive WaveNet [SZS+18].

Approach Title	Framework capability	Required data amounts & types	Objective Perf. (MCD, lower is better)	Subjective Perf.
GMM [TBT07] (2007)	One-to-One**	Moderate amount of parallel data	4.6	MOS: 3.3 / 5 (Quality-wise)
DKPLS [HSVG12] (2012)	One-to-One**	Small to Moderate amount of parallel data	5.1	MOS: 3.5 / 5 (Quality-wise)
Spectrogram Factorization [WVK+13] (2013)	One-to-One**	Small amount of parallel data	5.5	85% quality-wise preference rate over GMM
PPG for M2O VC [SLW+16] (2016)	Any-to-One*	Large amount of non-parallel data + Moderate amount of data from target speaker	6.2	MOS: 3.8 / 5 (Quality-wise)
Error Reduction Network [ZSR+18] (2018)	Any-to-One**	Large amount of non-parallel data + Moderate amount of data from target speaker	6.19	MOS: 3.4 / 5 (Quality-wise)
GAN + WaveNet [SZS+1] (2018)	One-to-One**	Large amount of non-parallel data + Moderate amount of parallel data	4.1+	MOS: 4.05 / 5 (Quality-wise)
Conv-S2S-VC [KTK+18] (2018)	Many-to-Many + Any-to-Many	Large amount of parallel data	6.22 (M2M) 6.63 (A2M)	MOS: 3.7 / 5 (M2M, Quality-wise) MOS: 3.6 / 5 (M2M, Similarity-wise)
CycleGAN-VC2 [KKTH19] (2019)	One-to-One**	Moderate amount of non-parallel data from the source and target speakers	6.26-7.22	MOS: 2.2-3.1 / 5 (Quality-wise)
DeepConversion [ZSZL20] (2020)	Many-to-One**	Large amount of non-parallel data + Moderate amount of parallel data	4.11 (Intra-gender) 5.2 (Inter-gender)	MOS: 3.98 / 5 (Quality-wise)
VQ-CPC [vNNK20] (2020)	Any-to-One***	Large amount of non-parallel data + Moderate amount of data from target speaker	Not measured	MOS: 3.64 / 5 (Quality-wise) MOS: 3.8 / 5 (Similarity-wise)
CycleGAN-VC3 [KKTH20] (2020)	One-to-One**	Moderate amount of non-parallel data from the source and target speakers	6.91-7.97+	MOS: 3.4-3.8 / 5 (Quality-wise) MOS: 3.1-3.6 / 5 (Similarity-wise)
Conv-S2S-VC2 [KTK21] (2021)	Many-to-Many + Any-to-Many****	Large amount of parallel data	6.1+	MOS: 4.3 / 5 (Quality-wise) MOS: 4.4 / 5 (Similarity-wise)
MaskCycleGAN-VC [KKTH21] (2021)	One-to-One**	Moderate amount of non-parallel data from the source and target speakers	6.73-7.64+	~60% quality and similarity-wise preference rate over CycleGAN-VC3

Figure 2.5: Table comparing the practical aspects of the literature's frameworks.

* = The original paper states Many-to-One, but since the dataset of N source speakers isn't involved in the training phase, we suspect the authors actually meant to refer to their approach as now what is more known as an any-to-one capable framework, since there is no reason the proposed conversion pipeline wouldn't take data from any English speaker, instead from only a limited set.

** = Capability not explicitly stated, assumption based on the nature of the model training methodology and/or the data used for the experiments stated in the paper.

*** = Capability not explicitly stated, assumption based on the author's suggestion of conditioning the decoder on a specific speaker and the intended speaker-independent nature of discovered acoustic units.

**** = Any-to-Many capability stated only by preceding paper.

+ = MCD measured after neural re-synthesis

Chapter 3

Proposed Approach

In this next chapter, we will present a series of experiments we've conducted with one of the state-of-the-art approaches and other open-source Deep Learning-based tools. Given the yielded results, we'll then describe the pipeline that we'll integrate into an API designed to communicate with a client application such as the one we've provided and documented in Chapter 4.

3.1 Analysis of Considered Approach

We've chosen the approach introduced by Kameoka et al. in 2021 [KTK21] as a starting point for experimentation, for its claimed high performance in subjective metrics, and the existence of an official open-source implementation provided by the authors [Kama].

The model presented by the authors has the purpose of mapping pairs of source-target Log-Mel-Spectrograms. It is a CNN-based model with an Encoder-Decoder architecture, whose main subcomponents are the following:

- Linear Layers, which are in fact 1x1 Convolution Layers with kernel size 1x1 and stride 1;
- Convolution Layers with various dilation factors;
- Causal Convolution Layers with various dilation factors;
- Gated Linear Units (GLU);
- Dropouts of 0.1 chance;
- Embedding Layers (E);
- Scaled Dot-Product Attention;

- Residual Blocks.

We have illustrated, based on the provided code [Kama], the mapper architecture (3.1), and the structure of each module (3.2 and 3.3) in the figures below.

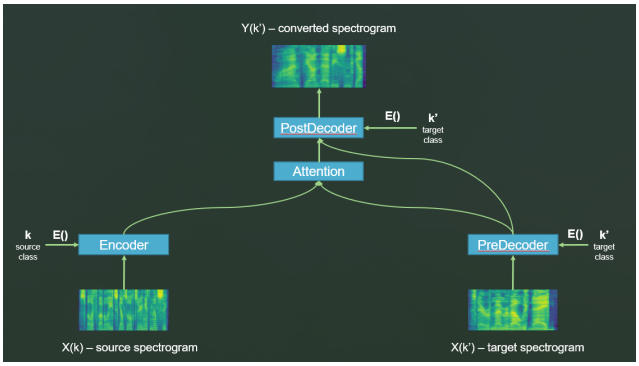

Figure 3.1: A simplified view of the overall architecture of ConvS2S-VC2 [KTK21].

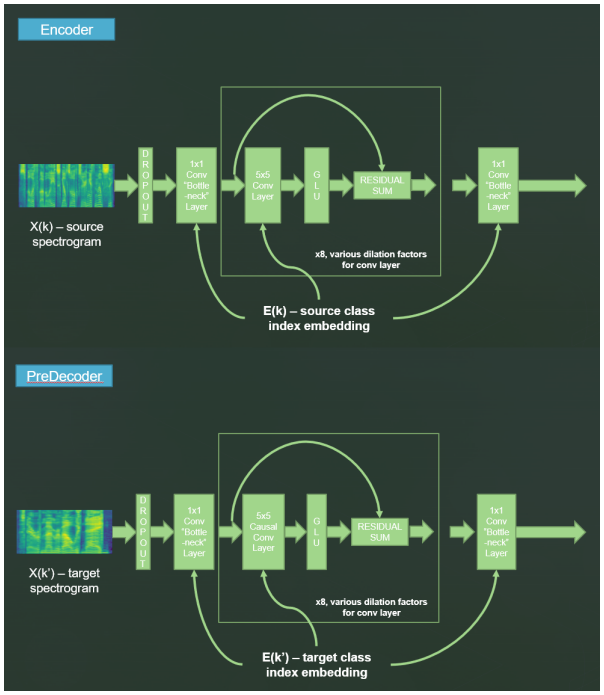

Figure 3.2: Schemes depicting the structure of ConvS2S-VC2's [KTK21] Encoder and PreDecoder modules.

The attention module consists of a Scaled Dot-Product Attention of the Encoder and PreDecoder outputs as the key and query vectors, respectively, and the value vector output sequence from the Encoder is multiplied by the Attention matrix in order to obtain a time-warped version of it [KTK+18].

The learning mechanism involves a composite loss, calculated as a weighted sum of an L1 loss, and a Diagonal Attention Loss meant to penalize non-linearity in the alignment of two source-target sequences [KTK+18].

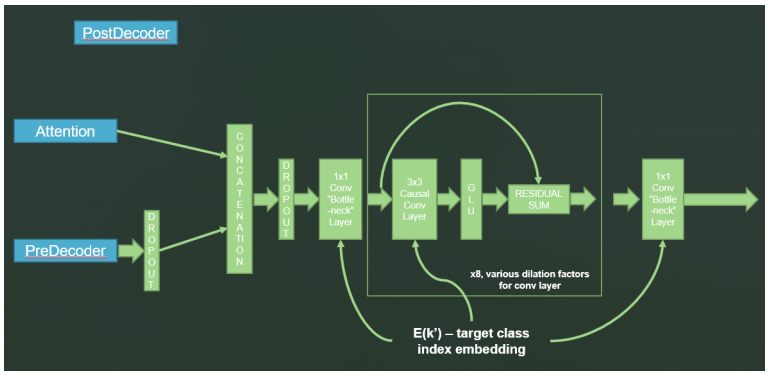

Figure 3.3: A scheme depicting the structure of ConvS2S-VC2's [KTK21] PostDecoder module.

Moreover, the inference process employs an autoregressive structure. Since, during conversion, the target spectrogram is not accessible, partial spectrograms that are being generated with each time step are being fed back to the PreDecoder, and used for predicting further features within the sequence [KTK21].

The authors have used an open-source implementation [Kay] of ParallelWaveGAN [YSK19] for synthesizing the Log-Mel-Spectrograms outputted by the mapper into waveforms.

3.2 Experiments

As a first experiment, we went through the necessary steps for setting up the framework for training a many-to-many mapper on a set of 6 speakers from the CMU Arctic dataset (aew, ahw, bdl, clb, eey, ljm). After training (200 epochs, 5e-5 learning rate, batch size 16), we used the mapper for converting Log-Mel-Spectrograms extracted from a subset of speeches we reserved for testing. We employed a ParallelWaveGAN model pre-trained on 5 speakers from the same dataset for synthesizing the waveforms from the converted spectrograms.

The obtained converted speeches were only moderately comprehensible, probably due to the fact that we used 593 utterances for training, as opposed to 1000. The similarity degree to the target speakers was mixed, since the vocoder was not trained on the exact same set of target speakers. This experiment served as a step to familiarize ourselves with the implementation.

We observed that the authors' preceding paper [KTK+18] has stated that Any-to-Many conversion can be enabled by modifying the Encoder module not to take the source speaker embedding as an input at training and test time.

We made what we deduced to be the required modifications to the provided

architecture for enabling this feature as a next experiment. We first trained a mapper on two disjoint subsets, one consisting of data from 7 source speakers, and the other from 6 target speakers. Another mapper was trained on the same source speakers, but only a single speaker as a target (aew). For waveform synthesis, we used the same pre-trained ParallelWaveGAN model.

Unfortunately, upon testing both mappers on test data from speakers inside and outside of the training set, the pipeline failed to produce any comprehensible speech waveforms. Granted, the F0, MCEP, AP, U/V feature was employed for that version, so it wasn't necessarily expected that the mapper would perform similarly on Log-Mel-Spectrograms from arbitrary speakers.

Following that, we've recorded the 593 phrases of the CMU Arctic dataset as said by a speaker we'll refer to as "czr". Using the newly recorded speeches, we've trained (600 epochs) a One-to-One czr-to-bdl mapper. Afterwards, we used a 2 minute and 50 second long speech uttered by speaker czr as test data.

Feeding the Log-Mel-Spectrogram extracted from the entire speech through the mapper didn't work, since the model was trained on utterances of at most 6 seconds in length. After dividing the speech into segments of at most 4 seconds, the resulting speech was great in the comprehensibility aspect, but not very similar to speaker bdl, possibly because the vocoder was not trained on this target speaker.

The next refinement we wanted to bring to the pipeline was to employ a different GAN-based neural vocoder. The first choice that came to mind was HiFiGAN [KKB20], since we were eager to experiment with modules from the NeMo Toolkit by NVIDIA, that are state-of-the-art in the ASR and TTS fields.

Moving forward, we trained 3 speaker-dependent HiFiGAN models from scratch on 1100 utterances from CMU Arctic speakers aew, bdl and rms, respectively. 32 speeches were left for validation steps. We used the training scripts offered as examples in the NeMo repository, modified the YAML configuration parameters (sampling rate, STFT parameters, segment length, normalization) and wrote the needed dataset manifest files.

We used the feature extractor provided by NeMo, and configured it to normalize to zero mean unit variance, since the mapper models output spectrograms within this range of values. The PyTorch Lightning trainer was configured to plot an L1 loss calculated between the source spectrogram and the spectrogram re-extracted from the synthesized audio (plot shown in figure 3.4). The vocoders took about 14 hours each on average to train on an NVIDIA RTX 3090.

By using the new vocoder trained on speaker bdl in the pipeline of our previous

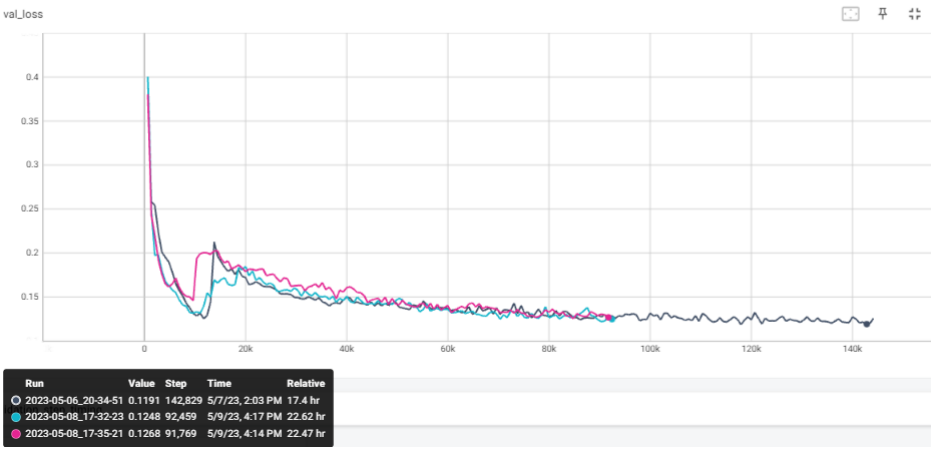

Figure 3.4: The losses computed at the validation steps for each speaker-dependent HiFiGAN training. Black: bdl ; Blue: aew ; Magenta: rms

experiment, the yielded conversion results were very high-pitched and with a lot of sound artifacts. This was because there were other differences between the features that were used for training the mapper and the vocoder, namely the base of the log filter (i.e. log10 vs. natural log), additional padding and the normalization strategy (the mapper used spectrograms normalized based on statistics computed across the entire dataset).

Consequently, in order for the two conversion pipeline models to work together correctly, both must be trained using the same feature extractor.

The feature extractor provided by NeMo was designed as a PyTorch module to inter-operate with specific data-loaders in mind, therefore we made some modifications so it can be used independently of the data-loader. Additionally, we added the option of normalization using pre-computed statistics to cover the possible contingency that the mapper would behave unexpectedly upon changing the normalization strategy.

Using the modified feature extractor set for per-spectrogram normalization, we re-trained (1000 epochs, 5e-5 learning rate, batch size 16) three One-to-One mappers that take speech data from speaker czr, and convert it to the speakers aew, bdl and rms, respectively. By using these in the pipeline, the yielded results sound very reminiscent of the target speakers with regard to the voice timbre, and are moderately comprehensible.

In Figure 3.5, we have a visual comparison of the spectrograms representing the source, the ground truth and the conversion output of the current pipeline. It is apparent that the converted spectrogram is more ”blurry” overall, which often poses as a difficulty for the HiFiGAN model at the vocoding end.

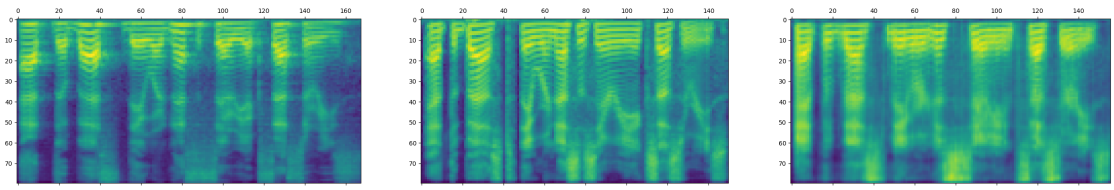

Figure 3.5: Three Log-Mel-Spectrograms of the sentence "At the best, they were necessary accessories.".

Left: Extracted from original speech from speaker czr

Middle: Extracted from original speech from speaker bdl

Right: Output of the czr-to-bdl mapper model trained on 593 source-target utterance pairs when feeding the spectrogram on the left through it.

This issue also appears in the TTS field, with spectrogram generators like FastPitch [Kor]. As an effort to remedy this, the NeMo toolkit also provides a Spectrogram Enhancer model that is based on StyleGANv2 [KLA+20], a generative model for realistic images, now repurposed for restoring details to "blurry" spectrograms [Kor].

Since a similar effect can be observed with our obtained outputs, we felt that employing this GAN-based Enhancer in this Voice Conversion pipeline may lead to a slight improvement to the results.

For training this model, we required a sizable amount of "blurry" spectrograms, that could not be obtained by re-feeding the training data (the 593 sentences from speaker czr) through the mapper model, due to significant over-fitting.

Consequentially, we've recorded the next 539 sentences from the CMU Arctic dataset, as spoken by speaker czr.

After converting the extracted features from the newly recorded data to all 3 speakers, we've excluded some of the converted spectrogram outputs that were overly erratic due to audio lengths exceeding 4 seconds. Finally, the dataset for training the Enhancer consists of 496 pairs of "fake" and "real" Log-Mel-Spectrograms for each of the 3 target speakers (aew, bdl and rms).

For training the spectrogram enhancer, every "fake" spectrogram needed to have the same length as the "real" one it is paired with. To solve this issue, we performed an alignment for every pair using a DTW algorithm using the cosine distance between the paired spectrograms, based on [Kamb].

The default hyper-parameters caused very chaotic fluctuation in the generative loss, but we eventually found that a learning rate of 5e-10, Adam optimizer betas of 0.8 and 0.95 and a batch size of 4 made the generative loss plot resemble more of a convergence, as shown in figure 3.6. Moreover, the default configuration stated "no good stopping rule" for the training, advising to keep every checkpoint, so we left the process running for 42 hours on an NVIDIA RTX 3090.

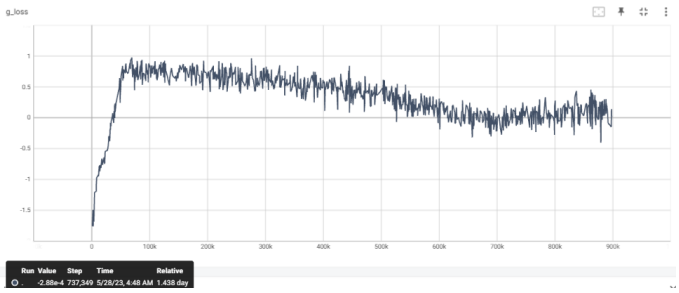

Figure 3.6: Plot graph of the spectrogram enhancer's generative loss during training, using our hyper-parameter settings.

We also made use of the newly recorded data to train the 3 mappers with even more data, namely a total of 1100 pairs of utterances for each. We reserved 32 waveforms for testing. We increased the batch size to 20, and let the training run for 2000 epochs. The L1 loss graph is shown in figure 3.7.

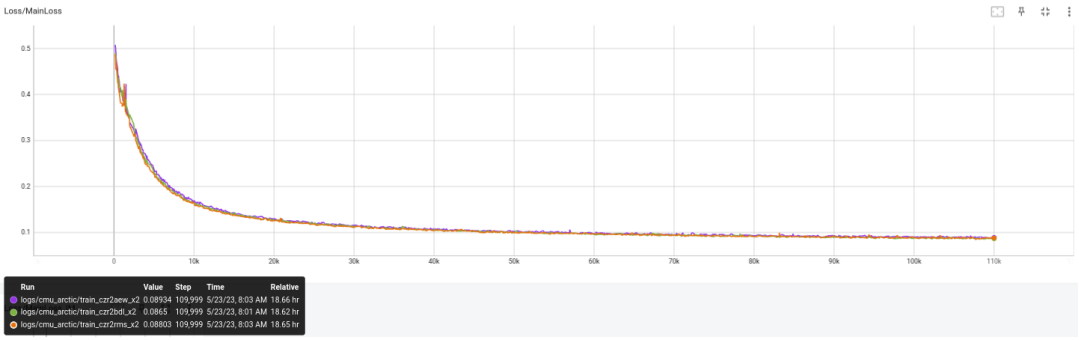

Figure 3.7: Plot graph of the L1 loss function for the final 2000 epoch training of the 3 mapper models.

Upon passing the data we reserved for testing through the newly trained mappers, the resulting Log-Mel-Spectrograms were looking more detailed when visualized (Figure 3.8), and when given through the same vocoders we've trained earlier, the speeches were noticeably more comprehensible, as expected.

We then inserted the spectrogram enhancer into the pipeline, between the mapper and the vocoder, and the enhanced spectrograms appeared "brighter" upon visualisation (Figure 3.8), and, unexpectedly, a detail enhancement effect was barely

visible. When passed to a vocoder model, the resulting audio was just as comprehensible as before, but with a slight decrease in loudness compared to the outputs generated from the unenhanced spectrograms, and with a subtle chorusing effect added.

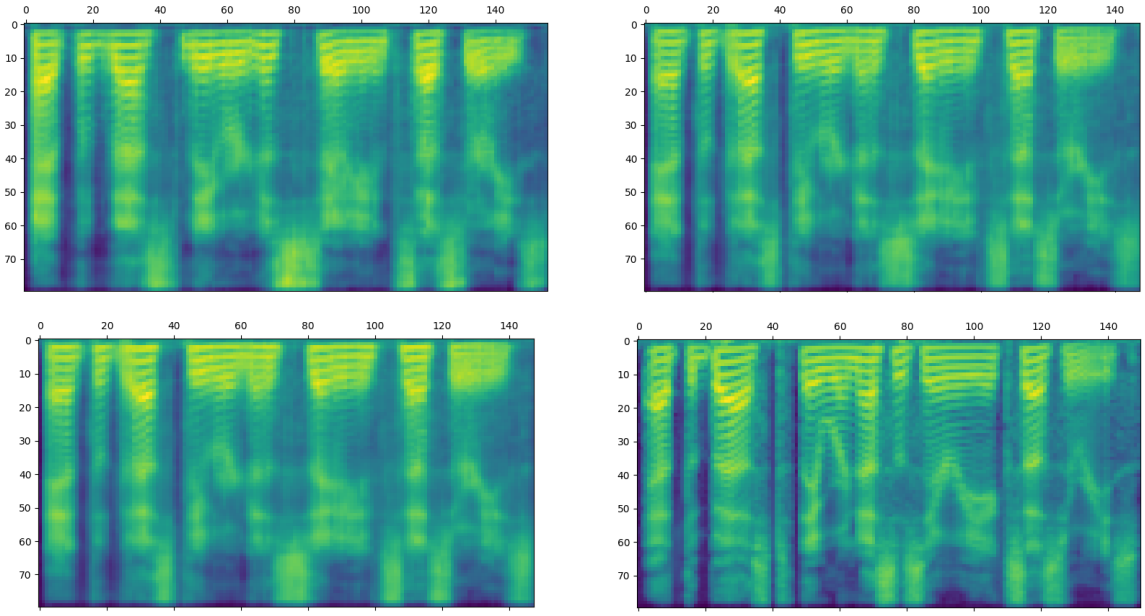

Figure 3.8: Four Log-Mel-Spectrograms of the sentence "At the best, they were necessary accessories."

Top-left: Output from czr-to-bdl mapper trained on 593 parallel utterances.

Top-right: Output from czr-to-bdl mapper trained on 1100 parallel utterances.

Bottom-left: The spectrogram on the top-right, processed by the enhancer.

Bottom-right: The ground truth speech, by speaker bdl.

3.3 Objective Results

We used the PyWORLD library for extracting the spectral envelope features from the ground truth and converted variants of the 32 speeches from CMU Arctic that we reserved for testing, and these were converted to 25th-order MCEP features using the pysptk library.

Finally, using the sprocket-vc library, we performed a DTW alignment for each pair of matching features, and computed the Mel-Cepstral Distortion (MCD) be-

tween them. The measurements made in three experimental conditions we described at 3.2 are presented in the table below (Figure 3.9).

	Mel-Cepstral Distortion (MCD) [dB]		
	czr-to-aew	czr-to-bdl	czr-to-rms
Trained on 593 parallel speeches	**8.21 +- 0.47**	8.28 +- 0.47	8.40 +- 0.44
Trained on 1100 parallel speeches	8.32 +- 0.54	**8.15 +- 0.40**	**8.29 +- 0.54**
Trained on 1100 parallel speeches + enhanced	8.63 +- 0.51	8.36 +- 0.37	8.45 +- 0.46

Figure 3.9: Table of measured MCDs in the three experimental conditions that yielded comprehensible speeches.

3.4 Integrated Pipeline & Implementation

After the completion of all the experiments above, we proceeded to integrate the following Voice Conversion pipeline within a Django REST API:

- Preprocessor - An algorithm that splits the raw waveforms received from the client into segments of at most 4 seconds, and the same feature extractor used for the experiments above was used for extracting the Log-Mel-Spectrograms from each segment;
- Converter - The model that performs the mapping of spectrograms from source speaker ”czr” to the target speaker contained in the request;
- Enhancer - The speaker-independent model meant for restoring details within the converted spectrograms;
- Vocoder - The model that synthesizes the waveforms from the enhanced converted spectrograms.

All of the neural models will be pre-loaded onto the server’s video memory, and the configuration settings (paths, speaker labels, etc.) for the API have been provided within a JSON file.

The provided API takes POST requests that have the following information embedded in their body in JSON form:

- Target Speaker - A string that can be one of the three names given for the target speakers on which the models have been trained: ”aew”/”bdl”/”rms”;

- Audio Format - A string containing the audio coding format of the file containing the speech data; currently, the only accepted format is Wave (”WAV”), but we did provide an implementation draft for handling FLAC files;
- Sample Rate - The number of samples per second of coded in the source audio file (e.g. 32000);
- Audio Data - The source file, assumed to be an audio recording of a speech from speaker czr, with all of its headers, encoded as a Base64 string.

API responses have the same structure as the requests, except the audio data will be the converted speech audio file, with the matching sample rate (i.e. 16000).

Chapter 4

Application

In this chapter we'll discuss, from a Software Engineering perspective, a mobile client application that we have provided, that serves as a portable and intuitive medium of interaction with the REST API we have proposed at Chapter 3, as well as a tool that covers a user's basic needs in speaker-identity modification tasks.

4.1 Analysis & Design

Firstly, we'll describe the application's structure and behaviour, without referring to the implementation details, and as high-level as possible.

4.1.1 Use Cases & Functionalities

We've considered the following functional requirements necessary for the mobile client application, such that it will cover a user's basic needs in speaker-identity modification tasks:

1. Speech recording using the device microphone;
2. Converting the original speeches to one of the provided speakers via REST API communication, assuming the device and the API are online;
3. Local storage of the original and converted speeches;
4. Ability to browse the stored speeches by their speaker class;
5. Ability to rename or delete the stored speeches;
6. Playing the original and converted speech content using the device loudspeaker(s);
7. Sharing the stored speeches via an external app (e.g. Google Drive, WhatsApp).

Every requirement except for #2 will be available for offline use. These have been compiled into a Use Case diagram below (4.1).

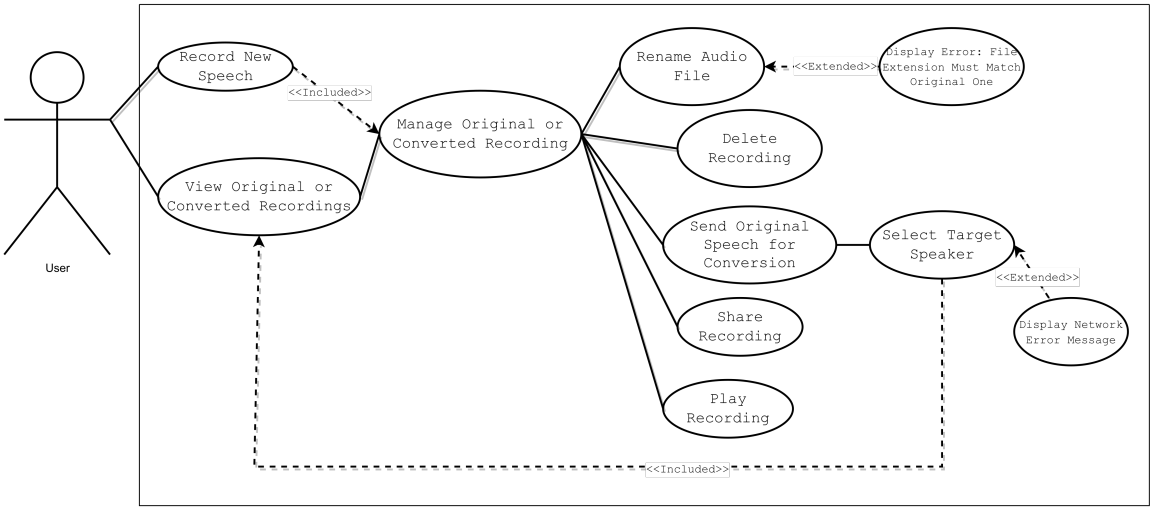

Figure 4.1: Use Case Diagram for our provided mobile application.

4.1.2 High-level Structure

Structure-wise, each of the main high-level components of our application can be directly associated with one or more of the functional requirements stated at 4.1.1:

- The Local Storage - requirement #1 presumes the writing of the recorded audio data on disk, requirement #3 stated the actual persistence of the files, requirement #5 needs storage access for managing the files on disk and requirement #6 presumes the loading of audio files in memory in order for them to be played back.
- The Local Database and the Database Manager - a database that contains metadata of all the recordings on the storage, can simplify the implementation of requirement #4.
- Core Device APIs - these give access to native code tied to the device's hardware.
  - Network Connectivity - used by requirement #2 for communication with the REST API.
  - Microphone Utility - needed by requirement #1.
  - Speaker Utility - needed by requirement #6.

- Service layer - 4 sets of calls to the native APIs that are wrapped into more simplified interfaces, each set being dedicated to a single requirement.
- UI - the visual elements that the application user interacts with, effectively tied to all of the client's functional requirements.
- View-Models - the layer where the information that needs to be presented by the UI is being stored, and where some UI state management is being handled (more on this at 4.2.2).
- Controller - a unified interface provided for the View-Models that acts like a mediator of the 4 Service sub-components.

The relationships between these high-level components have been modeled in the diagram 4.2 below.

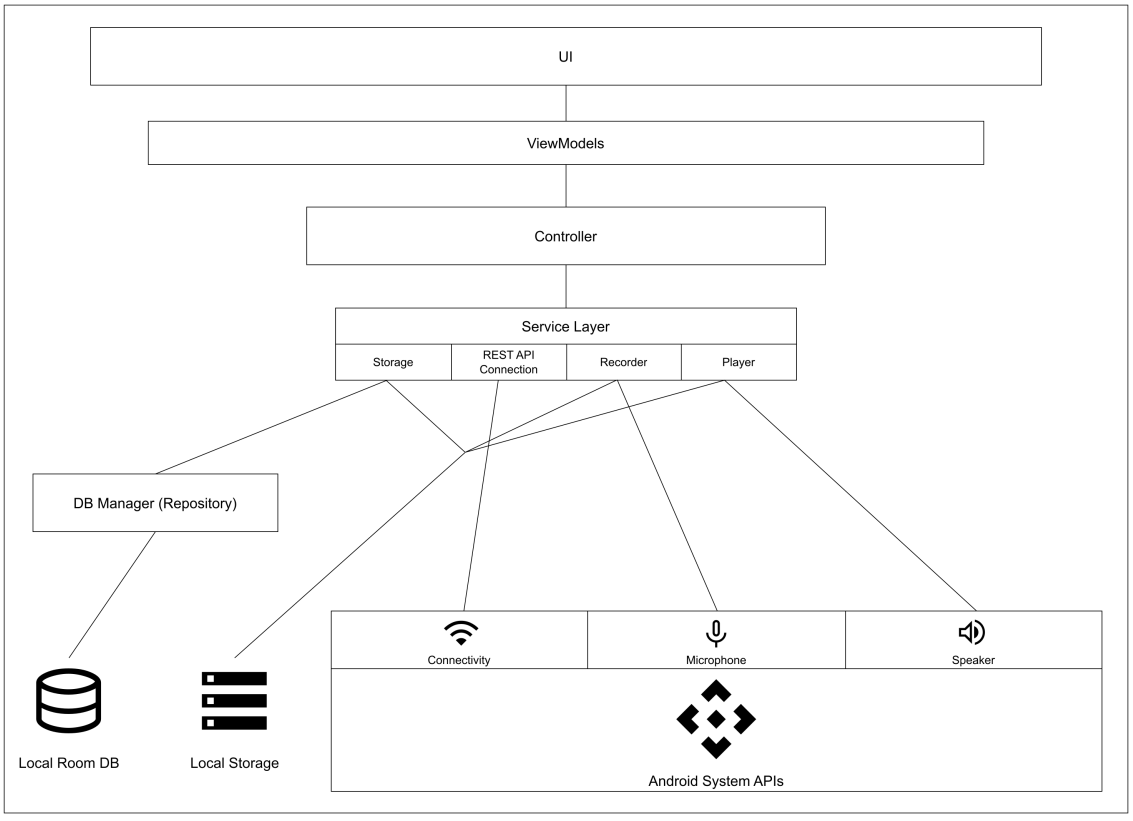

Figure 4.2: Diagram that highlights the main high-level components of the provided mobile application and how they are linked to one another.

A hypothetical implementation of our application using strictly the MVVM architecture without a Controller would require adopting the "Clean architecture". This implies the wrapping of the business logic implemented by the Service components into multiple UseCase components for separation of concerns, that would further be injected into the View-Models. Since the Service sub-components have

high degree of interaction between them, we felt that by providing a centralized interface for the app business logic in the form of a Controller would simplify the architecture's dependency graph.

4.1.3 Flow of Functionalities

Audio Recording

For this functionality, the user must interact with a button on the Main Screen to start and stop the recording, and hold the device microphone reasonably close to the speech source, while in a noise-free, indoor environment.

After starting, the UI component will run a stopwatch of the elapsed recording time. The ViewModel component will hold the elapsed seconds since the recording start, and every second, the UI will request this number to be incremented, and will be presented back immediately. The Recorder Service sub-component will handle the recording process in the background using the native Android microphone API.

Once the user stops the recording or the recording time limit has been reached, the Recorder Service finishes writing the audio data to disk, and the Storage Service adds an entity containing information about the recording to the local database. Finally, options for Audio Playback, Speech Conversion, deletion, renaming and sharing will be shown on the screen.

This flow is illustrated in the Sequence Diagram 4.3 below.

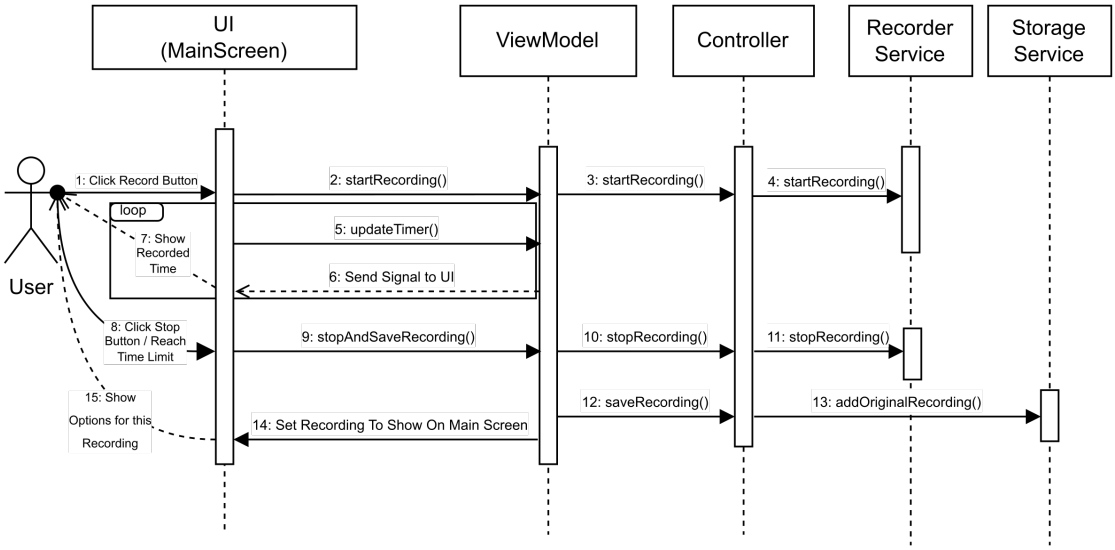

Figure 4.3: Sequence Diagram that describes the flow of the recording functionality (requirement #1)

Speech Conversion

After recording a speech, the user can interact with a button that will open a Dialog box over the Main screen, dedicated to choosing the target speaker to convert the speech to.

The list of available target speakers will be held by the ViewModel component, and the UI will present it in a drop-down menu placed in the Dialog box. Once the user selects a target speaker, the ViewModel will hold it. Afterwards, the user may interact with a button that triggers a REST API call via the Connection Service. Since the response containing the converted speech will take some time, the user will be prompted with a ”Waiting...” indicator.

If the response from the API was received successfully, the Storage Service will handle writing the received audio file to disk, and adding an entry to the local database. Additionally, the Controller component will signal the UI navigator to redirect the user to the Recordings screen.

In the case of a network issue regarding the API call, the Controller will signal the UI to close the Dialog box and show a ”Network Error” indicator.

This flow is shown by the Sequence Diagram 4.4 below.

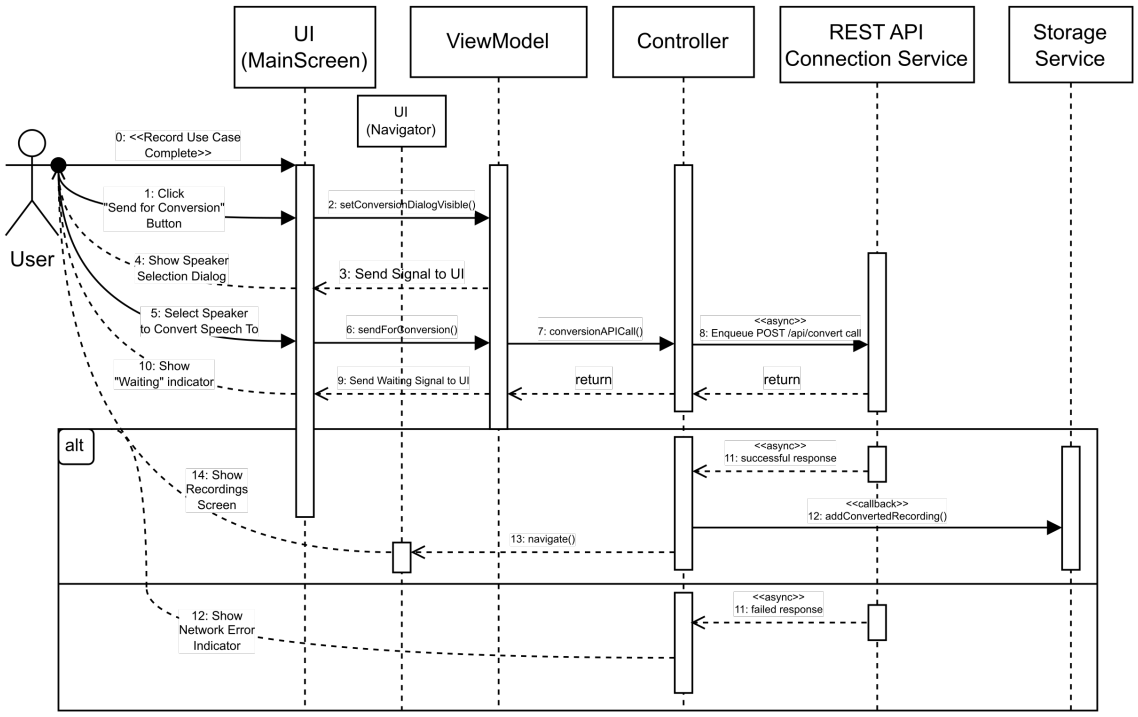

Figure 4.4: Sequence Diagram that describes the flow of the speech conversion via API functionality (requirement #2)

Browsing Saved Recordings

The application's Recordings screen will allow the user to view the converted speeches filtered by the target speaker, as well as the original recorded speeches. This menu will be accessible via a button that's always present on the Main screen.

A drop-down menu will contain the filtering options, and upon one's selection, the ViewModel will signal the Storage Service to modify the entity list bound to the UI according to the newly applied filter, as illustrated in the Sequence Diagram 4.5.

Each recording will be shown in the UI as a "card", containing the file name, datetime, and three action buttons for deletion, sharing and a "View" action. The "View" action will redirect the user to the Main screen, where the usual options for any newly recorded speech (except for Conversion if a non-original recording was chosen for viewing) will be presented.

This flow is illustrated by the Sequence Diagram 4.4 below.

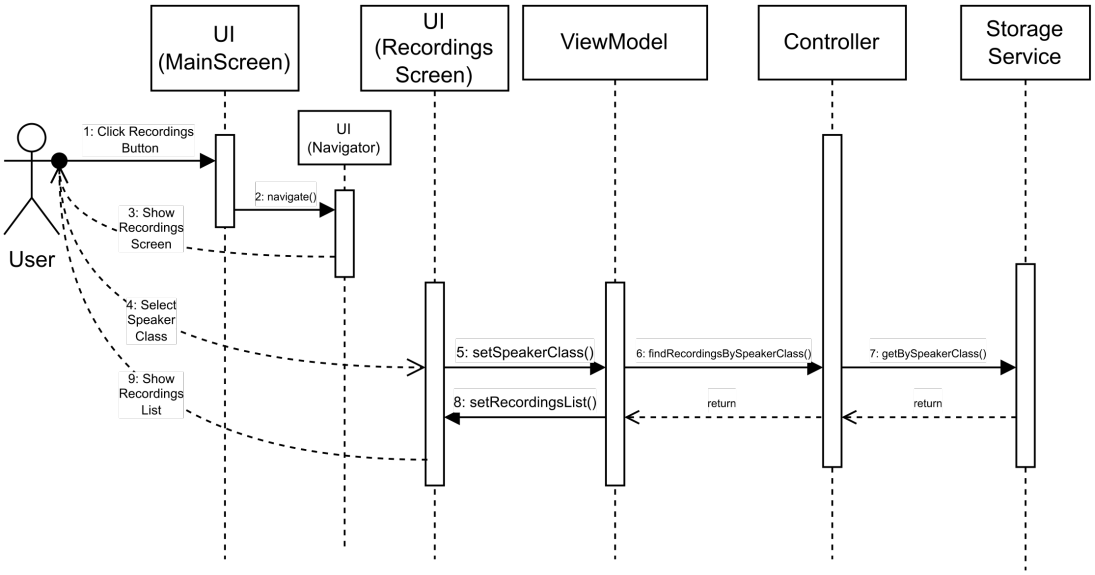

Figure 4.5: Sequence Diagram that describes the flow of the speech browsing functionality (requirement #4)

Audio Playback

An audio player will be available on the Main screen once a Recording was selected for viewing. This will allow the user to play any audio file saved by the application on the local storage, also pause its playback or seek through the file using a slider.

When the user interacts with the "Play" button, the Player Service sub-component will start playback of the audio file through the device's loudspeaker(s) via the native Android API. Additionally, the ViewModel will query the Player Service in small intervals for the elapsed playback time, so that it is presented in the UI audio player, and also for updating the slider's position frequently.

During playback, the user may pause or seek through the audio file by interacting with the player controls, which will make the correspondent calls to the Player Service. Once the track has reached its end, the player will reset the elapsed time to 0.

This flow is illustrated by the Sequence Diagram 4.6 below.

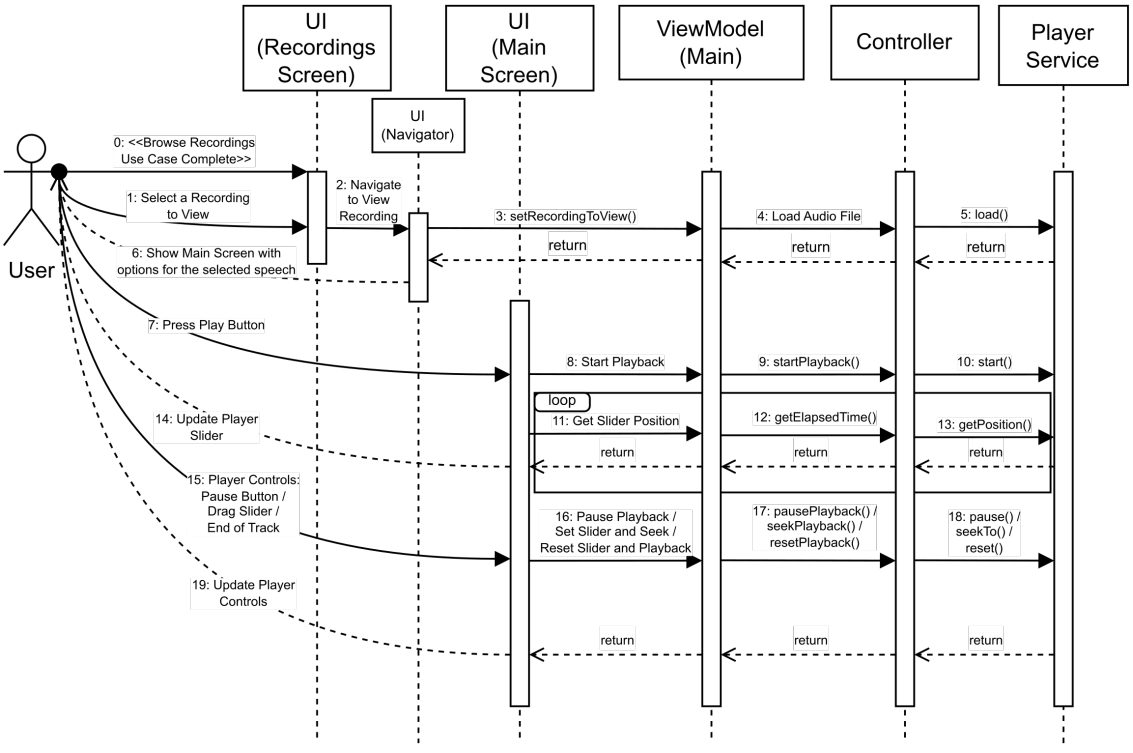

Figure 4.6: Sequence Diagram that describes the flow of the audio playback functionality (requirement #6)

4.2 Implementation

Next, we'll discuss a set of modern technologies for building mobile applications, how we've leveraged them in our application, and some of the implementation details.

4.2.1 Technologies used

Kotlin

The Kotlin programming language was introduced by JetBrains as a modern, better alternative to Java, with its first stable release in 2016.

Since it also compiles to Java Virtual Machine (JVM) bytecode, it has been designed to be inter-operable with pre-existing Java code. This feature has made its adoption process within the Android development space quite seamless, to the point where Google has named Kotlin "the preferred language for Android app development" [Lar19] in 2019.

Some of Kotlin's advantages over Java are:

- Multi-paradigm - It supports object-oriented patterns, as well as functional patterns (e.g. passing lambda functions), since functions are "First-Class Objects".
- Null-safety - As also featured in languages like C#, TypeScript and Dart, the null-coalescing ("?") and safe navigation ("?.") operators are very useful for defining different behaviours when dealing with objects that are explicitly defined by the developer as nullable.
- Coroutines - Meant for simplifying invocation of asynchronous methods on different threads, a very common requirement in mobile app development. For example, functions that perform network-related operations must be invoked under the I/O thread pool instead of the Main thread pool reserved for UI operations.
- Extension functions - Ability to define additional behaviour for any class, without the need of inheritance (e.g. custom datetime formatting functions).
- Data classes - Meant for eliminating the need for explicit definition of constructors, getters and setters, that also support object destructuring, similarly to how the ES6 JavaScript standard first introduced it. These are very useful for defining entities and DTOs.

- String interpolation - A concise way of building character strings, a feature that dates back to languages like Bash, Perl and PHP.

All of these amount to a more ergonomic language in the development process, with a more concise syntax.

Android Jetpack Libraries

- **Compose UI** - A native UI library for Kotlin that leverages declarative and functional programming patterns for tight coupling of the visual and behavioural aspects of UI elements. The ergonomics of such an approach to building UIs have been proven by the popularity of libraries like React.js.
- **Room** - A persistence library built on top of SQLite that relies on meta-programming for compile-time syntax checking of SQL queries and generating the code required for SQL statement execution.

Dagger-Hilt DI

Dependency Injection frameworks are used for decoupling the installation of singleton components of an application's architecture from its behaviour at runtime. Their use is highly recommended in more complex apps, with the benefits of keeping the code more structured, and eliminating lots of boilerplate code.

Dagger is a DI framework for Java and Kotlin that relies on meta-programming to build an application's dependency graph at compile-time. Hilt is the standard way of incorporating Dagger within Android applications, aiming to simplify the Dagger-related infrastructure.

Retrofit

As a HTTP client, we used Retrofit, which is built on top of OkHttp, for its suitability for REST APIs, type-safety and simplicity of implementation.

4.2.2 Concrete Implementation Aspects

Persistence & File Management

We chose to employ a local database of entities that holds the filename, the creation datetime, the speaker class and the format (file extension) of each recording saved by the application.

The Room persistence library makes the definition of the required database operations very concise. All that is needed is the definition of the method headers annotated with the operation type or the SQL query, as shown in the code snippet 4.7, inside of a Data Access Object (DAO). These methods are then called in the Repository.

The Recordings screen requires a live view of the filtered local data, since deletion should be doable on the same screen as the Recordings list. For this, the LiveData class was quite convenient, since it is an observable data container that signals data changes automatically to its observers. The Room DAO interoperates seamlessly with this class, the only requirement being to specify LiveData as the method return type, as shown in the code snippet 4.7.

```
@Query("SELECT * FROM recording WHERE speaker_class = :speakerClass")
fun findBySpeakerClass(speakerClass: SpeakerClass): LiveData<List<Recording>>
```

Figure 4.7: Code snippet from the DAO for definition of the filter operation.

An alternative would've been to work exclusively with direct storage access, but the data binding process would've been more convoluted and the overall code would've been less clean.

UI & State Management

The classic way of building Android UIs is declaring and stylizing UI resources in XML documents and defining their behaviour by attaching Java or Kotlin code to the respective resources via Adapters and View-Holders.

Compose UI eliminates the decoupling of the UI's visual aspects from the behavioural ones by allowing the elements to be defined as Kotlin functions that are annotated as "Composables". These functions can be called within one another's scopes to create the UI structural tree. The library also provides pre-defined Composables that serve as alternatives to the traditional UI elements (e.g. LazyColumn for RecyclerView), that can be given stylizing options (i.e. alignment, padding, color,

etc.) via parameters upon invocation (as shown for LazyColumn in snippet 4.8).

The Android Jetpack suite also provides a ViewModel component that may be used to persist UI state that may be collected by UI components, and to access business logic.

We found this component to work quite well when coupled with Kotlin StateFlows, which are thread-safe data streams we used for state persistence of UI elements (i.e. Player controls, Dialog box state, Pending API response etc.) and bound data (i.e. Recordings list, selected Recording, etc.).

The Composable scopes can also collect and observe state information of the data provided via StateFlows by the ViewModel (as shown in snippet 4.8), that upon emitting a state change, a re-render of the containing Composable is triggered.

```
@Composable
fun RecordingsList(
    viewModel: RecordingsScreenViewModel,
    navControllerAccessObject: NavControllerAccessObject
) {

    val recordingsList by viewModel.currentRecordingsList.collectAsState()
    val recordingsListState by recordingsList.observeAsState()

    Surface(

        modifier = Modifier.fillMaxWidth()
    ) {

        LazyColumn(

            contentPadding = PaddingValues(horizontal = 16.dp, vertical = 16.dp),
            verticalArrangement = Arrangement.spacedBy(16.dp)
        ) { this: LazyListScope

            items(recordingsListState ?: listOf()) { this: LazyItemScope  it: Recording

                RecordingCard(viewModel, it, navControllerAccessObject)
            }

        }
    }
}
```

Figure 4.8: Code snippet for definition of the UI element representing the list of Recordings.

Network & REST API Communication

The definition of the API call for the Conversion functionality is made really easy by Retrofit, as shown in the code snippet 4.9.

The JSON object sent by Retrofit will contain the target speaker class, the audio file format, the sample rate and the audio file itself in Wave format (including the matching WAV file headers), coded as a Base64 string. We chose to record speech in this format due to its lack of lossy compression that may add audio artifacts to the input data, which are undesirable in the conversion pipeline.

```
@POST("convert/")
fun convert(@Body dto: ConversionDTO): Call<ConversionDTO>
```

Figure 4.9: Code snippet for definition of the Conversion API call.

The call generated by the Retrofit method will be executed asynchronously by the Controller. As mentioned in the Sequence Diagram 4.4, this call returns immediately, and the Controller signals the ViewModel through a StateFlow that a response is pending. Upon completion, the callback will signal otherwise through the same StateFlow. This whole process is shown in the code snippet 4.10.

```
fun conversionAPICall(
    filename: String,
    targetSpeaker: SpeakerClass,
    onSuccessUI: () -> Unit,
    onNetworkFailureUI: (String) -> Unit,
    ioScope: CoroutineScope
) {

    val bytes = storage.readRecording(SpeakerClass.ORIGINAL, filename)

    Log.i( tag: "AppController",  msg: "Successfully read recording $filename from local storage.")

    val dto = ConversionDTO(
        targetSpeaker = targetSpeaker.toString(),
        audioFormat = RecorderService.FILE_FORMAT.toString(),
        sampleRate = RecorderService.RECORDER_SAMPLE_RATE,
        audioData = bytes.toBase64()
    )

    retrofit.convert(dto).enqueue(object : Callback<ConversionDTO> {

        override fun onResponse(call: Call<ConversionDTO>, response: Response<ConversionDTO>) {...}
        override fun onFailure(call: Call<ConversionDTO>, t: Throwable) {...}
    })

    awaitingResponse.value = true
}
```

Figure 4.10: Code snippet for execution of the Conversion API call.

Runtime & Flow of dependencies

Unlike the classic approach involving XML and Adapters to Android UIs where there is an Activity for every screen, Compose UI-based applications may have a single Activity holding a Navigator component that handles switching between Composables for the current screen.

For each of the two application screens, there is a separate ViewModel to handle UI state, data binding and bridging the business logic and the UI. The two ViewModels are created and injected into the screen Composables by the Navigator, as shown in the code from Figure 4.11.

The rest of the DI is being handled by Hilt. The Controller, the Service classes and the Repository have their dependencies specified in a secondary constructor that is annotated as a Hilt injection point. The dependency graph can be observed in the flow of the composition relationships from the Class Diagram (Figure 4.12) we provided.

```
@Composable
fun AppNavigation() {

    val navController = rememberNavController()

    val mainScreenViewModel: MainScreenViewModel = hiltViewModel()
    val recordingsScreenViewModel: RecordingsScreenViewModel = hiltViewModel()

    val accessObject = NavControllerAccessObject(
        navController,
        mainScreenViewModel,
        recordingsScreenViewModel
    )

    NavHost(navController = navController, startDestination = "main") { this: NavGraphBuilder

        composable(
            route = "main"
        ) { it: NavBackStackEntry

            MainScreen(
                navControllerAccessObject = accessObject,
                viewModel = mainScreenViewModel
            )
        }

        composable(
            route = "recordings"
        ) { it: NavBackStackEntry

            RecordingsScreen(
                navControllerAccessObject = accessObject,
                viewModel = recordingsScreenViewModel
            )
        }
    }
}
```

Figure 4.11: The Compose UI Navigator injecting the ViewModels into the declared screen Composables.

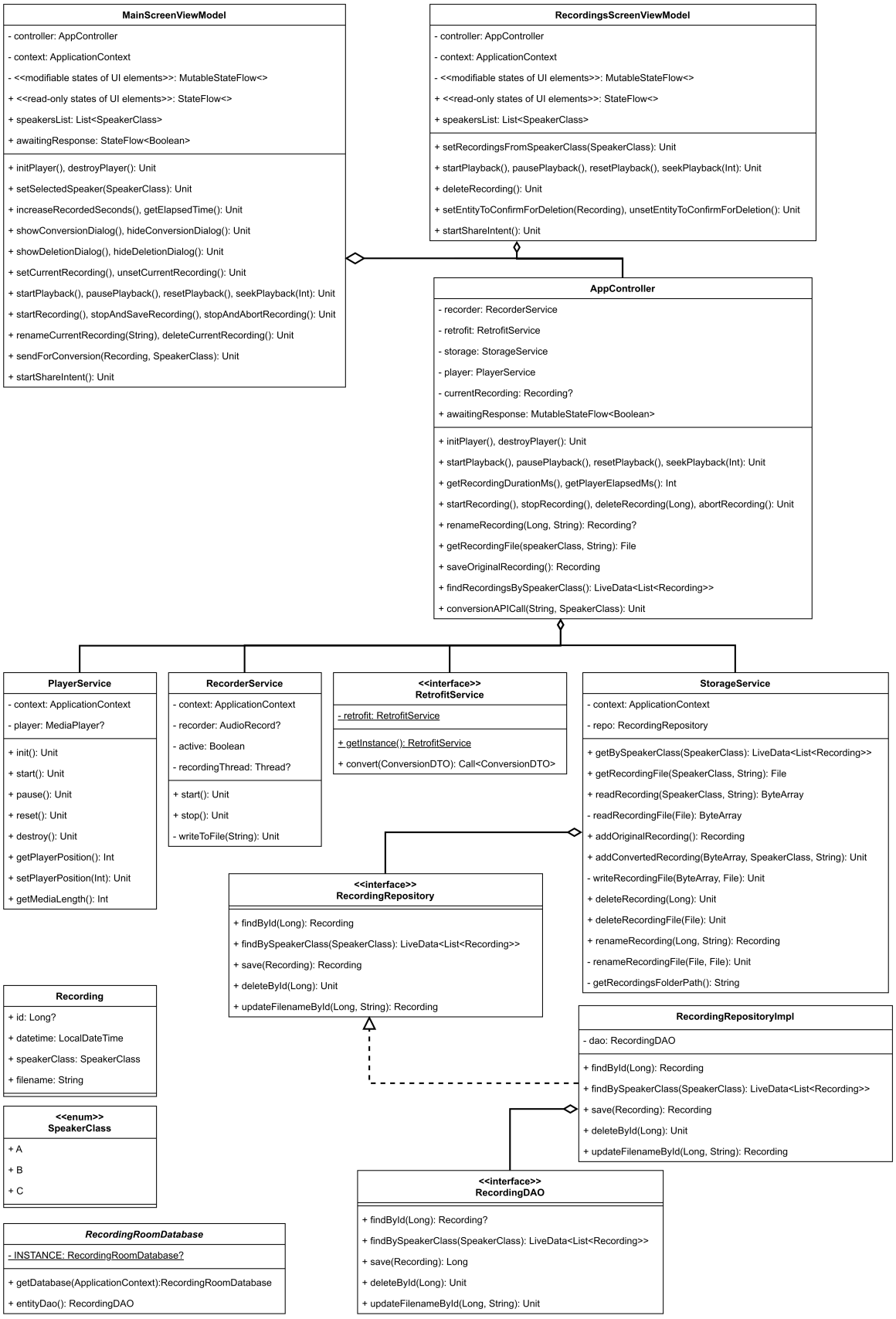

Figure 4.12: UML Diagram of the main classes involved in the functionality of our provided client application.

4.3 Usage

Lastly, we'll describe a series of interactions with our provided application's UI, purely from an end user's standpoint. Figures 4.13, 4.14, 4.15 and 4.16 show the UI flow for all the functionalities. We've also provided a short, demonstrative video[1].

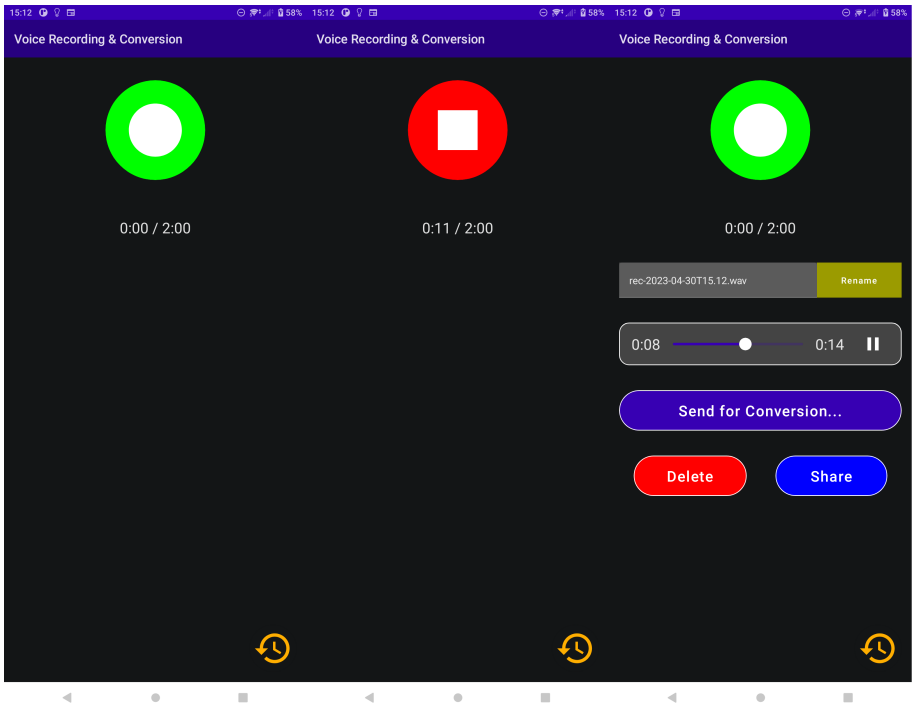

Figure 4.13:

When opening the app, the user is greeted with a button to start recording using the device microphone, a timer that shows the elapsed recording time once started and a floating action button for navigating to the recordings screen.

The recording can be stopped by clicking the stop button, or it will stop automatically at the 2 minute time limit.

Once stopped, the user can use the text field below the timer to rename the file to their liking, as long as the original file extension is kept and the new filename does not contain system-reserved characters (i.e. `|\?*<":>+[]/'`).

Below the filename text field, there is the Audio Player for playing the newly recorded audio track that contains the following controls:

- A timer for showing the elapsed playback time;
- A slider for seeking through the audio track;
- A Play/Pause button.

[1] - Video for a short demonstration of the provided app.

Upon clicking the ”Send for Conversion...” button, the Dialog box in Figure 4.14 will be shown.

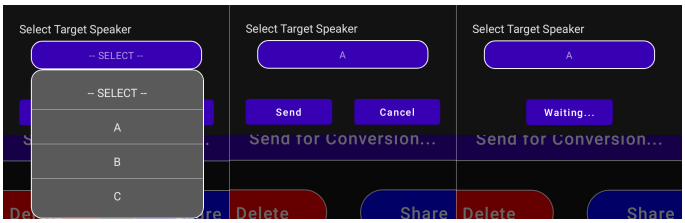

Figure 4.14: Note - The available target speakers will be the ones accepted by the API.

This box will have a drop-down menu for selecting the speaker to which the original speech must be converted to, a ”Send” and a ”Cancel” button. The target speakers from which the user may choose are limited to the ones that the API models were trained on (listed at 3.4).

The ”Send” button must be clicked to begin the Conversion process, and a ”Waiting...” indicator will be shown until a network error occurs or the server has successfully responded.

If the app could not connect to the API, the Dialog box closes itself, and a ”Network Error” indicator is shown on the Main screen.

If the app has successfully received the converted speech, it will navigate to the Recordings screen and show the recordings under the speaker class the new converted speech belongs to (shown in Figure 4.15).

The ”Share” and ”Delete” buttons from the Main screen have the same function as the ones from the recording cards, but for the recording that is currently viewed.

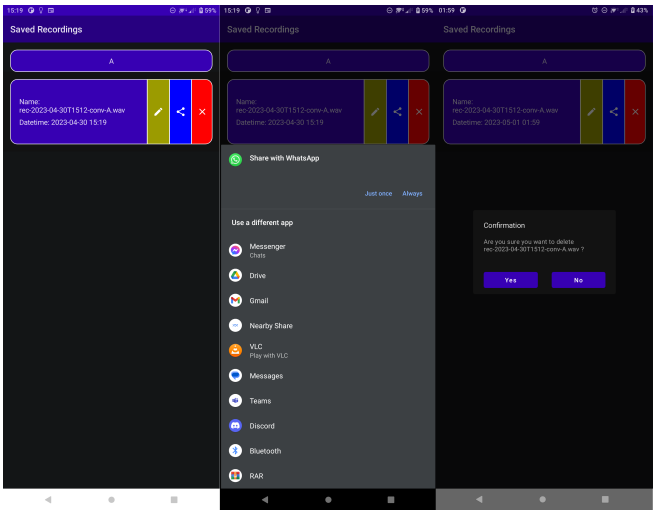

Figure 4.15:

At the Recordings screen, the user can view the list of recordings under a certain speaker class, each of their names and creation datetimes being showed in card-like containers. For each recording card, there are 3 buttons for the ”View”, ”Share” and ”Delete” actions.

The user can click the ”Share” button to redirect usage of a recording file to an external Android app.

The ”Delete” button will show a confirmation dialog first, and after clicking ”Yes”, the Dialog box will close and the recording will be deleted from the list.

The Recordings screen can show the saved speeches under all speaker classes, including the original ones. This can be changed from the drop-down menu (shown in Figure 4.16), and the list will be updated according to the choice.

For any recording card, the ”View” button can be clicked to show the options for renaming and Playback of an audio file at the Main screen like in Figure 4.13. Conversion is available only when an original speech was selected.

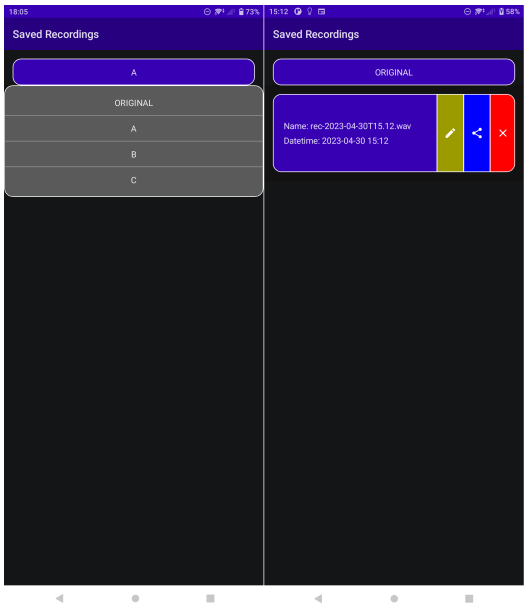

Figure 4.16:

Chapter 5

Conclusions

Concluding our paper, we've provided an overview of the literature regarding Machine Learning and Deep Learning techniques applied in the domain of Voice Conversion, by structuring it on top of the main feature extraction methods, as well as observing a number of innovations brought to the Deep Learning paradigm that proved quite relevant and valuable in the VC space.

Then, we conducted a series of experiments starting from one of the most notable approaches [KTK21]. We made an attempt to enable conversion from an open speaker set inspired by the original authors' previous work, we expanded the CMU Arctic dataset with a large amount of data, and leveraged other state-of-the-art neural modules [HMK$^+$] that were originally conceived for the Text-to-Speech field, namely the HiFiGAN neural vocoder for waveform synthesis and a StyleGAN-based model repurposed for enhancing Log-Mel-Spectrograms.

Additionally, we've also developed a mobile client application for Android using a modern toolkit, purposed for basic Voice Conversion-related tasks like speech recording, file storage, management and sharing, audio playback, and voice conversion to a fixed number of speakers. The conversion task itself is carried out by communicating with a REST API that runs received speech through a pipeline consisting of a number of preprocessing modules and neural models.

Bibliography

[FBGO09] Daniel Felps, Heather Bortfeld, and Ricardo Gutierrez-Osuna. Foreign accent conversion in computer assisted pronunciation training. *Speech Communication*, 51(10):920–932, 2009. Spoken Language Technology for Education.

[GH10] Michael Gutmann and Aapo Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. *Journal of Machine Learning Research - Proceedings Track*, 9:297–304, 01 2010.

[HMK+] Eric Harper, Somshubra Majumdar, Oleksii Kuchaiev, Li Jason, Yang Zhang, Evelina Bakhturina, Vahid Noroozi, Sandeep Subramanian, Koluguri Nithin, Huang Jocelyn, Fei Jia, Jagadeesh Balam, Xuesong Yang, Micha Livne, Yi Dong, Sean Naren, and Boris Ginsburg. "NeMo: a toolkit for Conversational AI and Large Language Models".

[HPH18] Daniel W. Hook, Simon J. Porter, and Christian Herzog. Dimensions: Building context for search and evaluation. *Frontiers in Research Metrics and Analytics*, 3:23, 2018. https://www.frontiersin.org/articles/10.3389/frma.2018.00023/pdf.

[HSVG12] Elina Helander, Hanna Silen, Tuomas Virtanen, and Moncef Gabbouj. Voice conversion using dynamic kernel partial least squares regression. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(3):806–817, 2012.

[Kama] Hirokazu Kameoka. Official PyTorch implementation for ConvS2S-VC.

[Kamb] Herman Kamper. Dynamic Time Warping Notebook.

[Kay] Tomoki Kayashi. Unofficial PyTorch implementation for Parallel-WaveGAN.

[KHT+20] Hirokazu Kameoka, Wen-Chin Huang, Kou Tanaka, Takuhiro Kaneko, Nobukatsu Hojo, and Tomoki Toda. Many-to-many voice transformer network, 2020.

[KK17] Takuhiro Kaneko and Hirokazu Kameoka. Parallel-data-free voice conversion using cycle-consistent adversarial networks, 2017.

[KKB20] Jungil Kong, Jaehyeon Kim, and Jaekyoung Bae. Hifi-gan: Generative adversarial networks for efficient and high fidelity speech synthesis, 2020.

[KKdB+19] Kundan Kumar, Rithesh Kumar, Thibault de Boissiere, Lucas Gestin, Wei Zhen Teoh, Jose Sotelo, Alexandre de Brebisson, Yoshua Bengio, and Aaron Courville. Melgan: Generative adversarial networks for conditional waveform synthesis, 2019.

[KKTH19] Takuhiro Kaneko, Hirokazu Kameoka, Kou Tanaka, and Nobukatsu Hojo. Cyclegan-vc2: Improved cyclegan-based non-parallel voice conversion, 2019.

[KKTH20] Takuhiro Kaneko, Hirokazu Kameoka, Kou Tanaka, and Nobukatsu Hojo. Cyclegan-vc3: Examining and improving cyclegan-vcs for mel-spectrogram conversion. pages 2017–2021, 10 2020.

[KKTH21] Takuhiro Kaneko, Hirokazu Kameoka, Kou Tanaka, and Nobukatsu Hojo. Maskcyclegan-vc: Learning non-parallel voice conversion with filling in frames, 2021.

[KLA+20] Tero Karras, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. Analyzing and improving the image quality of stylegan, 2020.

[KMKd99] Hideki Kawahara, Ikuyo Masuda-Katsuse, and Alain de Cheveigné. Restructuring speech representations using a pitch-adaptive time–frequency smoothing and an instantaneous-frequency-based f0 extraction: Possible role of a repetitive structure in sounds1speech files available. see http://www.elsevier.nl/locate/specom1. *Speech Communication*, 27(3):187–207, 1999.

[Kor] Roman Korostik. "GAN-based Spectrogram Enhancer".

[KTK+18] Hirokazu Kameoka, Kou Tanaka, Damian Kwasny, Takuhiro Kaneko, and Nobukatsu Hojo. Convs2s-vc: Fully convolutional sequence-to-sequence voice conversion, 2018.

[KTK21] Hirokazu Kameoka, Kou Tanaka, and Takuhiro Kaneko. Fasts2s-vc: Streaming non-autoregressive sequence-to-sequence voice conversion, 2021.

[Lar19] Frederic Lardinois, 2019. "Kotlin is now Google's favorite language for Android app development".

[MYO16] Masanori MORISE, Fumiya YOKOMORI, and Kenji Ozawa. World: A vocoder-based high-quality speech synthesis system for real-time applications. *IEICE Transactions on Information and Systems*, E99.D:1877–1884, 07 2016.

[OLV18] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding, 2018.

[PGB+11] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, et al. The kaldi speech recognition toolkit. In *IEEE 2011 workshop on automatic speech recognition and understanding*, number CONF. IEEE Signal Processing Society, 2011.

[SLW+16] Lifa Sun, Kun Li, Hao Wang, Shiyin Kang, and Helen Meng. Phonetic posteriorgrams for many-to-one voice conversion without parallel data training. In *2016 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6, 2016.

[SZS+18] Berrak Sisman, Mingyang Zhang, Sakriani Sakti, Haizhou Li, and Satoshi Nakamura. Adaptive wavenet vocoder for residual compensation in gan-based voice conversion. In *2018 IEEE Spoken Language Technology Workshop (SLT)*, pages 282–289, 2018.

[TBT07] Tomoki Toda, Alan W. Black, and Keiichi Tokuda. Voice conversion based on maximum-likelihood estimation of spectral parameter trajectory. *IEEE Transactions on Audio, Speech, and Language Processing*, 15(8):2222–2235, 2007.

[THA+17] Kei Tanaka, Sunao Hara, Masanobu Abe, Masaaki Sato, and Shogo Minagi. Speaker dependent approach for enhancing a glossectomy patient's speech via gmm-based voice conversion. pages 3384–3388, 08 2017.

[TKKH18] Kou Tanaka, Hirokazu Kameoka, Takuhiro Kaneko, and Nobukatsu Hojo. Atts2s-vc: Sequence-to-sequence voice conversion with attention and context preservation mechanisms, 2018.

[vdODZ+16] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *CoRR*, abs/1609.03499, 2016.

[vNNK20] Benjamin van Niekerk, Leanne Nortje, and Herman Kamper. Vector-quantized neural networks for acoustic unit discovery in the zerospeech 2020 challenge, 2020.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[WSRS+17] Yuxuan Wang, RJ Skerry-Ryan, Daisy Stanton, Yonghui Wu, Ron J. Weiss, Navdeep Jaitly, Zongheng Yang, Ying Xiao, Zhifeng Chen, Samy Bengio, Quoc Le, Yannis Agiomyrgiannakis, Rob Clark, and Rif A. Saurous. Tacotron: Towards end-to-end speech synthesis, 2017.

[WVK+13] Zhizheng Wu, Tuomas Virtanen, Tomi H. Kinnunen, Chng Eng Siong, and Haizhou Li. Exemplar-based voice conversion using non-negative spectrogram deconvolution. In *Speech Synthesis Workshop*, 2013.

[YSK19] Ryuichi Yamamoto, Eunwoo Song, and Jae-Min Kim. Parallel wavegan: A fast waveform generation model based on generative adversarial networks with multi-resolution spectrogram, 2019.

[ZPIE17] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks, 2017.

[ZSR+18] Mingyang Zhang, Berrak Sisman, Sai Sirisha Rallabandi, Haizhou Li, and Li Zhao. Error reduction network for dblstm-based voice conversion. In *2018 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, pages 823–828, 2018.

[ZSZL20] Mingyang Zhang, Berrak Sisman, Li Zhao, and Haizhou Li. Deepconversion: Voice conversion with limited parallel training data. *Speech Communication*, 122:31–43, 2020.