Project Report

on

Replication of Voice Using Deep Learning

Submitted to

Sant Gadge Baba Amravati University

In partial Fulfillment of the Requirement

For the Degree of

Bachelor of Engineering in

Computer Science and Engineering

Submitted by:

Saurabh Kedar
Sarvesh Sonar
Smitesh Sonar
Trunay Wanjari

Under the Guidance of

Dr. J. M. Patil

Department of Computer Science and Engineering

Shri Sant Gajanan Maharaj College of Engineering,

Shegaon – 444 203 (M.S.)

2022-23

SHRI SANT GAJANAN MAHARAJ COLLEGE OF ENGINEERING,
SHEGAON – 444 203 (M.S.)
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that **Mr. Saurabh Kedar, Mr. Sarvesh Sonar, Mr. Smitesh Sonar** and **Mr. Trunay Wanjari,** students of final year B.E. in the year 2022-23 of Computer Science and Engineering Department of this institute has completed the project work entitled **"Replication of Voice Using Deep Learning"** based on syllabus and has submitted a satisfactory account of his work in this report which is recommended for the partial fulfillment of degree of Bachelor of Engineering in Computer Science and Engineering.

Dr. J. M. Patil
Project Guide

31/5/23
Dr. S. B. Patil
Head of Department
Computer Science and Engineering

Dr. S. B. Somani
Principal
S.S.G.M. College of Engineering,
Shegoan

SHRI SANT GAJANAN MAHARAJ COLLEGE OF ENGINEERING,

SHEGAON – 444 203 (M.S.)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that the project work entitled "**Replication of Voice Using Deep Learning**" submitted by **Mr. Saurabh Kedar, Mr. Sarvesh Sonar, Mr. Smitesh Sonar** and **Mr. Trunay Wanjari,** students of final year B.E. in the year 2022-23 of Computer Science and Engineering Department of this institute, is a satisfactory account of his work based on syllabus which is recommended for the partial fulfillment of degree of Bachelor of Engineering in Computer Science and Engineering.

Internal Examiner **External Examiner**

Date: **Date:**

Acknowledgement

The real spirit of achieving a goal is through the way of excellence and lustrous discipline. I would have never succeeded in completing my task without the cooperation, encouragement and help provided to me by various personalities.

I would like to take this opportunity to express my heartfelt thanks to my guide Prof. J. M. Patil, for his esteemed guidance and encouragement, especially through difficult times. His suggestions broaden my vision and guided me to succeed in this work. I am also very grateful for his guidance and comments while studying part of my seminar and learnt many things under his leadership.

I extend my thanks to Dr. S.B. Patil Head of Computer Science & Engineering Department, Shri Sant Gajanan Maharaj College of Engineering, Shegaon for their valuable support that made me consistent performer.

I also extend my thanks to Dr. S. B. Somani, Principal Shri Sant Gajanan Maharaj College of Engineering, Shegaon for their valuable support.

Also, I would like to thanks to all teaching and non-teaching staff of the department for their encouragement, cooperation and help. My greatest thanks are to all who wished me success especially my parents, my friends whose support and care makes me stay on earth.

Mr. Saurabh Kedar
Mr. Sarvesh Sonar
Mr. Smitesh Sonar
Mr. Trunay Wanjari
Final Year B. E. Sem-VIII, CSE
Session 2022-23

Contents

Abstract

Deep learning grows have led to impressive findings from a text to speech domain. A neural network with deep neural networks is typically a compilation of countless hours of expertly recorded discourse from a single speaker had been employed to train the entire system.. Recent research introduced a three-stage process to replicating voice training from only a few seconds of input without restructuring the character's language. We adapt the framework with a newer vocoder model to make it run in real-time. Audio synthesis is growing as a technological study hotspot. Human-computer interaction will keep evolving so that computers can interact with humans. Voice will be a great deal stating to connection among machines and human beings' technique in the subsequent years considering it has numerous benefits rather than a single process. Voice replication is a branch for voice technology that can replicate a specific person's voice. Also, to avoid the robotic generated voice, which is common on numerous gadgets but is not always interactive with humans. A method over completing voice communication in real time replication with only a few samples is proposed to solve the issue of offering a substantial amount of specimens and a long delay over conversation replication in prior years. We available an artificial neural text to speech over the network formation system the fact that will produce audio expression in the sounds belonging to various speakers, which include those that went unnoticed through training.

List of Figures and Tables

Abbreviations

TTS	Text-to-speech
AI	Artificial Intelligence
SV2TTS	Speech Vector to Text-to-speech
LSTM	Long short-term memory
GE2E	Generalized end-to-end
UMAP	Uniform Manifold Approximation and Projection
STFT	Short Time Fourier transform
MLPG	Maximum Likelihood Parameter Generation algorithm
SPSS	Statistical Parametric Speech Synthesis
HMM	Hidden Markov model
ASR	Automatic Speech Recognition

Chapter 1

INTRODUCTION

1. INTRODUCTION

1.1 Preface-

Deep learning models have become predominant in many fields of applied machine learning. Text-to-speech (TTS), the process of synthesizing artificial speech from a text prompt, is no exception. Deep models that would produce more natural-sounding speech than the traditional concatenative approaches begun appearing in 2016. Much of the research focus has been since gathered around making these deep models more efficient, sound more natural, or training them in an end-to-end fashion. Inference has come from being hundreds of times slower than real-time on GPU to possible in real-time on a mobile CPU. As for the quality of the generated speech, Shen et al. demonstrate near-human naturalness. Interestingly, speech naturalness is best rated with subjective metrics; and comparison with actual human speech leads to the conclusion that there might be such a thing as" speech more natural than human speech".

Datasets of professionally recorded speech are a scarce resource. Synthesizing a natural voice with a correct pronunciation, lively intonation and a minimum of background noise requires training data with the same qualities. Furthermore, data efficiency remains a core issue of deep learning. Training a common text-to-speech model such as Tacotron typically requires hundreds of hours of speech. Yet the ability to generate speech with any voice is attractive for a range of applications, be they useful or merely a matter of customization. Research has led to frameworks for voice conversion and voice cloning. They differ in that voice conversion is a form of style transfer on a speech segment from a voice to another, whereas voice cloning consists in capturing the voice of a speaker to perform text-tospeech on arbitrary inputs.

While the complete training of a single-speaker TTS model is technically a form of voice cloning, the interest rather lies in creating a fixed model able to incorporate newer voices with little data. The common approach is to condition a TTS model trained to generalize to new speakers on an embedding of the voice to clone. The embedding is low-dimensional and derived by a speaker encoder model that takes reference speech as input. This approach is typically more data efficient than training a separate TTS model for each speaker, in addition to being orders of magnitude faster and less computationally expensive. Interestingly, there is a large discrepancy between the duration of reference speech needed to clone a voice among the different methods, ranging from half an hour per speaker to only a few seconds. This factor is usually determining of the similarity of the generated voice with respect to the true voice of the speaker.

Our objective is to achieve a powerful form of voice cloning. The resulting framework must be able to operate in a zero-shot setting, that is, for speaker's unseen during training. It should incorporate a speaker's voice with only a few seconds of reference speech. These desired results are shown to be fulfilled. Their results are impressive, but not backed by any public implementation. We reproduce their framework and make our implementation open-source. In addition, we integrate a model based on the framework to make it run in real-time, i.e., to generate speech in a time shorter or equal to the duration of the produced speech.

The structure of this document goes as follows. We begin with a short introduction on methods of TTS with machine learning. Follows a review of the evolution of the state of the art for TTS with speech naturalness as the core metric. We conclude with a presentation of a toolbox we designed to interface the framework.

1.2 Overview-

In the making of replication toolbox, we used SV2TTS (Speaker Verification to Text-to-Speech) framework, which is a real-time voice cloning system. The framework is based on previous works by Google, including the GE2E loss, Tacotron, and WaveNet models. SV2TTS is a three-stage pipeline that consists of a speaker encoder, a synthesizer, and a vocoder.

1.2.1 Speaker Encoder-

The first stage of the framework involves a speaker encoder, which takes a short utterance from a single speaker and derives an embedding—a meaningful representation of the speaker's voice. This embedding is designed to place similar voices closer together in a latent space.

To avoid segments that are mostly silent when sampling partial utterances from complete utterances, we use the webrtcvad python package to perform Voice Activity Detection (VAD). This yields a binary flag over the audio corresponding to whether or not the segment is voiced. We perform a moving average on this binary flag to smooth out short spikes in the detection, which we then binarize again[1]. Finally, we perform a dilation on the flag with a kernel size of s + 1, where s is the maximum silence duration tolerated. The audio is then trimmed of the unvoiced parts. We found the value s = 0.2s to be a good choice that retains a natural speech prosody. This process is illustrated in Figure 1.2.1.1. A last preprocessing step applied to the audio waveforms is normalization, to make up for the varying volume of the speakers in the dataset.

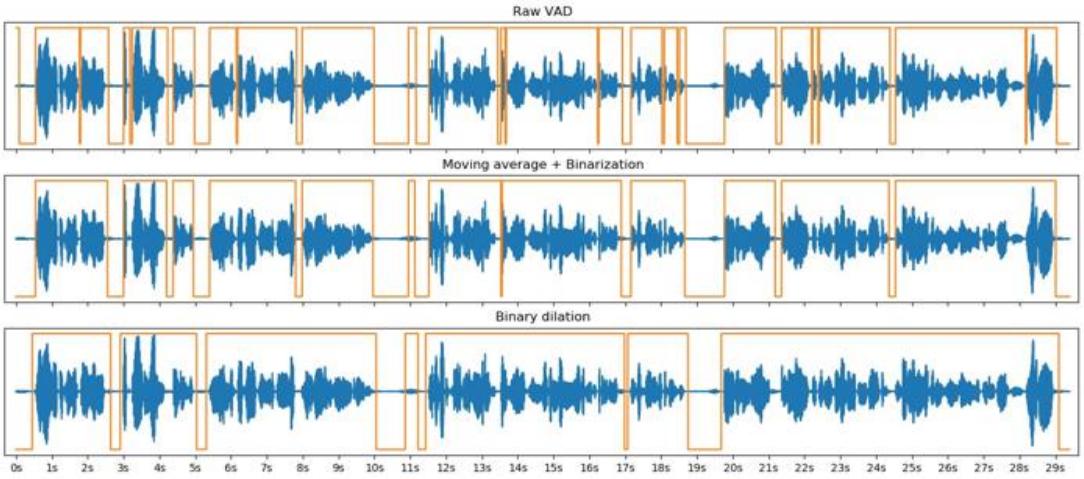


Figure 1.2.1.1: The steps to silence removal with VAD, from top to bottom. The orange line is the binary voice flag where the upper value means that the segment is voiced, and unvoiced when lower.

The combined several noisy datasets to make for a large corpus of speech of quality similar to what is found in the wild. These datasets are LibriSpeech, VoxCeleb1, VoxCeleb2 and an internal dataset, to which we do not have access. LibriSpeech is a corpus of audiobooks making up for 1000 hours of audio from 2400 speakers, split equally in two sets “clean” and “other”. The clean set is supposedly made up of cleaner speech than the other set, even though some parts of the clean set still contain a lot of noise. VoxCeleb1 and VoxCeleb2 are made up from audio segments extracted from you tube videos of celebrities. VoxCeleb1 has 1.2k speakers, while VoxCeleb2 has about 6k. Both these datasets have non-English speakers. We used heuristics based on the nationality of the speaker to filter non-English ones out of the training set in VoxCeleb1, but couldn’t apply those same heuristics to VoxCeleb2 as the nationality is not referenced in that set. Note that it is unclear without experimentation as to whether having non-English speakers hurts the training of the encoder (the authors make no note of it either). All these datasets are sampled at 16kHz.

The authors test different combinations of these datasets and observe the effect on the quality of the embeddings. They adjust the output size of LSTM model (the size of the embeddings) to 64 or 256 according to the number of speakers. They evaluate the subjective naturalness and similarity with ground truth of the speech generated by a synthesizer trained from the embeddings produced by each model. They also report the equal error rate of the encoder on speaker verification.

These results indicate that the number of speakers is strongly correlated with the good performance of not only the encoder on the verification task, but also of the entire framework on the quality of the speech generated and, on its ability, to clone a voice. The small jump in naturalness, similarity and EER gained by including VoxCeleb2 could possibly indicate that the variation of languages is hurting the training. The internal dataset of the authors is a proprietary voice search corpus from 18k English speakers. The encoder trained on this dataset performs significantly better, however we only have access to public datasets. We thus proceed with LibriSpeech-Other, VoxCeleb1 and VoxCeleb2.

We train the speaker encoder for one million steps. To monitor the training, we report the EER and we observe the ability of the model to cluster speakers. We periodically sample a batch of 10 speakers with 10 utterances each, compute the utterance embeddings and projecting them in a two-dimensional space with UMAP. As embeddings of different speakers are expected to be further apart in the latent space than embeddings from the same speakers, it is expected that clusters of utterances from a same speaker form as the training progresses. We report our UMAP projections in Figure 1.2, where this behaviour can be observed.

As mentioned before, the authors have trained their model for 50 million steps on their proprietary dataset. While both our dataset and our model are smaller, our model still hasn't converged at 1 million steps. The loss decreases steadily with little variance and could still decrease more, but we are bound by time [1].

The resulting model yields very strong results nonetheless. In fact, we computed the test set EER to be 4.5%. This is an astonishingly low value in light of the 10.14% of the authors for the same set with 50 times more steps. We do not know whether our model is actually performing that well or if the EER computation procedure of the authors is different enough than ours to produce values so far apart.

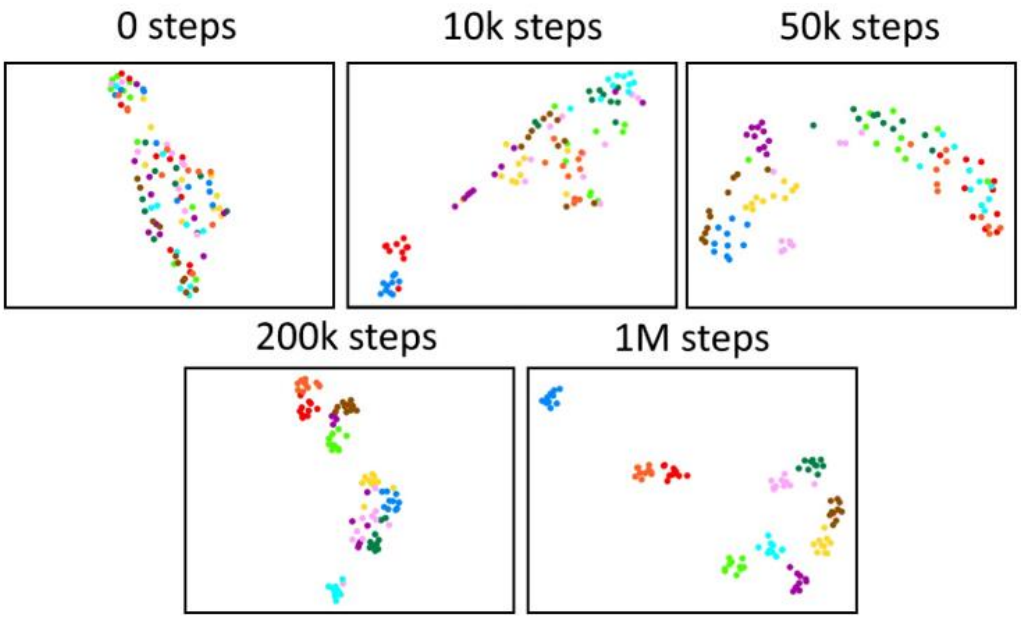


Figure 1.2.1.2: UMAP projections of utterance embeddings from randomly selected batches from the train set at different iterations of our model.

1.2.2 Synthesizer-

The second stage is the synthesizer, which generates a spectrogram from text, conditioned on the speaker's embedding. The synthesizer used in SV2TTS is Tacotron 2, which is a popular model known for its text-to-speech synthesis capabilities. In this stage, WaveNet, a deep generative model, is not utilized.

In SV2TTS, the authors consider two datasets for training both the synthesizer and the vocoder. These are LibriSpeech-Clean which we have mentioned earlier and VCTK which is a corpus of only 109 native English speakers recorded with professional equipment. The speech in VCTK is sampled at 48kHz and down sampled to 24kHz in their experiments, which is still higher than the 16kHz sampling of LibriSpeech. They find that a synthesizer trained on LibriSpeech generalizes better than on VCTK when it comes to similarity, but at the cost of speech naturalness. They assess this by training the synthesizer on one set, and testing it on the other[1]. We decided to work with the dataset that would offer the best voice cloning similarity on unseen speakers, and therefore picked LibriSpeech. We have also tried using the newer LibriTTS dataset created by the Tacotron team. This dataset is a cleaner version of the whole LibriSpeech corpus with noisy speakers pruned out, a higher sampling rate of 24kHz and the punctuation that LibriSpeech lacks.

Following the preprocessing recommendations, we use an Automatic Speech Recognition (ASR) model to force-align the LibriSpeech transcripts to text. We found the Montreal Forced Aligner1 to perform well on this task. We've also made a cleaner version of these alignments public2 to save some time for other users in need of them. With the audio aligned to the text, we split utterances on silences longer than 0.4 seconds. This helps the synthesizer to converge, both because of the removal of silences in the target spectrogram, but also due to the reduction of the median duration of the utterances in the dataset, as shorter sequences offer less room for timing errors. We ensure that utterances are not shorter than 1.6 seconds, the duration of partial utterances used for training the encoder, and no longer than 11.25 seconds so as to save GPU memory for training. We do not split on a silence that would create an utterance too short or too long if possible. The distribution of the length of the utterances in the dataset is plotted in Figure 1.2.2.1. Note how long silences already account for 64 hours (13.7%) of the dataset.

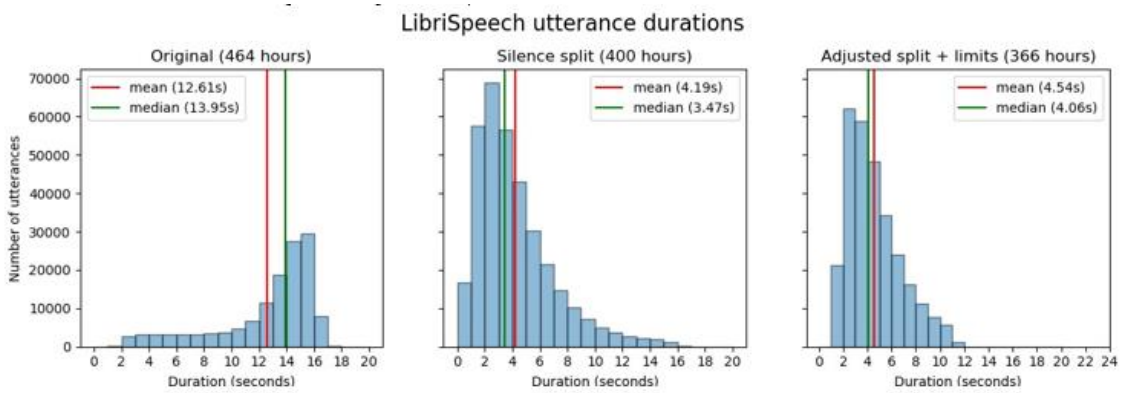


Figure 1.2.2.1: (left) Histogram of the duration of the utterances in LibriSpeech-Clean, (middle) after splitting on silences, (right) after constraining the length and readjusting the splits.

Isolating the silences with force-aligning the text to the utterances additionally allows to create a profile of the noise for all utterances of the same speaker. We use a python implementation of the LogMMSE algorithm. LogMMSE cleans an audio speech segment by profiling the noise in the earliest few frames (which will usually not contain speech yet) and updating this noise profile on non-speech frames continuously

throughout the utterance. We adapt this implementation to profile the noise and to clean the speech in two separate steps. On par with the authors, we found this additional preprocessing step to greatly help reduce the background noise of the synthesized spectrograms.

It is difficult to provide any quantitative assessment of the performance of the model. We can observe that the model is producing correct outputs through informal listening tests, but a formal evaluation would require us to setup subjective score polls to derive the MOS. While some authors we referred to could do so, this is beyond our reach. In the case of the synthesizer however, one can also verify that the alignments generated by the attention module are correct. We plot an example in Figure 1.2.2.2. Notice the number of decoder steps matching the number of frames predicted by the number of decoder outputs per step [1]. Notice also how the predicted spectrogram is smoother than the ground truth, a typical behavior of the model predicting the mean in presence of noise.

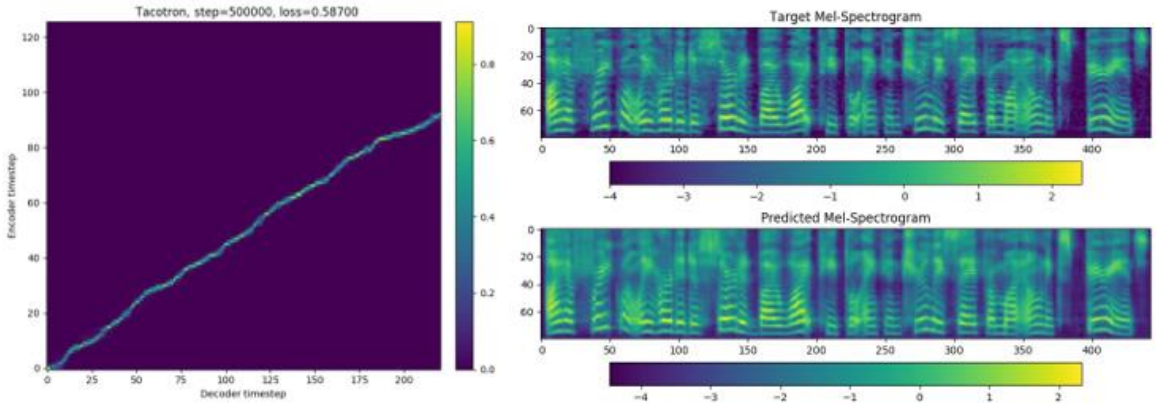


Figure 1.2.2.2: (left) Example of alignment between the encoder steps and the decoder steps. (right) Comparison between the GTA predicted spectrogram and the ground truth spectrogram.

Before training the vocoder, we can evaluate some aspects of the trained synthesizer using Griffin-Lim as vocoder. Griffin-Lim is not a machine learning model but rather an iterative algorithm that estimates the source audio signal of a spectrogram. Audio generated this way typically conserves few of the voice characteristics of the speaker, but the speech is intelligible. The speech generated by the synthesizer matches correctly the text, even in the presence of complex or fictitious words. The prosody is however sometimes unnatural, with pauses at unexpected locations in the sentence, or

the lack of pauses where they are expected. This is particularly noticeable with the embedding of some speakers who talk slowly, showing that the speaker encoder does capture some form of prosody. The lack of punctuation in LibriSpeech is partially responsible for this, forcing the model to infer punctuation from the text alone. This issue was highlighted by the authors as well, and can be heard on some of their samples3 of LibriSpeech speakers. The limits we imposed on the duration of utterances in the dataset are likely also problematic. Sentences that are too short will be stretched out with long pauses, and for those that are too long the voice will be rushed. When generating several sentences at inference time, we need to manually insert breaks to delimit where to split the input text so as to synthesize the spectrogram in multiple parts. This has the advantage of creating a batch of inputs rather than a long input, allowing for fast inference.

We can further observe how some voice features are lost with Griffin-Lim by computing the embeddings of synthesized speech and projecting them with UMAP along with ground truth embeddings. An example is given in Figure 1.2.2.3. We observe that the synthesized embeddings clusters are close to their respective ground truth embeddings cluster. The loss of emerging features is also visible, e.g for the pink, red and the two blue speakers the synthesized utterances have a lower inter-cluster variance than their ground truth counterpart. This phenomenon occurs with the gray and purple speakers as well.

Tacotron usually operates faster than real-time. We measure an inference speed of 5× to 10× real-time, by comparing the time of generation with the duration of the generated spectrogram.

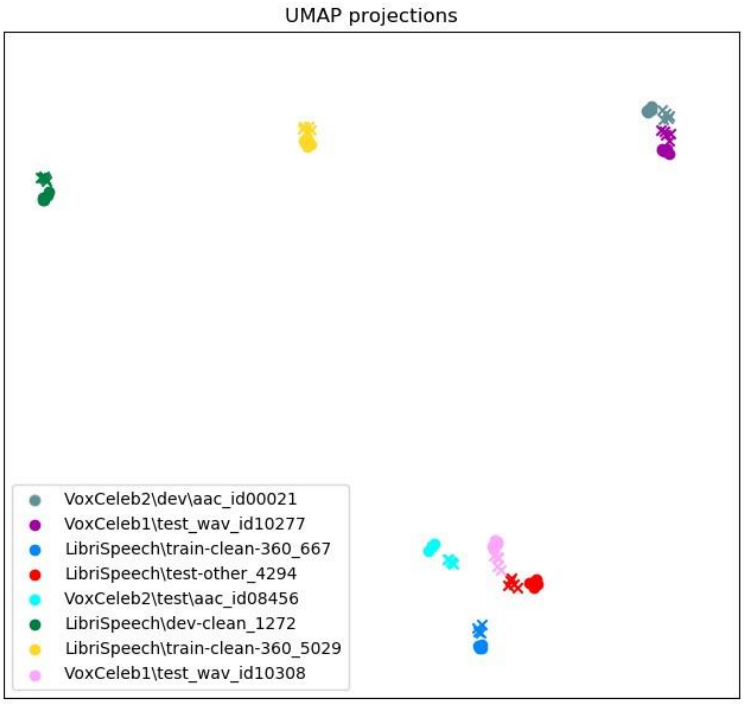


Figure 1.2.2.3: Projections of ground truth embeddings and of Griffin-Lim synthesized speech embeddings generated from the same ground truth embeddings. Ground truth embeddings are drawn with circles and synthesized embeddings with crosses.

1.2.3 Vocoder-

The final stage of the pipeline is the vocoder, which converts the spectrograms generated by the synthesizer into an audio waveform. The authors of the SV2TTS framework employed WaveNet as the vocoder. This means that the entire Tacotron 2 framework is effectively reapplied during the vocoder stage.

When dealing with short utterances, the vocoder usually runs below real-time. The inference speed is highly dependent of the number of folds in batched sampling. Indeed, the network runs nearly in constant time with respect to the number of folds, with only a small increase in time as the number of folds grows. We find it is simpler to talk about a threshold duration of speech above which the model runs in real time. On our setup, this threshold is of 12.5 seconds; meaning that for utterances that are shorter than this threshold, the model will run slower than real-time. It seems that performance varies unexpectedly with some environment factors (such as the operating system) on PyTorch, and therefore we express our results with respect to a single same configuration.

The implementation on our hands does not have the custom GPU operation and implementing it is beyond our capabilities. Rather, we focus on the pruning aspect mentioned by the authors. They claim that a large sparse WaveRNN will perform better and faster than a smaller dense one. We have experimented with the pruning algorithm but did not complete the training of a pruned model, due to time limits. This is a milestone we hope to achieve at a later date.

Sparse tensors are, at the time of writing, yet an experimental feature in PyTorch. Their implementation might not be as efficient as the one the authors used. Through experiments, we find that the matrix multiply operation addmm for a sparse matrix and a dense vector only breaks even time-wise with the dense-only addmm for levels of sparsity above 91%. Below this value, using sparse tensors will actually slow down the forward pass speed. The authors report sparsity levels of 96.4% and 97.8% (Kalchbrenner et al., 2018, Table 5) while maintaining decent performances. Our tests indicate that, at best, a sparsity level of 96.4% would lower the real-time threshold to 7.86 seconds, and a level of 97.8% to 4.44 seconds. These are optimistic lower bounds on the actual threshold due to our assumption of constant time inference, and also because some layers in the model cannot be scarified. This preliminary analysis indicates that pruning the vocoder would be beneficial to inference speed.

We did manage to get a prototype working by February, but this model is no longer compatible with changes we've made to the framework. We are determined to provide a working implementation before the defense of this thesis, but we cannot report of new experiments for now. Our impressions of the prototype was that our implementation was successful in creating a TTS model that could clone most voices, but not some uncommon ones . Some artifacts and background noise were present due to the poor quality of our synthesizer, which is why we had to revise the quality of our data and our preprocessing procedures. One drawback of the SV2TTS framework is the necessity to train models in sequential order. Once a new encoder is trained, the synthesizer must be retrained and so must the vocoder. Waiting for models to train so as to know on what to focus next has been a recurring situation in the development of our framework.

1.3 Motivation-

The motivation behind voice replication is to enable the creation of artificial voices that can sound like specific individuals, while still preserving the semantic content of the speech. This can be useful for a variety of applications, such as creating personalized voice assistants or generating synthetic speech for people who have lost their ability to speak. By studying voice cloning, researchers can gain a deeper understanding of the parameters that influence speech signals and human pronunciation mechanisms. Additionally, the development of effective voice replication techniques can have practical applications in fields such as entertainment, education, and healthcare.

1.4 Proposed Objectives-

1. To generate a TTS framework that can generate natural-sounding speech for a variety of individuals using as little data as possible.
2. To develop a zero-shot learning scenario where only a few minutes of transcribed speech from a target speaker are used to create new speech in that speaker's voice, with no modification to the model's variables.
3. To combine a speaker encoder model with Tacotron2 and speaker verification training to create a multi-speaker system for voice cloning.
4. To convert the voice into speaker embedding and the Encoder module would be able to assist us with this process.
5. To extract the final output speech to address issues such as slowdown errors and noise in Tacotron2 models when dealing with out-of-set speakers.
6. To create a methodology that can produce stable results with new target speaker data, while also ensuring precise pronunciation, fluent phrases, and minimal background noise.

Chapter 2

LITERATURE SURVEY

2. LITERATURE SURVEY

[1] Transfer learning from speaker verification to multispeaker text-to-speech synthesis

Ye Jia, Yu Zhang, Ron J. Weiss, et al., [1] The article defines A neural network that is artificial connections system built around multispeaker TTS as that is a combination makes use of the speaker's encoder system to generate superior speech for unknown its speakers. The platform requires minimal information to operate and doesn't demand superior clean conversation or transcripts. But the capacity of the system to achieve living thing-level genuineness and accent transfer is limited. Future work might focus on emulate adaptation along with conditioning the virtual instrument on separate from speaker and accent embeddings to overcome these limitations.

[2] Research on Voice Cloning with a Few Samples

Li Zhao, Feifan Chen, et al., [2] This article presents a new method for replicating of words founded on the a few samples that are faster and can be used on low-performance devices. The method employs a three-module structure of encoder, synthesizer, and vocoder and can be quickly optimized and improved. However, Chinese speech cloning lags behind English speech cloning due to a lack of data and the complexity of Chinese prosody. Future research will concentrate on improving network structure and cloning efficiency. Overall, this paper adds to the literature on speech cloning and suggests promising directions for future research.

[3] Cloning one’s voice using very limited data in the wild

Dongyang Dai, Yuanzhe Chen, Li Chen, et al., [3] This paper proposes Hieratron, a voice cloning model framework composed of bottleneck2mel as well as text2bottleneck parts. Each of the modules Receive instruction on different types Of knowledge, allowing for speaking replicating with only a tiny amount of of poor quality seek speaker the information. Moreover the framework allows for greater control over the synthesized voice, including cross-language and cross-style voice cloning. Overall, the paper improves the total amount of understanding about voice cloning by proposing a new framework who addresses some of the shortcomings of previous approaches.

[4] Natural tts synthesis by conditioning wavenet on mel spectrogram predictions

Jonathan Shen, Ruoming Pang, Ron J. Weiss, et al., [5] This paper describes Tacotron 2, a fully neural TTS system that predicts Mel spectrograms using a series such as sequence connections attention-grabbing network and synthesises speech using a modified WaveNet vocoder. This system gets modern facilities sound reproduction that is comparable to human-like speech and is trained using existing data without having to make use of complex feature engineering methods.

[5] Dian: Duration informed auto-regressive network for voice cloning

Wei Song, Xin Yuan, et al., DIAN, an end-to-end TTS approach for voice cloning that uses a Transformerbased length model and acoustic modelling which requires no consideration, is described in this work. The provided time information is utilised for broadening the sensor's output cycle, removing missing along with recurring issues as well as the attention mechanism across both the decoder and encoder parts. The suggested systems are capable of synthesise expression in conjunction with satisfactory large understanding as well as quality, and reasonable speaker The parallel and style analogy, ranking third with regard to of pronunciation quality and fourth in relation to participant resemblance and along similarities Within the M2VoC Track 1 is a task.

[6] A real-time speaker-dependent neural vocoder

In this paper, the FFTNet is a novel deep learning method for synthesis of audio waveforms that outperforms WaveNet in terms of speed and naturalness of speech when used as a vocoder. WaveNet previously showed the ability to generate excellent audio directly from a convolutional neural network. According to a mean opinion score test, FFTNet outperforms WaveNet in terms of speed and produces more natural-sounding speech.

[7] Recent advances in Google real-time HMM-driven unit selection synthesizer

This paper describes enhancements to Google's HMM-driven choosing units speech production system as well, with emphasis on decreasing latency and boosting pronunciation while handling massive data sets. The paper's authors introduce a number runtime system optimisations, involving a hybrid search approach and a new voice building strategy for effectively dealing with large databases while lowering build times. The improvements are critical for real-world large-scale applications that call for optimal performance along with minimal latency, and therefore.

[8] Deep voice 2: Multi-speaker neural text-to-speech

By applying negligible-dimensional attainable writer embeddings that this paper describes a procedure for providing multiple voices from only one neural TTS (text to speech) model. The authors indicate that Deep Speech 2 and an artificial neural vocoder for post-processing exceed cutting-edge TTS for a single speaker theories, Tacotron as well as Deep Sound 1. They also demonstrate that on one neural TTS structure can acquire several hundred unique voices from multi-speaker TTS information sets. with high sound synthesis and preserved speaker identities in a little over fifty minutes of data per speaker.

Chapter 3

PROPOSED METHODOLOGY

3. PROPOSED METHODOLOGY

3.1 Deep Learning:

Deep learning is part of a broader family of machine learning methods, which is based on artificial neural networks with representation learning. Learning can be supervised, semi-supervised or unsupervised. Deep-learning architectures such as deep neural networks, deep belief networks, deep reinforcement learning, recurrent neural networks, convolutional neural networks and transformers have been applied to fields including computer vision, speech recognition, natural language processing, machine translation, bioinformatics, drug design, medical image analysis, climate science, material inspection and board game programs, where they have produced results comparable to and in some cases surpassing human expert performance.

The adjective "deep" in deep learning refers to the use of multiple layers in the network. Early work showed that a linear perceptron cannot be a universal classifier, but that a network with a nonpolynomial activation function with one hidden layer of unbounded width can. Deep learning is a modern variation that is concerned with an unbounded number of layers of bounded size, which permits practical application and optimized implementation, while retaining theoretical universality under mild conditions. In deep learning the layers are also permitted to be heterogeneous and to deviate widely from biologically informed connectionist models, for the sake of efficiency, trainability and understandability.

Deep learning is a method in artificial intelligence (AI) that teaches computers to process data in a way that is inspired by the human brain. Deep learning models can recognize complex patterns in pictures, text, sounds, and other data to produce accurate insights and predictions. You can use deep learning methods to automate tasks that typically require human intelligence, such as describing images or transcribing a sound file into text.

Artificial intelligence (AI) attempts to train computers to think and learn as humans do. Deep learning technology drives many AI applications used in everyday products,

such as the following:

- Digital assistants
- Voice-activated television remotes
- Fraud detection
- Automatic facial recognition

It is also a critical component of emerging technologies such as self-driving cars, virtual reality, and more. Deep learning models are computer files that data scientists have trained to perform tasks using an algorithm or a predefined set of steps. Businesses use deep learning models to analyze data and make predictions in various applications.

3.2 Python Programming Language:

Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation via the off-side rule. Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly procedural), object oriented and functional programming. It is often described as a "batteries included" language due to its comprehensive standard library. Python is a multi-paradigm programming language. Object-oriented programming and structured programming are fully supported, and many of their features support functional programming and aspect-oriented programming (including metaprogramming and metaobjects). Many other paradigms are supported via extensions, including design by contract and logic programming. Python uses dynamic typing and a combination of reference counting and a cycle-detecting garbage collector for memory management. It uses dynamic name resolution (late binding), which binds method and variable names during program execution. Its design offers some support for functional programming in the Lisp tradition. It has filter, meandrous functions; list comprehensions, dictionaries, sets, and generator expressions. The standard library has two modules (itertools and functools) that implement functional tools borrowed from Haskell and Standard ML.

3.3 PyTorch:

PyTorch is an open-source machine learning framework designed to accelerate the path from research prototyping to production deployment. PyTorch was created to provide flexibility and speed during the development and implementation of deep learning neural networks. Examples of deep learning software built on top of PyTorch include Tesla's Autopilot, Uber's Pyro, Hugging Face's Transformers, PyTorch Lightning, and Catalyst. PyTorch is an optimized tensor library for deep learning that uses GPUs and CPUs to greatly accelerate computation speed. It is a Python-based package that provides two high-level features: tensor computation (like NumPy) with strong GPU acceleration and deep neural networks built on a tape-based autograd system. PyTorch provides a wide variety of tensor routines to accelerate and fit scientific computation needs, such as slicing, indexing, mathematical operations, linear algebra, and reductions. PyTorch is an open-source machine learning framework designed to accelerate the path from research prototyping to production deployment. PyTorch was created to provide flexibility and speed during the development and implementation of deep learning neural networks. Examples of deep learning software built on top of PyTorch include Tesla's Autopilot, Uber's Pyro, HuggingFace's Transformers, PyTorch Lightning, and Catalyst.

3.4 Statistical parametric speech synthesis:

Statistical parametric speech synthesis (SPSS) refers to a group of data-driven TTS methods that emerged in the late 90s. In SPSS, the relationship between the features computed on the input text and the output acoustic features is learned by a statistical generative model (called the acoustic model). A complete SPSS framework thus also includes a pipeline to extract features from the text to synthesize, as well as a system able to reconstruct an audio waveform from the acoustic features produced by the acoustic model (such a system is called a vocoder). Unlike the acoustic model, these two parts of the framework may be entirely engineered and make use of no statistical methods. While modern deep TTS models are usually not referred to as SPSS, the SPSS pipeline as depicted in Figure 3.4.1 applies just as well to those newer methods.

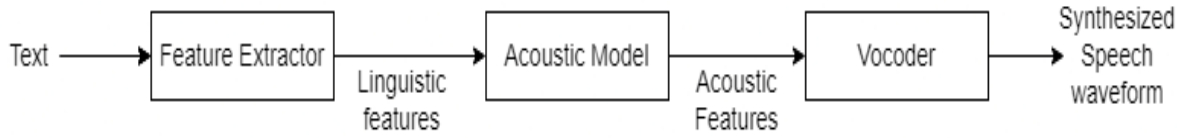


Figure 3.4.1: The general SPSS pipeline

The role of the feature extractor is to provide data that is more indicative of what the speech produced by the model is expected to sound like. Speech is a complex process, and directly feeding characters to a weak acoustic model will prove not to be effective. Providing additional features from natural language processing (NLP) techniques may greatly reduce the extent of the task to be learned by the acoustic model. It may however result in trade-offs when it comes to naturalness, especially for rare or unknown words. Indeed, manually engineered heuristics do not quite fully characterize all intricacies of spoken language. For this reason, feature extraction can also be done with trained models. The line between the feature extractor and the acoustic model can then become blurry, especially for deep models. In fact, a tendency that is common across all areas where deep models have overtaken traditional machine learning techniques is for feature extraction to consist of less heuristics, as highly nonlinear models become able to operate at higher levels of abstraction.

A common feature extraction technique is to build frames that will integrate surrounding context in a hierarchical fashion. For example, a frame at the syllable level could include the word that comprises it, its position in the word,

The reason why the acoustic model does not directly predict an audio waveform is that audio happens to be difficult to model: it is a particularly dense domain and audio signals are typically highly nonlinear. A representation that brings out features in a more tractable manner is the time-frequency domain. Spectrograms are smoother and much less dense than their waveform counterpart. They also have the benefit of being two-dimensional, thus allowing models to better leverage spatial connectivity. Unfortunately, a spectrogram is a lossy representation of the waveform that discards the phase. There is no unique inverse transformation function, and deriving one that produces natural-sounding results is not trivial. When referring to speech, this generative function is called a vocoder. The choice of the vocoder is an important factor in determining the quality of the generated audio.

3.5 Evolution of the state of the art in text-to-speech:

The state of the art in SPSS has for long remained a hidden Markov model (HMM) based framework. This approach, laid out in Figure 3.5.1, consists in clustering the linguistic features extracted from the input text with a decision tree, and to train a HMM per cluster. The HMMs are tasked to produce a distribution over spectrogram coefficients, their derivative, second derivative and a binary flag that indicates which parts of the generated audio should contain voice. With the maximum likelihood parameter generation algorithm (MLPG) , spectrogram coefficients are sampled from this distribution and eventually fed to the MLSA vocoder [4]. It is possible to modify the voice generated by conditioning the HMMs on a speaker or tuning the generated speech parameters with adaptation or interpolation techniques Note that, while this framework used to be state of the art for SPSS, it was still inferior in terms of the naturalness of the generated speech compared to the well-established concatenative approaches.

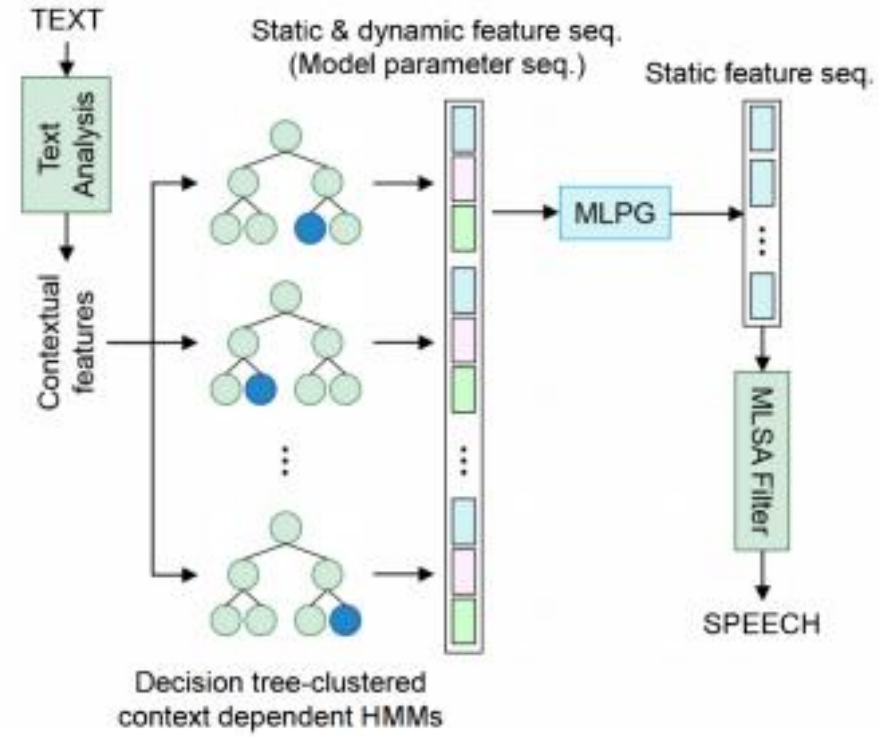


Figure 3.5.1: The general HMM-based TTS pipeline.

Chapter 4

PROPOSED ALGORITHM

4. PROPOSED ALGORITHM

4.1 Transfer Learning from Speaker Verification to Multispeaker Text-To-Speech Synthesis:

Our approach to real-time voice cloning is largely based on SV2TTS throughout this document. It describes a framework for zero-shot voice cloning that only requires 5 seconds of reference speech. This paper is only one of the many publications from the Tacotron series5 authored at Google. Interestingly, the SV2TTS paper does not bring much innovation of its own, rather it is based on three major earlier works from Google: the GE2E loss, Tacotron and WaveNet. The complete framework is a three-stage pipeline, where the steps correspond to the models listed in order previously. Many of the current TTS tools and functionalities provided by Google, such as the Google assistant6 or the Google cloud services, make use of these same models [1]. While there are many open-source reimplementation of these models online, there is none of the SV2TTS framework to our knowledge. The three stages of the framework are as follows:

- A speaker encoder that derives an embedding from the short utterance of a single speaker. The embedding is a meaningful representation of the voice of the speaker, such that similar voices are close in latent space.
- A synthesizer that, conditioned on the embedding of a speaker, generates a spectrogram from text. This model is the popular Tacotron 2 without WaveNet (which is often referred to as just Tacotron due to its similarity to the first iteration).
- A vocoder that infers an audio waveform from the spectrograms generated by the synthesizer. The authors used WaveNet as a vocoder, effectively reapplying the entire Tacotron 2 framework.

At inference time, the speaker encoder is fed a short reference utterance of the speaker to clone. generates an embedding that is used to condition the synthesizer, and a text processed as a phoneme sequence is given as input to the synthesizer. The vocoder It takes the output of the synthesizer to generate the speech waveform. This is illustrated in Figure 1.2.2.1.

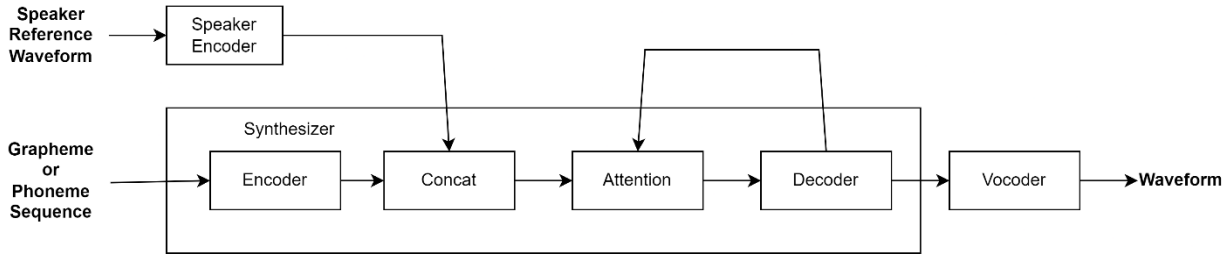


Figure 4.1.1: The SV2TTS framework during inference.

All models in the SV2TTS framework can be trained on different datasets independently. It is critical to have a noise resistant encoder capable of capturing the many characteristics of human speech. As a result, an extensive corpus of multiple speaking would be ideal to instruct the encoder, despite the strict audio quality requirements. Furthermore, the digital encoder is trained with the GE2E loss, which requires only the speaker's name as labels. The model in GE2E learns from a speaker verification task that has nothing connected with conversation cloning purposes However, the task demands that the network create an embedding that represents the speaker's voice in an important manner [1].

As shown in Figure 4.1.1, the system is divided into three modules: encoder, synthesizer, and vocoder. In this research, the speaker's voice is first converted into the speaker's reference waveform, which then gets sent to the speaker encoder. In order to transform the speaker's voice into speaker embedding, i.e. text, we utilize a Speaker Encoder, which is trained using numerous distinct speakers. Also, the encoder has received GE2E loss training, and this is used for speaker validation tasks, as a part of working with the Speaker Encoder. The project specifies the method for incorporating results. At every step of the training process, the GE2E loss function updates the network in a way that emphasizes examples that are tricky to verify. A Phoneme is the smallest sound in a spoken word . Graphene is a written symbol (letter or letters) that represents a sound. So simply, the phoneme-grapheme connection is the relationship between sounds and letters. e.g. ‘x’ which is the speech sounds k/s together, and that is made up of k/w.For utilizing the ‘Text to Speech’ model here we use a Graphene or Phoneme sequence by which we are able to distinguish the speaker’s voice letters i.e. Graphene into small sound words i.e. Phoneme. After distinguishing the sound words, the Graphene or Phoneme sequence is passed on to

Encoder which converts voice into speaker embedding i.e text. After obtaining the embedding i.e., text, Synthesizer plays an important role which is used for converting the text into Mel-Spectrogram. The Tacotron 2 modular synthesizer is now optimized. In Tacotron 2, replace Wavenet with a changed network. To expand the span of a single encoder frame, every single letter in the spoken a series is first ingrained as a vector of values followed by overlaid. At the same time, enter the phoneme sequence, which will quickly combine and improve speaking.

To produce the device output frames, these encoders frames are transmitted via in both directions LSTM.the LSTM algorithm is a code word for long short-term recall. It is a constant brain network (RNN) memory extension model or architecture. RNN is an artificial neural network that works on the current input through taking into account the previous output (feedback) and storing it in its memory to stay a short amount of time (short-term memory). The system's a combination aspect includes the embedding of a speaker waveform sequence and an a grapheme or phoneme sequence. To generate the decoder's input frame, the mind's mechanism now focuses on the encoder output each frame. The model is autoregressive model because each decoder participation frame is linked to the previous decoding device frame result.This cascading vector is projected onto an entire MEL spectrum frame once it traverses two one-way LSTM layer layers. When an additional projection prediction network of the same vectors into a scalar, it emits a value that is higher than the threshold, the frame generation is slowed down. Before becoming a MEL spectrogram, every single pair of frames undergoes transmission by way of an additional network technology.

The Algorithm follows the steps as:

1. For converting the speaker’s voice into speaker embedding i.e text we use a Speaker Encoder which is composed of many different speakers to train the encoder.
2. Because of the Speaker Encoder, In addition, the encoder has received GE2E loss training which is used for speaker verification tasks. The task specifies the way of output embedding.
3. GE2E loss function updates the network in a way that emphasizes examples that are difficult to verify at each step of the training process.
4. A Phoneme is the smallest sound in a spoken word. Graphene is a written symbol (letter or letters) that represents a sound. So simply, the phoneme-

grapheme connection is the relationship between sounds and letters. e.g. ‘x’ which is the speech sounds k/s together, and qu that is made up of k/w.

5. For utilizing the ‘Text to Speech’ model here we use a Graphene or Phoneme sequence by which we are able to distinguish the speaker’s voice letters i.e. Graphene into small sound words i.e. Phoneme.
6. After distinguishing the sound words, the Graphene or Phoneme sequence is passed on to Encoder which converts voice into speaker embedding i.e text.
7. After obtaining the embedding i.e. text, Synthesizer plays an important role which is used for converting the text into Mel-Spectrogram.
8. The synthesizer is based on optimized Tacotron 2. Replace Wavenet in Tacotron 2 with a modified network. Each character in the text sequence is first embedded as a vector and then convolved to increase the span of a single encoder frame.
9. Meanwhile, input the corresponding phoneme sequence, which can quickly converge and improve pronunciation. These encoder frames are transmitted via bidirectional LSTM to produce encoder output frames.
10. LSTM stands for long short-term memory. It is a model or architecture that extends the memory of recurrent neural networks (RNN). Talking about RNN, it is a network that works on the present input by taking into consideration the previous output (feedback) and storing in its memory for a short period of time (short-term memory).
11. The concat part of the system add embedding of speaker waveform sequence and Grapheme or Phoneme Sequence.
12. Now, The attention mechanism focuses on the encoder output frame to generate the decoder input frame.
13. Each decoder input frame is connected to the previous decoder frame output, thus making the model autoregressive.
14. This cascading vector passes through two one-way LSTM layers before being projected onto a single MEL spectrum frame.
15. Another projection prediction network of the same vector into a scalar emits a value above a certain threshold to stop frame generation.
16. The entire sequence of frames passes through a residual network before becoming a MEL spectrogram.
17. The target MEL spectrogram of the synthesizer has more acoustic

characteristics than the speaker encoder. They are calculated in 12.5ms steps from a 50ms window and fed into 80-dimensional MFCC.

18. MFCCs are a compact representation of the spectrum (When a waveform is represented by a summation of the possibly infinite number of sinusoids) of an audio signal.
19. The Vocoder is used for mel-spectrogram into waveform.
20. The vocoder is used for speech synthesis and speech is generated from it by using the LPCNET model improved by Wavernn.
21. LPCNET was proposed as a way to reduce the complexity of neural synthesis by using linear prediction (LP) to assist an autoregressive model.
22. After performing vocoder actions, it generates the waveform.
23. For the conversion of text-to-speech, the method of J. Shen will be used which is mainly based on Tacotron 2 is a neural network architecture for speech synthesis directly from the text.

In this way the algorithm follows the steps.

Chapter 5

DESIGN AND IMPLEMENTATION

5. DESIGN AND IMPLEMENTATION

5.1 Design:

Consider a dataset of utterances grouped by their speaker. We denote the j^{th} utterance of the i^{th} speaker as u_{ij}. Utterances are in the waveform domain. We denote by x_{ij} the log-mel spectrogram of the utterance uij. A log-mel spectrogram is a deterministic, non-invertible (lossy) function that extracts speech features from a waveform, so as to handle speech in a more tractable fashion in machine learning.

The encoder E computes the embedding $e_{ij} = E(x_{ij};wE)$ corresponding to the utterance u_{ij}, where wE are the parameters of the encoder. Additionally, the authors define a speaker embedding as the centroid of the embeddings of the speaker's utterances:

$$\mathbf{c}_i = \frac{1}{n}\sum_{j=1}^{n}\mathbf{e}_{ij}$$

The synthesizer S, parametrized by wS, is tasked to approximate xij given ci and tij, the transcript of utterance uij. We have $\mathbf{x}\hat{}_{ij} = S(\mathbf{c}_i,\mathbf{t}_{ij};\mathbf{w}_S)$. In our implementation, we directly use the utterance embedding rather than the speaker embedding (we motivate this choice in section 3.4), giving instead $\mathbf{x}\hat{}_{ij} = S(\mathbf{u}_{ij},\mathbf{t}_{ij};\mathbf{w}_S)$[2].

Finally, the vocoder V, parametrized by wV, is tasked to approximate uij given xˆij. We have $\mathbf{u}\hat{}_{ij} = V(\mathbf{x}\hat{}_{ij};\mathbf{w}_V)$.

One could train this framework in an end-to-end fashion with the following objective function [2]:

$$min_{\mathbf{w}_E,\mathbf{w}_S,\mathbf{w}_V} L_V(\mathbf{u}_{ij}, V(S(E(\mathbf{x}_{ij};\mathbf{w}_E),\mathbf{t}_{ij};\mathbf{w}_S);\mathbf{w}_V))$$

Where LV is a loss function in the waveform domain. This approach has drawbacks:

- It requires training all three models on a same dataset, meaning that this dataset would ideally need to meet the requirements for all models: a large number of speakers for the encoder but at the same time, transcripts for the synthesizer. A low-level noise for the synthesizer and somehow an average noise level for the encoder (so as to be able to handle noisy input speech). These conflicts are problematic and would lead to training models that could perform better if trained separately on distinct datasets. Specifically, a small dataset will likely lead to poor generalization and thus poor zero-shot performance.
- The convergence of the combined model could be very hard to reach. In

particular, the Tacotron synthesizer could take a significant time before producing correct alignments.

An evident way of addressing the second issue is to separate the training of the synthesizer and of the vocoder. Assuming a pretrained encoder, the synthesizer can be trained to directly predict the mel spectrograms of the target audio:

$$min_{\mathbf{w}_S} L_S(\mathbf{x}_{ij}, S(\mathbf{e}_{ij}, \mathbf{t}_{ij}; \mathbf{w}_S))$$

Remains the optimization of the speaker encoder. Unlike the synthesizer and the vocoder, the encoder does not have labels to be trained on. The task is lousily defined as producing "meaningful" embeddings that characterize the voice in the utterance. One could conceive of a way to train the speaker encoder as an autoencoder, but it would require the corresponding up sampling model to be made aware of the text to predict. Either the dataset is constrained to a same sentence, either one needs transcripts and the up-sampling model is the synthesizer. In both cases the quality of the training is impaired by the dataset and unlikely to generalize well. Fortunately, the GE2E loss brings a solution to this problem and allows to train the speaker encoder independently of the synthesizer.

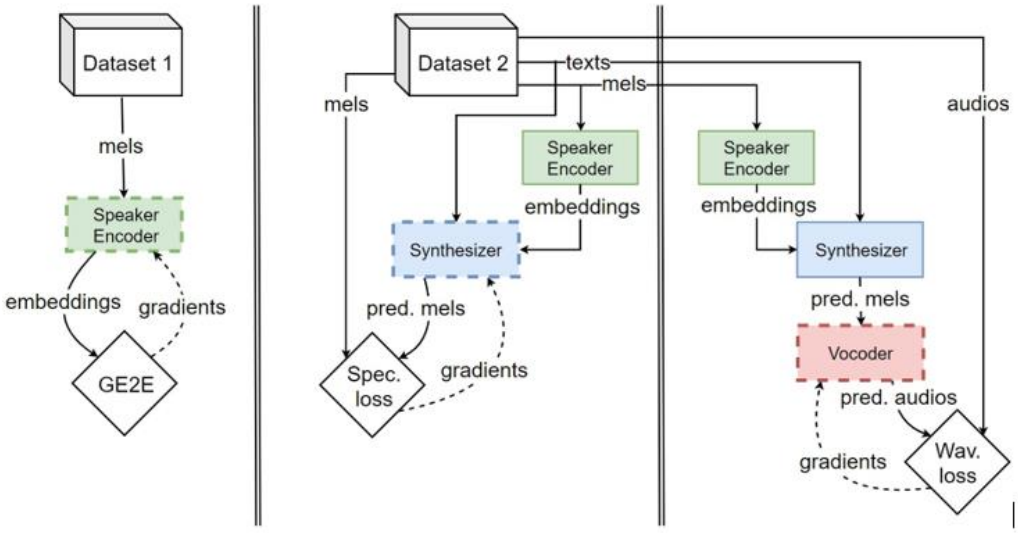


Figure 5.1.1: The sequential three-stage training of SV2TTS

While all parts of the framework are trained separately, there is still the requirement for the synthesizer to have embeddings from a trained encoder and for the vocoder to have mel spectrograms from a trained synthesizer (if not training on ground truth spectrogram). Figure 1.2.1.2 illustrates how each model depends on the previous one

for training. The speaker encoder needs to generalize well enough to produce meaningful embeddings on the dataset of the synthesizer; and even when trained on a common dataset, it still has to be able to operate in a zero-shot setting at inference time.

5.2 Speaker encoder:

The encoder model and its training procedure are described over several papers,we reproduced this model with a PyTorch implementation of our own. We synthesize the parts that are pertinent to SV2TTS as well as our choices of implementation.

5.2.1 Model architecture:

The model is a 3-layer LSTM with 768 hidden nodes followed by a projection layer of 256 units. While there is no reference in any of the papers as to what a projection layer is, our intuition is that it is simply a 256 outputs fully-connected layer per LSTM that is repeatedly applied to every output of the LSTM. When we first implemented the speaker encoder, we directly used 256 units LSTM layers instead, for the sake of quick prototyping, simplicity and for a lighter training load. This last part is important, as the authors have trained their own model for 50 million steps (although on a larger dataset), which is technically difficult for us to reproduce. We found this smaller model to perform extremely well, and we haven't found the time to train the larger version later on [2].

The inputs to the model are 40-channels log-mel spectrograms with a 25ms window width and a 10ms step. The output is the L2-normalized hidden state of the last layer, which is a vector of 256 elements. Our implementation also features a ReLU layer before the normalization, with the goal in mind to make embeddings sparse and thus more easily interpretable.

5.2.2 Generalized End-to-End loss:

The speaker encoder is trained on a speaker verification task. Speaker verification is a typical application of biometrics where the identity of a person is verified through their voice. A template is created for a person by deriving their speaker embedding

(see equation 1) from a few utterances. This process is called enrollment. At runtime, a user identifies himself with a short utterance and the system compares the embedding of that utterance with the enrolled speaker embeddings. Above a given similarity threshold, the user is identified. The GE2E loss simulates this process to optimize the model.

At training time, the model computes the embeddings eij ($1 \le i \le N, 1 \le j \le M$) of M utterances of fixed duration from N speakers. A speaker embedding ci is derived for each speaker: $\mathbf{c}_i = \frac{1}{M}\sum_{j=1}^{M} \mathbf{e}_{ij}$. The similarity matrix $\mathbf{S}_{ij,}$k is the result of the two-by-two comparison of all embeddings $\mathbf{e}_{ij}$ against every speaker embedding $\mathbf{c}_k$ ($1 \le k \le N$) in the batch[2]. This measure is the scaled cosine similarity:

$$\mathbf{S}_{ij,k} = w \cdot cos(\mathbf{e}_{ij}, \mathbf{c}_k) + b = w \cdot \mathbf{e}_{ij} \cdot \|\mathbf{c}_k\|_2 + b$$

where w and b are learnable parameters. This entire process is illustrated in Figure 5.3.1.1. From a computing perspective, the cosine similarity of two L2-normed vectors is simply their dot product, hence the rightmost hand side of equation 2. An optimal model is expected to output high similarity values when an utterance matches the speaker (i = k) and lower values elsewhere (i 6= k). To optimize in this direction, the loss is the sum of row-wise SoftMax losses [2].

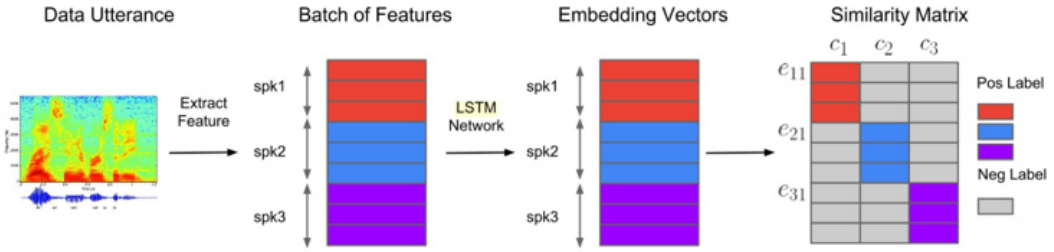


Figure 5.2.2.1: The construction of the similarity matrix at training time

Note that each utterance eij is included in the centroid ci of the same speaker when computing the loss. This creates a bias towards the correct speaker independently of the accuracy of the model; and the authors argue that it also leaves room for trivial solutions. To prevent this, an utterance that is compared against its own speaker's embedding will be removed from the speaker embedding. The similarity matrix is then defined as:

$$w \cdot \cos\left(\mathbf{e}_{ij}, \mathbf{c}_i^{(-j)}\right) + b$$

if i = k

Sji,k = w · cos(eij,ck) + b otherwise. Where the exclusive centroids $\mathbf{c}_i^{(-j)}$ are

defined as:

$$\mathbf{c}_i^{(-j)} = \frac{1}{M-1} \sum_{\substack{m=1 \\ m \neq j}}^{M} \mathbf{e}_{im}$$

The fixed duration of the utterances in a training batch is of 1.6 seconds. These are partial utterances sampled from the longer complete utterances in the dataset. While the model architecture is able to handle inputs of variable length, it is reasonable to expect that it performs best with utterances of the same duration as those seen in training. Therefore, at inference time an utterance is split in segments of 1.6 seconds overlapping by 50%, and the encoder forwards each segment individually. The resulting outputs are averaged then normalized to produce the utterance embedding [2]. This is illustrated in figure 5.2.2.2. Curiously, the authors of SV2TTS advocate for 800ms windows at inference time but still 1.6 seconds ones during training. We prefer to keep 1.6 seconds for both, as is done in GE2E.

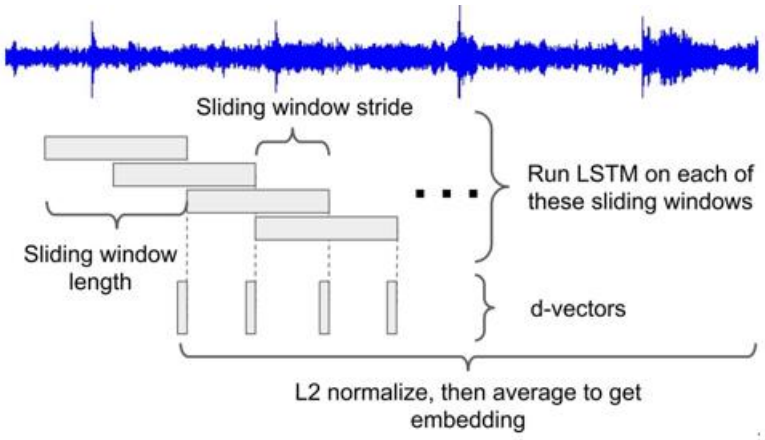


Figure 5.2.2.2: Computing the embedding of a complete utterance

As for the number of speakers, it's important to consider the time complexity of computing the similarity matrix, which is O(N2M). Therefore, the number of speakers should not be excessively large to avoid significantly slowing down the training process, as opposed to simply selecting the largest batch size that fits on the GPU. However, it is still possible to parallelize multiple batches on the same GPU while synchronizing operations across batches for efficiency. It was found to be crucial to vectorize all operations when computing the similarity matrix in order to minimize the number of GPU transactions.

5.2.3 Implementation:

- Applies the preprocessing operations used in training the Speaker Encoder to a waveform either on disk or in memory. The waveform will be resampled to match the data hyperparameters.

r > audio.py > ...

```
from scipy.ndimage.morphology import binary_dilation
from encoder.params_data import *
from pathlib import Path
from typing import Optional, Union
from warnings import warn
import numpy as np
import librosa
import struct

try:
    import webrtcvad
except:
    warn("Unable to import 'webrtcvad'. This package enables noise removal and is recommended.")
    webrtcvad=None

int16_max = (2 ** 15) - 1


def preprocess_wav(fpath_or_wav: Union[str, Path, np.ndarray],
                   source_sr: Optional[int] = None,
                   normalize: Optional[bool] = True,
                   trim_silence: Optional[bool] = True):
    """
    Applies the preprocessing operations used in training the Speaker Encoder to a waveform
    either on disk or in memory. The waveform will be resampled to match the data hyperparameters.

    :param fpath_or_wav: either a filepath to an audio file (many extensions are supported, not
    just .wav), either the waveform as a numpy array of floats.
    :param source_sr: if passing an audio waveform, the sampling rate of the waveform before
    preprocessing. After preprocessing, the waveform's sampling rate will match the data
    hyperparameters. If passing a filepath, the sampling rate will be automatically detected and
    this argument will be ignored.
    """
    # Load the wav from disk if needed
    if isinstance(fpath_or_wav, str) or isinstance(fpath_or_wav, Path):
        wav, source_sr = librosa.load(str(fpath_or_wav), sr=None)
    else:
        wav = fpath_or_wav

    # Resample the wav if needed
    if source_sr is not None and source_sr != sampling_rate:
        wav = librosa.resample(wav, source_sr, sampling_rate)

    # Apply the preprocessing: normalize volume and shorten long silences
    if normalize:
        wav = normalize_volume(wav, audio_norm_target_dBFS, increase_only=True)
    if webrtcvad and trim_silence:
        wav = trim_long_silences(wav)

    return wav
```

audio.py > ...

```python
def wav_to_mel_spectrogram(wav):
    """
    Derives a mel spectrogram ready to be used by the encoder from a preprocessed audio waveform.
    Note: this not a log-mel spectrogram.
    """
    frames = librosa.feature.melspectrogram(
        wav,
        sampling_rate,
        n_fft=int(sampling_rate * mel_window_length / 1000),
        hop_length=int(sampling_rate * mel_window_step / 1000),
        n_mels=mel_n_channels
    )
    return frames.astype(np.float32).T


def trim_long_silences(wav):
    """
    Ensures that segments without voice in the waveform remain no longer than a
    threshold determined by the VAD parameters in params.py.

    :param wav: the raw waveform as a numpy array of floats
    :return: the same waveform with silences trimmed away (length ⩽ original wav length)
    """
    # Compute the voice detection window size
    samples_per_window = (vad_window_length * sampling_rate) // 1000

    # Trim the end of the audio to have a multiple of the window size
    wav = wav[:len(wav) - (len(wav) % samples_per_window)]

    # Convert the float waveform to 16-bit mono PCM
    pcm_wave = struct.pack("%dh" % len(wav), *(np.round(wav * int16_max)).astype(np.int16))

    # Perform voice activation detection
    voice_flags = []
    vad = webrtcvad.Vad(mode=3)
    for window_start in range(0, len(wav), samples_per_window):
        window_end = window_start + samples_per_window
        voice_flags.append(vad.is_speech(pcm_wave[window_start * 2:window_end * 2],
                                         sample_rate=sampling_rate))
    voice_flags = np.array(voice_flags)

    # Smooth the voice detection with a moving average
    def moving_average(array, width):
        array_padded = np.concatenate((np.zeros((width - 1) // 2), array, np.zeros(width // 2)))
        ret = np.cumsum(array_padded, dtype=float)
        ret[width:] = ret[width:] - ret[:-width]
        return ret[width - 1:] / width

    audio_mask = moving_average(voice_flags, vad_moving_average_width)
    audio_mask = np.round(audio_mask).astype(np.bool)


    # Dilate the voiced regions
    audio_mask = binary_dilation(audio_mask, np.ones(vad_max_silence_length + 1))
    audio_mask = np.repeat(audio_mask, samples_per_window)

    return wav[audio_mask == True]


def normalize_volume(wav, target_dBFS, increase_only=False, decrease_only=False):
    if increase_only and decrease_only:
        raise ValueError("Both increase only and decrease only are set")
    dBFS_change = target_dBFS - 10 * np.log10(np.mean(wav ** 2))
    if (dBFS_change < 0 and increase_only) or (dBFS_change > 0 and decrease_only):
        return wav
    return wav * (10 ** (dBFS_change / 20))
```

- At inference time, the speaker encoder is fed a short reference utterance of the speaker to clone. It generates an embedding that is used to condition the synthesizer, and a text processed as a phoneme sequence is given as input to the synthesizer.

r > inference.py > ...

```python
from encoder.params_data import *
from encoder.model import SpeakerEncoder
from encoder.audio import preprocess_wav   # We want to expose this function from here
from matplotlib import cm
from encoder import audio
from pathlib import Path
import numpy as np
import torch

_model = None # type: SpeakerEncoder
_device = None # type: torch.device


def load_model(weights_fpath: Path, device=None):
    """
    Loads the model in memory. If this function is not explicitely called, it will be run on the
    first call to embed_frames() with the default weights file.

    :param weights_fpath: the path to saved model weights.
    :param device: either a torch device or the name of a torch device (e.g. "cpu", "cuda"). The
    model will be loaded and will run on this device. Outputs will however always be on the cpu.
    If None, will default to your GPU if it"s available, otherwise your CPU.
    """
    # TODO: I think the slow loading of the encoder might have something to do with the device it
    #   was saved on. Worth investigating.
    global _model, _device
    if device is None:
        _device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    elif isinstance(device, str):
        _device = torch.device(device)
    _model = SpeakerEncoder(_device, torch.device("cpu"))
    checkpoint = torch.load(weights_fpath, _device)
    _model.load_state_dict(checkpoint["model_state"])
    _model.eval()
    print("Loaded encoder \"%s\" trained to step %d" % (weights_fpath.name, checkpoint["step"]))


def is_loaded():
    return _model is not None



def embed_frames_batch(frames_batch):
    """
    Computes embeddings for a batch of mel spectrogram.

    :param frames_batch: a batch mel of spectrogram as a numpy array of float32 of shape
    (batch_size, n_frames, n_channels)
    :return: the embeddings as a numpy array of float32 of shape (batch_size, model_embedding_size)
    """
    if _model is None:
        raise Exception("Model was not loaded. Call load_model() before inference.")

    frames = torch.from_numpy(frames_batch).to(_device)
    embed = _model.forward(frames).detach().cpu().numpy()
    return embed
```

- Computes where to split an utterance waveform and its corresponding mel spectrogram to obtain partial utterances of <partial_utterance_n_frames> each. Both the waveform and the mel spectrogram slices are returned, so as to make each partial utterance waveform correspond to its spectrogram

inference.py > ...

```
def compute_partial_slices(n_samples, partial_utterance_n_frames=partials_n_frames,
                           min_pad_coverage=0.75, overlap=0.5):
    """
    Computes where to split an utterance waveform and its corresponding mel spectrogram to obtain
    partial utterances of <partial_utterance_n_frames> each. Both the waveform and the mel
    spectrogram slices are returned, so as to make each partial utterance waveform correspond to
    its spectrogram. This function assumes that the mel spectrogram parameters used are those
    defined in params_data.py.

    The returned ranges may be indexing further than the length of the waveform. It is
    recommended that you pad the waveform with zeros up to wave_slices[-1].stop.

    :param n_samples: the number of samples in the waveform
    :param partial_utterance_n_frames: the number of mel spectrogram frames in each partial
    utterance
    :param min_pad_coverage: when reaching the last partial utterance, it may or may not have
    enough frames. If at least <min_pad_coverage> of <partial_utterance_n_frames> are present,
    then the last partial utterance will be considered, as if we padded the audio. Otherwise,
    it will be discarded, as if we trimmed the audio. If there aren't enough frames for 1 partial
    utterance, this parameter is ignored so that the function always returns at least 1 slice.
    :param overlap: by how much the partial utterance should overlap. If set to 0, the partial
    utterances are entirely disjoint.
    :return: the waveform slices and mel spectrogram slices as lists of array slices. Index
    respectively the waveform and the mel spectrogram with these slices to obtain the partial
    utterances.
    """
    assert 0 <= overlap < 1
    assert 0 < min_pad_coverage <= 1

    samples_per_frame = int((sampling_rate * mel_window_step / 1000))
    n_frames = int(np.ceil((n_samples + 1) / samples_per_frame))
    frame_step = max(int(np.round(partial_utterance_n_frames * (1 - overlap))), 1)

    # Compute the slices
    wav_slices, mel_slices = [], []
    steps = max(1, n_frames - partial_utterance_n_frames + frame_step + 1)
    for i in range(0, steps, frame_step):
        mel_range = np.array([i, i + partial_utterance_n_frames])
        wav_range = mel_range * samples_per_frame
        mel_slices.append(slice(*mel_range))
        wav_slices.append(slice(*wav_range))

    # Evaluate whether extra padding is warranted or not
    last_wav_range = wav_slices[-1]
    coverage = (n_samples - last_wav_range.start) / (last_wav_range.stop - last_wav_range.start)
    if coverage < min_pad_coverage and len(mel_slices) > 1:
        mel_slices = mel_slices[:-1]
        wav_slices = wav_slices[:-1]

    return wav_slices, mel_slices
```

- Creating a model for speaker encoder which Computes the embeddings of a batch of utterance spectrograms.

er > model.py > SpeakerEncoder > forward

```
from encoder.params_model import *
from encoder.params_data import *
from scipy.interpolate import interp1d
from sklearn.metrics import roc_curve
from torch.nn.utils import clip_grad_norm_
from scipy.optimize import brentq
from torch import nn
import numpy as np
import torch


class SpeakerEncoder(nn.Module):
    def __init__(self, device, loss_device):
        super().__init__()
        self.loss_device = loss_device

        # Network defition
        self.lstm = nn.LSTM(input_size=mel_n_channels,
                            hidden_size=model_hidden_size,
                            num_layers=model_num_layers,
                            batch_first=True).to(device)
        self.linear = nn.Linear(in_features=model_hidden_size,
                                out_features=model_embedding_size).to(device)
        self.relu = torch.nn.ReLU().to(device)

        # Cosine similarity scaling (with fixed initial parameter values)
        self.similarity_weight = nn.Parameter(torch.tensor([10.])).to(loss_device)
        self.similarity_bias = nn.Parameter(torch.tensor([-5.])).to(loss_device)

        # Loss
        self.loss_fn = nn.CrossEntropyLoss().to(loss_device)

    def do_gradient_ops(self):
        # Gradient scale
        self.similarity_weight.grad *= 0.01
        self.similarity_bias.grad *= 0.01

        # Gradient clipping
        clip_grad_norm_(self.parameters(), 3, norm_type=2)

    def forward(self, utterances, hidden_init=None):
        """
        Computes the embeddings of a batch of utterance spectrograms.

        :param utterances: batch of mel-scale filterbanks of same duration as a tensor of shape
        (batch_size, n_frames, n_channels)
        :param hidden_init: initial hidden state of the LSTM as a tensor of shape (num_layers,
        batch_size, hidden_size). Will default to a tensor of zeros if None.
        :return: the embeddings as a tensor of shape (batch_size, embedding_size)
        """
        # Pass the input through the LSTM layers and retrieve all outputs, the final hidden state
        # and the final cell state.
        out, (hidden, cell) = self.lstm(utterances, hidden_init)

        # We take only the hidden state of the last layer
        embeds_raw = self.relu(self.linear(hidden[-1]))

        # L2-normalize it
        embeds = embeds_raw / (torch.norm(embeds_raw, dim=1, keepdim=True) + 1e-5)

        return embeds
```

- Preprocess the Dataset and apply on model for training.

preprocess.py > ...

```python
from datetime import datetime
from functools import partial
from multiprocessing import Pool
from pathlib import Path

import numpy as np
from tqdm import tqdm

from encoder import audio
from encoder.config import librispeech_datasets, anglophone_nationalites
from encoder.params_data import *


_AUDIO_EXTENSIONS = ("wav", "flac", "m4a", "mp3")

class DatasetLog:
    """
    Registers metadata about the dataset in a text file.
    """
    def __init__(self, root, name):
        self.text_file = open(Path(root, "Log_%s.txt" % name.replace("/", "_")), "w")
        self.sample_data = dict()

        start_time = str(datetime.now().strftime("%A %d %B %Y at %H:%M"))
        self.write_line("Creating dataset %s on %s" % (name, start_time))
        self.write_line("-----")
        self._log_params()

    def _log_params(self):
        from encoder import params_data
        self.write_line("Parameter values:")
        for param_name in (p for p in dir(params_data) if not p.startswith("__")):
            value = getattr(params_data, param_name)
            self.write_line("\t%s: %s" % (param_name, value))
        self.write_line("-----")

    def write_line(self, line):
        self.text_file.write("%s\n" % line)

    def add_sample(self, **kwargs):
        for param_name, value in kwargs.items():
            if not param_name in self.sample_data:
                self.sample_data[param_name] = []
            self.sample_data[param_name].append(value)
```

r > preprocess.py > ...

```python
def _preprocess_speaker(speaker_dir: Path, datasets_root: Path, out_dir: Path, skip_existing: bool):
    # Give a name to the speaker that includes its dataset
    speaker_name = "_".join(speaker_dir.relative_to(datasets_root).parts)

    # Create an output directory with that name, as well as a txt file containing a
    # reference to each source file.
    speaker_out_dir = out_dir.joinpath(speaker_name)
    speaker_out_dir.mkdir(exist_ok=True)
    sources_fpath = speaker_out_dir.joinpath("_sources.txt")

    # There's a possibility that the preprocessing was interrupted earlier, check if
    # there already is a sources file.
    if sources_fpath.exists():
        try:
            with sources_fpath.open("r") as sources_file:
                existing_fnames = {line.split(",")[0] for line in sources_file}
        except:
            existing_fnames = {}
    else:
        existing_fnames = {}

    # Gather all audio files for that speaker recursively
    sources_file = sources_fpath.open("a" if skip_existing else "w")
    audio_durs = []
    for extension in _AUDIO_EXTENSIONS:
        for in_fpath in speaker_dir.glob("**/*.%s" % extension):
            # Check if the target output file already exists
            out_fname = "_".join(in_fpath.relative_to(speaker_dir).parts)
            out_fname = out_fname.replace(".%s" % extension, ".npy")
            if skip_existing and out_fname in existing_fnames:
                continue

            # Load and preprocess the waveform
            wav = audio.preprocess_wav(in_fpath)
            if len(wav) == 0:
                continue

            # Create the mel spectrogram, discard those that are too short
            frames = audio.wav_to_mel_spectrogram(wav)
            if len(frames) < partials_n_frames:
                continue

            out_fpath = speaker_out_dir.joinpath(out_fname)
            np.save(out_fpath, frames)
            sources_file.write("%s,%s\n" % (out_fname, in_fpath))
            audio_durs.append(len(wav) / sampling_rate)

    sources_file.close()

    return audio_durs
```

- Train the speaker encoder model on the preprocess dataset.

```
ler > train.py > ...
from pathlib import Path

import torch

from encoder.data_objects import SpeakerVerificationDataLoader, SpeakerVerificationDataset
from encoder.model import SpeakerEncoder
from encoder.params_model import *
from encoder.visualizations import Visualizations
from utils.profiler import Profiler


def sync(device: torch.device):
    # For correct profiling (cuda operations are async)
    if device.type == "cuda":
        torch.cuda.synchronize(device)


def train(run_id: str, clean_data_root: Path, models_dir: Path, umap_every: int, save_every: int,
          backup_every: int, vis_every: int, force_restart: bool, visdom_server: str,
          no_visdom: bool):
    # Create a dataset and a dataloader
    dataset = SpeakerVerificationDataset(clean_data_root)
    loader = SpeakerVerificationDataLoader(
        dataset,
        speakers_per_batch,
        utterances_per_speaker,
        num_workers=4,
    )

    # Setup the device on which to run the forward pass and the loss. These can be different,
    # because the forward pass is faster on the GPU whereas the loss is often (depending on your
    # hyperparameters) faster on the CPU.
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    # FIXME: currently, the gradient is None if loss_device is cuda
    loss_device = torch.device("cpu")

    # Create the model and the optimizer
    model = SpeakerEncoder(device, loss_device)
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate_init)
    init_step = 1

    # Configure file path for the model
    model_dir = models_dir / run_id
    model_dir.mkdir(exist_ok=True, parents=True)
    state_fpath = model_dir / "encoder.pt"

    # Load any existing model
    if not force_restart:
        if state_fpath.exists():
            print("Found existing model \"%s\", loading it and resuming training." % run_id)
            checkpoint = torch.load(state_fpath)
            init_step = checkpoint["step"]
            model.load_state_dict(checkpoint["model_state"])
            optimizer.load_state_dict(checkpoint["optimizer_state"])
            optimizer.param_groups[0]["lr"] = learning_rate_init
        else:
            print("No model \"%s\" found, starting training from scratch." % run_id)
    else:
        print("Starting the training from scratch.")
    model.train()
```

5.3 Synthesizer:

The synthesizer is Tacotron 2 without Wavenet. We use an open-source Tensorflow implementation4 of Tacotron 2 from which we strip Wavenet and implement the modifications added by SV2TTS.

5.3.1 Model Architecture:

We briefly present the top-level architecture of the modified Tacotron 2 without Wavenet. For further details, we invite the reader to take a look at the Tacotron papers.

Tacotron is a recurrent sequence-to-sequence model that predicts a mel spectrogram from text. It features an encoder-decoder structure (not to be mistaken with the speaker encoder of SV2TTS) that is bridged by a location-sensitive attention mechanism. Individual characters from the text sequence are first embedded as vectors. Convolutional layers follow, so as to increase the span of a single encoder frame. These frames are passed through a bidirectional LSTM to produce the encoder output frames. This is where SV2TTS brings a modification to the architecture: a speaker embedding is concatenated to every frame that is output by the Tacotron encoder. The attention mechanism attends to the encoder output frames to generate the decoder input frames. Each decoder input frame is concatenated with the previous decoder frame output passed through a pre-net, making the model autoregressive. This concatenated vector goes through two unidirectional LSTM layers before being projected to a single mel spectrogram frame. Another projection of the same vector to a scalar allows the network to predict on its own that it should stop generating frames by emitting a value above a certain threshold. The entire sequence of frames is passed through a residual post-net before it becomes the mel spectrogram [4]. This architecture is represented in Figure 5.3.1.1.

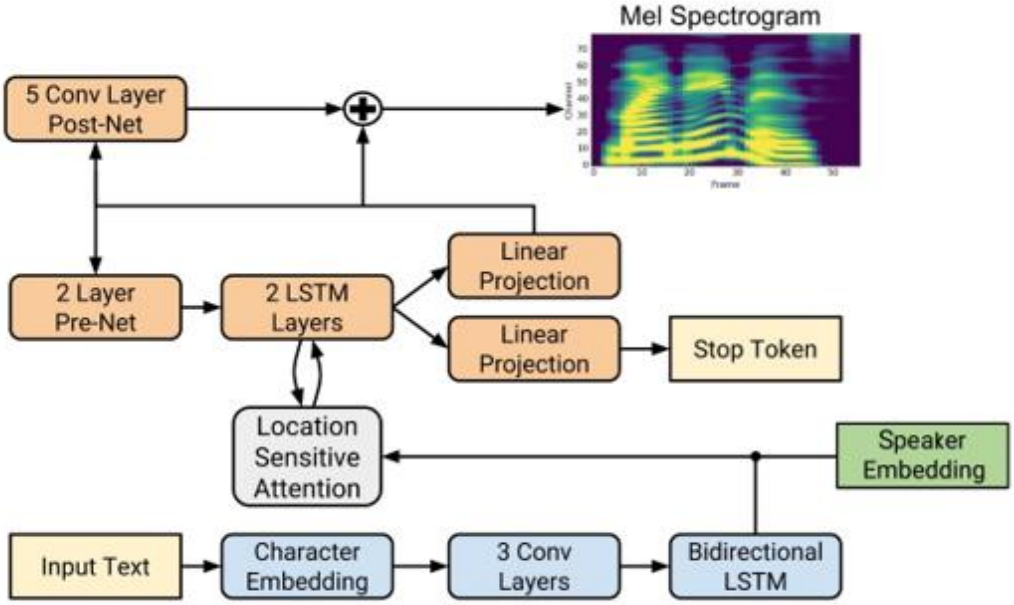


Figure 5.3.1.1: The modified Tacotron architecture

The target mel spectrograms for the synthesizer contain additional features compared to those used for the speaker encoder. They are generated using a 50ms window and a 12.5ms step, resulting in spectrograms with 80 channels. In our implementation, the input texts are not modified for pronunciation, and the characters are processed as they are [4]. However, there are a few cleaning procedures applied, such as replacing abbreviations and numbers with their complete textual forms, converting all characters to ASCII, normalizing whitespaces, and converting all characters to lowercase. Although punctuation could be utilized, it is not present in our datasets.

5.3.2 Implementation:

- This code implementation of audio processing functions for tasks like spectrogram generation, waveform manipulation, and audio representation conversion. It includes features such as loading/saving audio, preemphasis, spectrogram computation, inverse spectrogram conversion, and utility functions for STFT, padding, scale conversion, and normalization. These functions are useful for speech and audio applications like synthesis, conversion, and analysis.

nesizer > audio.py > ...

```python
import librosa
import librosa.filters
import numpy as np
from scipy import signal
from scipy.io import wavfile
import soundfile as sf


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

def save_wav(wav, path, sr):
    wav *= 32767 / max(0.01, np.max(np.abs(wav)))
    #proposed by @dsmiller
    wavfile.write(path, sr, wav.astype(np.int16))

def save_wavenet_wav(wav, path, sr):
    sf.write(path, wav.astype(np.float32), sr)

def preemphasis(wav, k, preemphasize=True):
    if preemphasize:
        return signal.lfilter([1, -k], [1], wav)
    return wav

def inv_preemphasis(wav, k, inv_preemphasize=True):
    if inv_preemphasize:
        return signal.lfilter([1], [1, -k], wav)
    return wav

def start_and_end_indices(quantized, silence_threshold=2):
    for start in range(quantized.size):
        if abs(quantized[start] - 127) > silence_threshold:
            break
    for end in range(quantized.size - 1, 1, -1):
        if abs(quantized[end] - 127) > silence_threshold:
            break

    assert abs(quantized[start] - 127) > silence_threshold
    assert abs(quantized[end] - 127) > silence_threshold

    return start, end

def get_hop_size(hparams):
    hop_size = hparams.hop_size
    if hop_size is None:
        assert hparams.frame_shift_ms is not None
        hop_size = int(hparams.frame_shift_ms / 1000 * hparams.sample_rate)
    return hop_size
```

izer > audio.py > ...

```python
def linearspectrogram(wav, hparams):
    D = _stft(preemphasis(wav, hparams.preemphasis, hparams.preemphasize), hparams)
    S = _amp_to_db(np.abs(D), hparams) - hparams.ref_level_db

    if hparams.signal_normalization:
        return _normalize(S, hparams)
    return S

def melspectrogram(wav, hparams):
    D = _stft(preemphasis(wav, hparams.preemphasis, hparams.preemphasize), hparams)
    S = _amp_to_db(_linear_to_mel(np.abs(D), hparams), hparams) - hparams.ref_level_db

    if hparams.signal_normalization:
        return _normalize(S, hparams)
    return S

def inv_linear_spectrogram(linear_spectrogram, hparams):
    """Converts linear spectrogram to waveform using librosa"""
    if hparams.signal_normalization:
        D = _denormalize(linear_spectrogram, hparams)
    else:
        D = linear_spectrogram

    S = _db_to_amp(D + hparams.ref_level_db) #Convert back to linear

    if hparams.use_lws:
        processor = _lws_processor(hparams)
        D = processor.run_lws(S.astype(np.float64).T ** hparams.power)
        y = processor.istft(D).astype(np.float32)
        return inv_preemphasis(y, hparams.preemphasis, hparams.preemphasize)
    else:
        return inv_preemphasis(_griffin_lim(S ** hparams.power, hparams), hparams.preemphasis, hparams.preemphasize)
```

```python
def _lws_processor(hparams):
    import lws
    return lws.lws(hparams.n_fft, get_hop_size(hparams), fftsize=hparams.win_size, mode="speech")

def _griffin_lim(S, hparams):
    """librosa implementation of Griffin-Lim
    Based on https://github.com/librosa/librosa/issues/434
    """
    angles = np.exp(2j * np.pi * np.random.rand(*S.shape))
    S_complex = np.abs(S).astype(np.complex)
    y = _istft(S_complex * angles, hparams)
    for i in range(hparams.griffin_lim_iters):
        angles = np.exp(1j * np.angle(_stft(y, hparams)))
        y = _istft(S_complex * angles, hparams)
    return y

def _stft(y, hparams):
    if hparams.use_lws:
        return _lws_processor(hparams).stft(y).T
    else:
        return librosa.stft(y=y, n_fft=hparams.n_fft, hop_length=get_hop_size(hparams), win_length=hparams.win_size)

def _istft(y, hparams):
    return librosa.istft(y, hop_length=get_hop_size(hparams), win_length=hparams.win_size)

##########################################################
#Those are only correct when using lws!!! (This was messing with Wavenet quality for a long time!)
def num_frames(length, fsize, fshift):
    """Compute number of time frames of spectrogram
    """
    pad = (fsize - fshift)
    if length % fshift == 0:
        M = (length + pad * 2 - fsize) // fshift + 1
    else:
        M = (length + pad * 2 - fsize) // fshift + 2
    return M
```

- The model isn't instantiated and loaded in memory until needed or until load() is called. Instantiates and loads the model given the weights file that was passed in the constructor and Synthesizes mel spectrograms from texts and speaker embeddings

esizer > inference.py > Synthesizer > load

```python
import torch
from synthesizer import audio
from synthesizer.hparams import hparams
from synthesizer.models.tacotron import Tacotron
from synthesizer.utils.symbols import symbols
from synthesizer.utils.text import text_to_sequence
from vocoder.display import simple_table
from pathlib import Path
from typing import Union, List
import numpy as np
import librosa


class Synthesizer:
    sample_rate = hparams.sample_rate
    hparams = hparams

    def __init__(self, model_fpath: Path, verbose=True):
        """
        The model isn't instantiated and loaded in memory until needed or until load() is called.

        :param model_fpath: path to the trained model file
        :param verbose: if False, prints less information when using the model
        """
        self.model_fpath = model_fpath
        self.verbose = verbose

        # Check for GPU
        if torch.cuda.is_available():
            self.device = torch.device("cuda")
        else:
            self.device = torch.device("cpu")
        if self.verbose:
            print("Synthesizer using device:", self.device)

        # Tacotron model will be instantiated later on first use.
        self._model = None

    def is_loaded(self):
        """
        Whether the model is loaded in memory.
        """
        return self._model is not None
```

inference.py > Synthesizer > load

```python
def is_loaded(self):
    """
    Whether the model is loaded in memory.
    """
    return self._model is not None

def load(self):
    """
    Instantiates and loads the model given the weights file that was passed in the constructor.
    """
    self._model = Tacotron(embed_dims=hparams.tts_embed_dims,
                           num_chars=len(symbols),
                           encoder_dims=hparams.tts_encoder_dims,
                           decoder_dims=hparams.tts_decoder_dims,
                           n_mels=hparams.num_mels,
                           fft_bins=hparams.num_mels,
                           postnet_dims=hparams.tts_postnet_dims,
                           encoder_K=hparams.tts_encoder_K,
                           lstm_dims=hparams.tts_lstm_dims,
                           postnet_K=hparams.tts_postnet_K,
                           num_highways=hparams.tts_num_highways,
                           dropout=hparams.tts_dropout,
                           stop_threshold=hparams.tts_stop_threshold,
                           speaker_embedding_size=hparams.speaker_embedding_size).to(self.device)

    self._model.load(self.model_fpath)
    self._model.eval()

    if self.verbose:
        print("Loaded synthesizer \"%s\" trained to step %d" % (self.model_fpath.name, self._model.state_dict()["step"]))
```

- Preprocess the Dataset and apply on model for training.

```
iesizer > preprocess.py > ...
from multiprocessing.pool import Pool
from synthesizer import audio
from functools import partial
from itertools import chain
from encoder import inference as encoder
from pathlib import Path
from utils import logmmse
from tqdm import tqdm
import numpy as np
import librosa


def preprocess_dataset(datasets_root: Path, out_dir: Path, n_processes: int, skip_existing: bool, hparams,
                       no_alignments: bool, datasets_name: str, subfolders: str):
    # Gather the input directories
    dataset_root = datasets_root.joinpath(datasets_name)
    input_dirs = [dataset_root.joinpath(subfolder.strip()) for subfolder in subfolders.split(",")]
    print("\n    ".join(map(str, ["Using data from:"] + input_dirs)))
    assert all(input_dir.exists() for input_dir in input_dirs)

    # Create the output directories for each output file type
    out_dir.joinpath("mels").mkdir(exist_ok=True)
    out_dir.joinpath("audio").mkdir(exist_ok=True)

    # Create a metadata file
    metadata_fpath = out_dir.joinpath("train.txt")
    metadata_file = metadata_fpath.open("a" if skip_existing else "w", encoding="utf-8")

    # Preprocess the dataset
    speaker_dirs = list(chain.from_iterable(input_dir.glob("*") for input_dir in input_dirs))
    func = partial(preprocess_speaker, out_dir=out_dir, skip_existing=skip_existing,
                   hparams=hparams, no_alignments=no_alignments)
    job = Pool(n_processes).imap(func, speaker_dirs)
    for speaker_metadata in tqdm(job, datasets_name, len(speaker_dirs), unit="speakers"):
        for metadatum in speaker_metadata:
            metadata_file.write("|".join(str(x) for x in metadatum) + "\n")
    metadata_file.close()

    # Verify the contents of the metadata file
    with metadata_fpath.open("r", encoding="utf-8") as metadata_file:
        metadata = [line.split("|") for line in metadata_file]
    mel_frames = sum([int(m[4]) for m in metadata])
    timesteps = sum([int(m[3]) for m in metadata])
    sample_rate = hparams.sample_rate
    hours = (timesteps / sample_rate) / 3600
    print("The dataset consists of %d utterances, %d mel frames, %d audio timesteps (%.2f hours)." %
          (len(metadata), mel_frames, timesteps, hours))
    print("Max input length (text chars): %d" % max(len(m[5]) for m in metadata))
    print("Max mel frames length: %d" % max(int(m[4]) for m in metadata))
    print("Max audio timesteps length: %d" % max(int(m[3]) for m in metadata))


def create_embeddings(synthesizer_root: Path, encoder_model_fpath: Path, n_processes: int):
    wav_dir = synthesizer_root.joinpath("audio")
    metadata_fpath = synthesizer_root.joinpath("train.txt")
    assert wav_dir.exists() and metadata_fpath.exists()
    embed_dir = synthesizer_root.joinpath("embeds")
    embed_dir.mkdir(exist_ok=True)

    # Gather the input wave filepath and the target output embed filepath
    with metadata_fpath.open("r") as metadata_file:
        metadata = [line.split("|") for line in metadata_file]
        fpaths = [(wav_dir.joinpath(m[0]), embed_dir.joinpath(m[2])) for m in metadata]

    # TODO: improve on the multiprocessing, it's terrible. Disk I/O is the bottleneck here.
    # Embed the utterances in separate threads
    func = partial(embed_utterance, encoder_model_fpath=encoder_model_fpath)
    job = Pool(n_processes).imap(func, fpaths)
    list(tqdm(job, "Embedding", len(fpaths), unit="utterances"))
```

- Train the synthesizer model on the preprocess dataset.

esizer > train.py > ...

```python
from datetime import datetime
from functools import partial
from pathlib import Path

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

from synthesizer import audio
from synthesizer.models.tacotron import Tacotron
from synthesizer.synthesizer_dataset import SynthesizerDataset, collate_synthesizer
from synthesizer.utils import ValueWindow, data_parallel_workaround
from synthesizer.utils.plot import plot_spectrogram
from synthesizer.utils.symbols import symbols
from synthesizer.utils.text import sequence_to_text
from vocoder.display import *


def np_now(x: torch.Tensor): return x.detach().cpu().numpy()


def time_string():
    return datetime.now().strftime("%Y-%m-%d %H:%M")


def train(run_id: str, syn_dir: Path, models_dir: Path, save_every: int,  backup_every: int, force_restart: bool,
          hparams):
    models_dir.mkdir(exist_ok=True)

    model_dir = models_dir.joinpath(run_id)
    plot_dir = model_dir.joinpath("plots")
    wav_dir = model_dir.joinpath("wavs")
    mel_output_dir = model_dir.joinpath("mel-spectrograms")
    meta_folder = model_dir.joinpath("metas")
    model_dir.mkdir(exist_ok=True)
    plot_dir.mkdir(exist_ok=True)
    wav_dir.mkdir(exist_ok=True)
    mel_output_dir.mkdir(exist_ok=True)
    meta_folder.mkdir(exist_ok=True)

    weights_fpath = model_dir / f"synthesizer.pt"
    metadata_fpath = syn_dir.joinpath("train.txt")

    print("Checkpoint path: {}".format(weights_fpath))
    print("Loading training data from: {}".format(metadata_fpath))
    print("Using model: Tacotron")

    # Bookkeeping
    time_window = ValueWindow(100)
    loss_window = ValueWindow(100)
```

```python
def eval_model(attention, mel_prediction, target_spectrogram, input_seq, step,
               plot_dir, mel_output_dir, wav_dir, sample_num, loss, hparams):
    # Save some results for evaluation
    attention_path = str(plot_dir.joinpath("attention_step_{}_sample_{}".format(step, sample_num)))
    save_attention(attention, attention_path)

    # save predicted mel spectrogram to disk (debug)
    mel_output_fpath = mel_output_dir.joinpath("mel-prediction-step-{}_sample_{}.npy".format(step, sample_num))
    np.save(str(mel_output_fpath), mel_prediction, allow_pickle=False)

    # save griffin lim inverted wav for debug (mel -> wav)
    wav = audio.inv_mel_spectrogram(mel_prediction.T, hparams)
    wav_fpath = wav_dir.joinpath("step-{}-wave-from-mel_sample_{}.wav".format(step, sample_num))
    audio.save_wav(wav, str(wav_fpath), sr=hparams.sample_rate)

    # save real and predicted mel-spectrogram plot to disk (control purposes)
    spec_fpath = plot_dir.joinpath("step-{}-mel-spectrogram_sample_{}.png".format(step, sample_num))
    title_str = "{}, {}, step={}, loss={:.5f}".format("Tacotron", time_string(), step, loss)
    plot_spectrogram(mel_prediction, str(spec_fpath), title=title_str,
                     target_spectrogram=target_spectrogram,
                     max_len=target_spectrogram.size // hparams.num_mels)
    print("Input at step {}: {}".format(step, sequence_to_text(input_seq)))
```

5.4 Vocoder:

In SV2TTS and in Tacotron2, WaveNet is the vocoder. WaveNet has been at the heart of deep learning with audio since its release and remains state of the art when it comes to voice naturalness in TTS. It is however also known for being the slowest practical deep learning architecture at inference time. Several later papers brought improvements on that aspect to bring the generation near real-time or faster than real-time, with no or next to no hit to the quality of the generated speech. Nonetheless, WaveNet remains the vocoder in SV2TTS as speed is not the main concern and because Google's own WaveNet implementation with various improvements already generates at 8000 samples per second. This is in contrast with WaveNet which generates at 172 steps per second at best. At the time of the writing of this thesis, most open-source implementations of WaveNet are still vanilla implementations. We propose a simple scheme for describing the inference speed of autoregressive models. Given a target vector **u** with |**u**| samples to predict, the total time of inference $T(\mathbf{u})$ can be decomposed as:

$$T(\mathbf{u}) = |\mathbf{u}| \sum_{i=1}^{X} (c(op_i) + d(op_i))$$

where N is the number of matrix-vector products (∝ the number of layers) required to produce one sample, c(opi) is the computation time of layer i and d(opi) is the overhead of the computation (typically I/O operations) for layer i. Note that standard sampling rates for speech include 16kHz, 22.05kHz and 24kHz (while music is usually sampled at 44.1kHz), meaning that for just 5 seconds of audio |u| is close to 100,000 samples. The standard WaveNet architecture accounts for three stacks of 10 residual blocks of two layers each, leading to N = 60 [2].

WaveRNN, the model proposed, improves on WaveNet by not only reducing the contribution from N but also from u, c(opi) and d(opi). The vocoder model we use is an open source PyTorch implementation5 that is based on WaveRNN but presents quite a few different design choices made. We'll refer to this architecture as the "alternative WaveRNN".

5.4.1 Model Architecture:

In WaveRNN, the entire 60 convolutions from WaveNet are replaced by a single GRU layer. The authors maintain that the high non-linearity of a GRU layer alone is close enough to encompass the complexity of the entire WaveNet model. Indeed, they report a MOS of 4.51 ± 0.08 for Wavenet and 4.48 ± 0.07 for their best WaveRNN model. The inputs to the model are the GTA mel spectrogram generated by the synthesizer, with the ground truth audio as target. At training time, the model predicts fixed-size waveform segments. The forward pass of WaveRNN is implemented with only N = 5 matrix-vector products in a coarse-fine scheme where the lower 8 bits (coarse) of the target 16 bits sample are predicted first and then used to condition the prediction of the higher 8 bits (fine). The prediction is over the parameters of a distribution from which the output is sampled [2].

Finally, we improve on |u| with batched sampling. In batched sampling, the utterance is divided in segments of fixed length and the generation is done in parallel over all segments. To preserve some context between the end of a segment and the beginning of the subsequent one, a small section of the end of a segment is repeated at the beginning of the next one. This process is called folding. The model then forwards the folded segments. To retrieve the unfolded tensor, the overlapping sections of consecutive segments are merged by a cross-fade. This is illustrated in Figure 5.4.1.1. We use batched sampling with the alternative WaveRNN, with a segment length of 8000 samples and an overlap length of 400 samples. With these parameters, a folded batch of size 2 will yield a bit more than 1 second of audio for 16kHz speech.

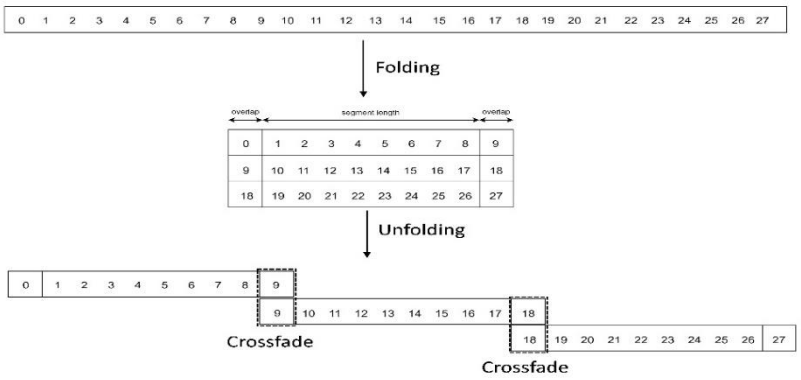


Figure 5.4.1.1: Batched sampling of a tensor. Note how the overlap is repeated over each next segment in the folded tensor.

The alternative WaveRNN is the architecture we use. There is no documentation nor paper for this model, so we rely on the source code and on the diagram of the author (Figure 5.4.1.2) to understand its inner workings. At each training step, a mel spectrogram and its corresponding waveform are cut in the same number of segments. The inputs to the model are the spectrogram segment t to predict and the waveform segment t − 1. The model is expected to output the waveform segment t of identical length. The mel spectrogram goes through an up sampling network to match the length of the target waveform (the number of mel channels remains the same). A Resnet-like model also uses the spectrogram as input to generate features that will condition the layers throughout the transformation of the mel spectrogram to a waveform. The resulting vector is repeated to match the length of the waveform segment. This conditioning vector is then split equally four ways along the channel dimension, and the first part is concatenated with the upsampled spectrogram and with the waveform segment of the previous timestep. The resulting vector goes through several transformation with skip connections: first two GRU layers then a dense layer. Between each step, the conditioning vector is concatenated with the intermediate waveform. Finally, two dense layers produce a distribution over discrete values that correspond to a 9-bit encoding of mu-law companded audio [5].

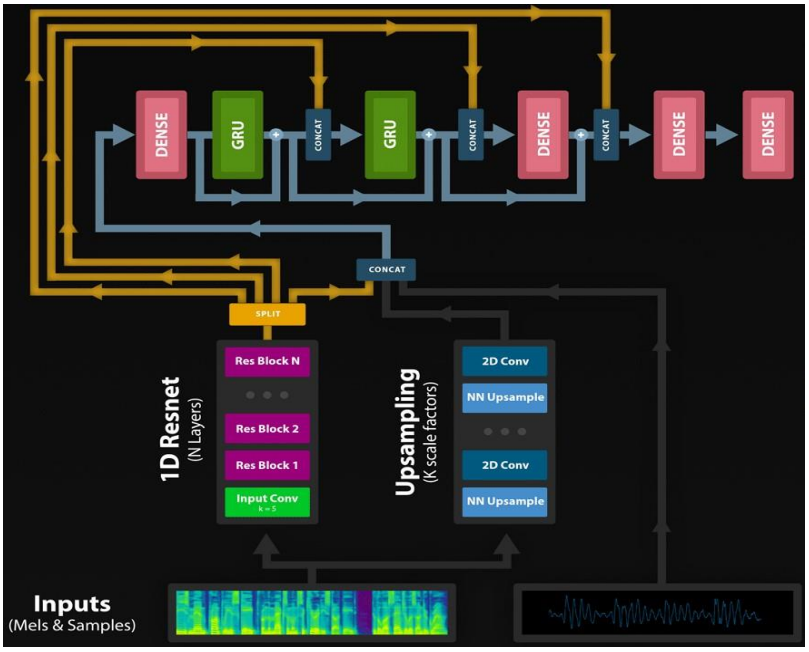


Figure 5.4.1.2: The alternative WaveRNN architecture.

5.4.2 Implementation:

- The Vocoder provides functions for audio sample conversion, loading/saving audio files, signal splitting/combining, 16-bit encoding, linear-to-mel spectrogram conversion,

:r > audio.py > save_wav

```
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)
```

- Infers the waveform of a mel spectrogram output by the synthesizer (the format must match that of the synthesizer.

```
> inference.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
```

- Trainining the Vocoder model on the preprocess vocoder dataset.

train.py > ...

```python
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)])
```

5.5 Replication Toolbox:

To facilitate easy access and usage of the framework without the need for prior study, we have developed a graphical interface known as the "SV2TTS toolbox." This interface, depicted in Figure 5.5.1, is designed using Python and the Qt4 graphical interface, ensuring cross-platform compatibility. Users can swiftly navigate the toolbox and utilize the framework's functionalities without extensive preparation.

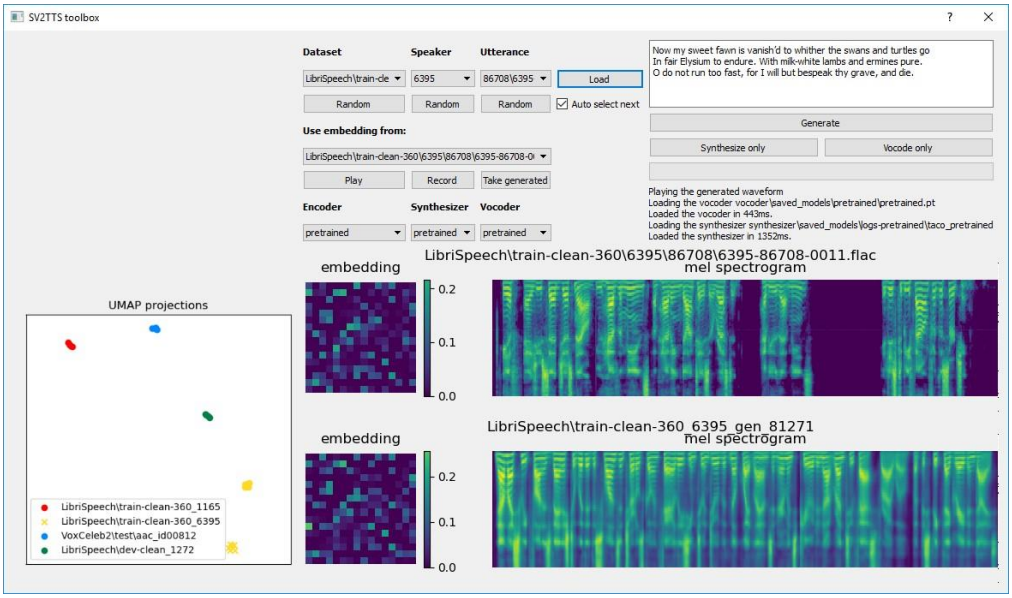


Figure 5.5.1: The SV2TTS toolbox interface. This image is best viewed on a digital support.

At the start, the user chooses an audio file containing an utterance from any dataset stored on their computer. The SV2TTS toolbox is equipped to handle various widely used speech datasets, and it can also be personalized to incorporate new ones. Additionally, users have the option to record their own utterances, enabling them to clone their unique voices.

After loading an utterance, the toolbox proceeds to calculate its embedding and automatically updates the UMAP projections. While the mel spectrogram of the utterance is displayed (middle row on the right), it serves as a visual reference and does not contribute to any computations. The embedding vector is illustrated using a heatmap plot located on the left of the spectrogram. It's important to note that embeddings are unidimensional vectors, and the square shape in the visualization does not convey any structural information about the

embedding values. The purpose of drawing embeddings is to provide visual indicators of the differences between two embeddings.

Once an embedding is computed, it can be utilized to generate a spectrogram. The user has the ability to input any desired text (located at the top right of the interface) for synthesis. It is important to note that our model does not support punctuation and will ignore it. To control the prosody of the synthesized utterance, the user needs to insert line breaks between the sections that should be synthesized individually. The resulting spectrogram is a concatenation of these parts and is displayed at the bottom right of the interface. Generating the spectrogram multiple times using the same sentences will yield different outputs. Additionally, the user can employ the vocoder to generate the segment corresponding to the synthesized spectrogram. A loading bar indicates the progress of the generation, and upon completion, the embedding of the synthesized utterance is generated and displayed on the left side of the synthesized spectrogram. This synthesized embedding can also be projected using UMAP. Users have the freedom to consider this embedding as a reference for further generation or experimentation.

5.5.1 Implementation-

- Create the User interface for using the toolbox.

```
ui.py > UI > set_audio_device
import sys
from pathlib import Path
from time import sleep
from typing import List, Set
from warnings import filterwarnings, warn

import matplotlib.pyplot as plt
import numpy as np
import sounddevice as sd
import soundfile as sf
import umap
from PyQt5.QtCore import Qt, QStringListModel
from PyQt5.QtWidgets import *
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas

from encoder.inference import plot_embedding_as_heatmap
from toolbox.utterance import Utterance

filterwarnings("ignore")


colormap = np.array([
    [0, 127, 70],
    [255, 0, 0],
    [255, 217, 38],
    [0, 135, 255],
    [165, 0, 165],
    [255, 167, 255],
    [97, 142, 151],
    [0, 255, 255],
    [255, 96, 38],
    [142, 76, 0],
    [33, 0, 127],
    [0, 0, 0],
    [183, 183, 183],
    [76, 255, 0],
], dtype=np.float) / 255

default_text = \
    "Welcome to the toolbox! To begin, load an utterance from your datasets or record one " \
    "yourself.\nOnce its embedding has been created, you can synthesize any text written here.\n" \
    "The synthesizer expects to generate " \
    "outputs that are somewhere between 5 and 12 seconds.\nTo mark breaks, write a new line. " \
    "Each line will be treated separately.\nThen, they are joined together to make the final " \
    "spectrogram. Use the vocoder to generate audio.\nThe vocoder generates almost in constant " \
    "time, so it will be more time efficient for longer inputs like this one.\nOn the left you " \
    "have the embedding projections. Load or record more utterances to see them.\nIf you have " \
    "at least 2 or 3 utterances from a same speaker, a cluster should form.\nSynthesized " \
    "utterances are of the same color as the speaker whose voice was used, but they're " \
    "represented with a cross."


class UI(QDialog):
    min_umap_points = 4
    max_log_lines = 5
    max_saved_utterances = 20

    def draw_utterance(self, utterance: Utterance, which):
        self.draw_spec(utterance.spec, which)
        self.draw_embed(utterance.embed, utterance.name, which)

    def draw_embed(self, embed, name, which):
        embed_ax, _ = self.current_ax if which == "current" else self.gen_ax
        embed_ax.figure.suptitle("" if embed is None else name)

        ## Embedding
```

- Running the toolbox in system.

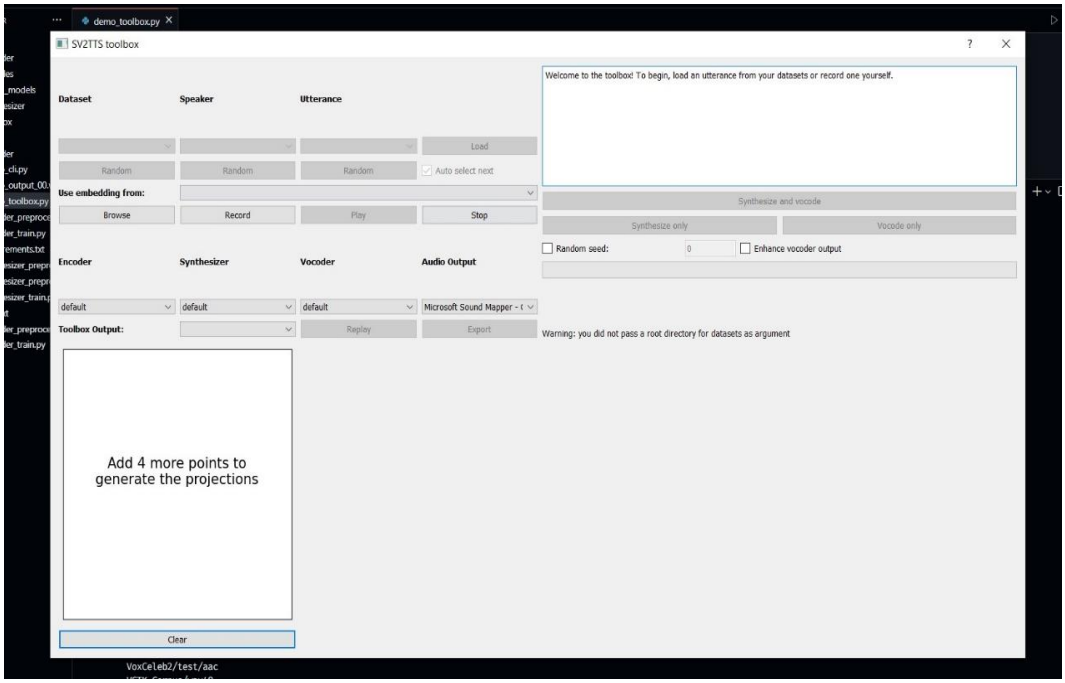
mo_toolbox.py > ...

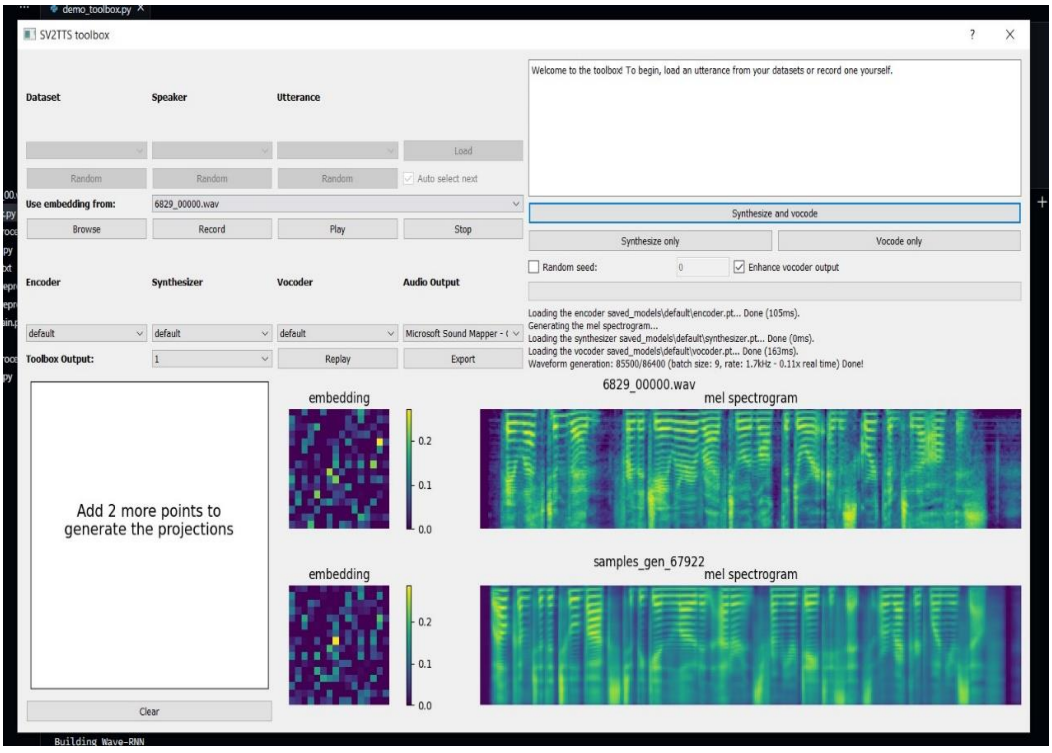
```python
import argparse
import os
from pathlib import Path

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)
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
C:\Users\Saurabh\Desktop\Voice>C:/Users/Saurabh/anaconda3/Scripts/activate

(base) C:\Users\Saurabh\Desktop\Voice>conda activate myenv

(myenv) C:\Users\Saurabh\Desktop\Voice>python demo_toolbox.py
Arguments:
    datasets_root:   None
    models_dir:      saved_models
    cpu:             False
    seed:            None

Warning: you did not pass a root directory for datasets as argument.
The recognized datasets are:
        LibriSpeech/dev-clean
        LibriSpeech/dev-other
        LibriSpeech/test-clean
        LibriSpeech/test-other
        LibriSpeech/train-clean-100
        LibriSpeech/train-clean-360
        LibriSpeech/train-other-500
        LibriTTS/dev-clean
        LibriTTS/dev-other
        LibriTTS/test-clean
        LibriTTS/test-other
        LibriTTS/train-clean-100
        LibriTTS/train-clean-360
        LibriTTS/train-other-500
        LJSpeech-1.1
        VoxCeleb1/wav
        VoxCeleb1/test_wav
        VoxCeleb2/dev/aac
        VoxCeleb2/test/aac
        VCTK-Corpus/wav48
```

- Displayed the User Interface.





Chapter 6

CONCLUSION

6. CONCLUSION

In conclusion, this study presents a neural network-based system for multispeaker Text-to-Speech (TTS) production. The system leverages a dynamically trained writer encoder network, a sequence-to-sequence TTS synthesis network, and a Tacotron 2-based artificial neural vocoder. It demonstrates the ability to generate high-quality speech for both seen and unseen speakers, achieving speech quality comparable to natural discourse about the target speakers.

Transfer learning plays a crucial role in the system, reducing the need for extensive multispeaker TTS data for training. The approach eliminates the requirement for writer identification labels in synthesizer training data and transcripts or high-quality linguistic annotations in speech encoder training data, making the system more flexible and practical.

However, limitations are identified, including the use of a low-dimensional vector for modeling speaker variations, which restricts the system's capacity to fully utilize extensive symbolic speech. Improving speaker similarity when comparing more than a few reference moments requires an adaptation method, and the system lacks the capability to modify accents.

Furthermore, despite the utilization of a WaveNet vocoder, the proposed approach falls short of achieving human-level naturalness. This is attributed to the challenge of creating speech with limited details per speaker and the utilization of datasets with poor data quality. The system also struggles to fully distinguish the speaker's voice from the prosody present in the text.

In summary, this neural network-based system shows promise for multispeaker TTS production, with the ability to generate high-quality speech for both known and unknown speakers. Further improvements are needed to address limitations related to speaker individuality, naturalness, accent modification, and the distinction between the speaker's voice and prosody. These findings provide a foundation for future advancements in multispeaker TTS technology.

Chapter 7

FUTURE SCOPE

7. FUTURE SCOPE

Deep learning-based voice replication has undergone tremendous progress in the past few years, but there is still plenty of room for future advancements. Here are some potential future uses of deep learning over voice replication:

- ***Enhanced Realism***: Developing dissimilar voices in voice replication is a crucial objective, and although significant progress has been made, there is still scope for improvement. Future advancements in deep learning can focus on enhancing the realism of replicated voices, aiming for even greater convincing power.
- ***Personalized Voices***: Personalized conversation replication is a different one fascinating field for studies. Algorithms for deep learning can be trained to replicate someone's voice versus making generic humanoid voices. This might be useful for voice-activated personal computers or even in keeping someone's speaking voice after passing away.
- ***Multilingual Voices***: Current voice replication systems are often limited to a single language, but the future could see the development of multilingual systems. These systems could replicate voices in multiple languages, making them useful for international communication and language learning applications.
- ***Expressive Voices***: Future deep learning algorithms can enable the replication of emotions and expressions in voices, enhancing their versatility for applications like chatbots, virtual assistants, and entertainment.
- ***Limited Data Training***: Deep learning-based conversation replication has the potential to revolutionize human-machine communication by creating highly sophisticated voice replication systems that mimic the intricacies of human speech.

The future of deep learning-based conversation replication holds immense potential to transform human-machine communication and interpersonal interactions. Advancements in this field are likely to result in more sophisticated and adaptable voice replication systems, bringing us closer to replicating the full range of human speech nuances and subtleties.

REFERENCES

[1] Jia, Ye, et al. "Transfer learning from speaker verification to multispeaker text-to-speech synthesis." Advances in neural information processing systems 31 (2018). Wan, Li, et al. "Generalized end-to-end loss for speaker verification." 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2018.

[2] Zhao, Li, and Feifan Chen. "Research on voice cloning with a few samples." 2020 International Conference on Computer Network, Electronic and Automation (ICCNEA). IEEE, 2020.

[3] Dai, Dongyang, et al. "Cloning one's voice using very limited data in the wild." ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2022.

[4] Shen Jonathan, et al. "Natural tts synthesis by conditioning wavenet on mel spectrogram predictions." 2018 IEEE international conference on acoustics, speech and signal processing (ICASSP). IEEE, 2018.

[5] Song, Wei, et al. "Dian: Duration informed auto-regressive network for voice cloning." ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2021.

[6] Jin, Zeyu, et al. "FFTNet: A real-time speaker-dependent neural vocoder." 2018 IEEE international conference on acoustics, speech and signal processing (ICASSP). IEEE, 2018.

[7] Gonzalvo, Xavi, et al. "Recent advances in Google real-time HMM-driven unit selection synthesizer." (2016).

[8] Arik, Sercan, et al. "Deep voice 2: Multi-speaker neural text-to-speech." arXiv preprint arXiv:1705.08947 (2017).

PUBLICATION DETAILS

PAPER TITLE	JOURNAL NAME	ISSUE DATE	e-ISSN/p-ISSN
Literature Review on Replication of Voice Using Deep Learning Technique	International Journal of Innovative Research in Computer and Communication Engineering	4 April 2023	2320-9801/ 2320-9801

PROJECT GROUP MEMBERS

Name: Saurabh Milind Kedar

Address: Kolbaswami Ward, Sindi rly (442105)

Email id: saurabhkedar2532@gmail.com

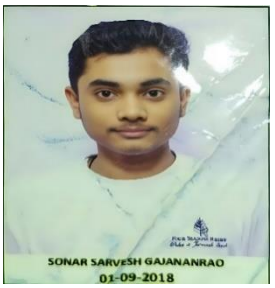
Mobile No: 7666530973

Name: Sarvesh Gajananrao Sonar

Address: Arjun Nagar, Amravati (444603)

Email id: sarveshsonar2018@gmail.com

Mobile No: 9307575728



Name: Smitesh Gajananrao Sonar

Address: Arjun Nagar, Amravati (444603)

Email id: smiteshsonar2018@gmail.com

Mobile No: 8459080443

Name: Trunay Murlidhar Wanjari

Address: Dattatrey Nagar, Nagpur (440024)

Email id: trunaywanjari5@gmail.com

Mobile No: 7558465140