POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING

Master of Science in Computer Engineering

Master Degree Thesis

Automated creation of Podcasts empowered by Text-to-Speech

Supervisors
Prof. Antonio Vetrò
Dr. Giovanni Garifo

Candidato
Simone Sasso

Academic Year 2021-2022

To my family

Abstract

The goal of Text-to-Speech (TTS) is to synthesize human-like speech from texts. Over the last decade, this research field has seen incredible improvements, thanks to the significant advances in deep learning and its extensive development. TTS models based on neural networks have been able to achieve results that are almost indistinguishable from human speech. Consequently, this technology has become more and more popular, drastically improving the way people interact with machines. Despite its current progress, neural TTS is far from a solved problem and still presents several criticalities. Both training and inference require heavy computational resources, and models tend to make mistakes when dealing with corner cases or text which belongs to a different domain with respect to the training set. This thesis will examine the development of a pipeline for the generation of podcasts, by using a Text-to-Speech model to read news articles. Since there are many different neural TTS architectures, there will be a discussion on the motivations that lead to the choice of the final model. This was trained on a high performance computing cluster, using an Italian public domain dataset. In order to adapt it to the synthesis of long news text, an additional preprocessing step has been introduced in the pipeline. Care has been taken to implement a normalizer that could correctly handle technical text, which is crucial when dealing with economic or scientific articles. The thesis will also explain how the model has been finetuned on a smaller dataset of a different speaker, successfully converting the synthesized voice in a short amount of time, thanks to transfer learning. As of today, there is a lack of high quality open source TTS models, outside of commercial services offered by big tech companies. The main reason is that creating a TTS dataset is an expensive process that requires the alignment of transcripts to tens of hours of recorded speech. In order to generate the dataset used for finetuning, a

different approach was followed. Leveraging the recent improvements in the Speech-to-Text field, it was possible to automate the dataset generation process without the need for transcribed text. Hopefully, the same technique can be applied to generate datasets for low resource languages, which are plagued by a scarcity of training data. In the end, it will be described how the model has been deployed as a microservice, exploring the strategies used to mitigate the long inference times.

Acknowledgements

I would like to thank Prof. Antonio Vetrò for his review of the evaluation process and Dr. Giovanni Garifo for his positive feedback during this thesis work.

Computational resources provided by hpc@polito, which is a project of Academic Computing within the Department of Control and Computer Engineering at the Politecnico di Torino (http://hpc.polito.it).

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Humans have been trying to build speaking machines for a long time, long before the advent of digital revolution [7].
Many techniques have since been developed to synthesize intelligible speech. However, the results always sounded unnatural, despite being intelligible. Natural speech synthesis has been possible only in recent years, when researchers started using neural networks to tackle this challenge. The field of study is known as Text-To-Speech (TTS), and its objective is to generate human-like speech from text. In particular, neural TTS started in 2016, with Google's WaveNet [8], which was a revolution in the TTS field. In May 2022, the first model to achieve human-level quality speech has been published [9].
During the same time period, digital audio consumption and podcast listening in the US showed a steady growth [10]. A podcast is a digital recording that is available on the internet [11]. Although this definition is rather generic, podcasts are generally produced by recording speech of physical people through a microphone, usually in a dedicated studio. Thanks to the great improvements in neural TTS, it would be interesting to explore whether it's possible to generate podcasts automatically, without human intervention, by training a model on a pre-recorded speech dataset.
In the context of this work, the goal is to produce a podcast that is able to perform a press review, by reading articles selected by the user. This requires the generated speech to be not only intelligible, but also natural and robust. A robust synthesis can generalize well with different domains

and it doesn't make mistakes when dealing with corner cases. This is especially crucial with news articles that span over a wide variety of domains and contain technical terms which are not present in the training dataset. In order to deal with this aspect, it's important to build a normalizer that can correctly handle numbers, abbreviations, symbols and units of measurement, translating them into spoken form.
In the following chapters, the foundational theory behind the TTS domain will be presented, followed by a brief historical background. Over the last decade, many different neural TTS architectures have been developed and it would be unfeasible to cover them all. Instead, the focus will be on the models that had a significant impact in the research domain. Then, there will be an analysis of the dataset used and it will be emphasized how challenging it is to find high quality speech datasets. A possible mitigation of this problem will be presented as well, which is based on one of the most recently released automatic speech recognition models. Afterwards, I will discuss the models that I trained, motivating why I chose them, describing the mathematical theory behind them and explaining how they have been trained and then fine-tuned on a new speaker's dataset. This allows the user to select between a male and a female voice. Finally, more practical aspects will be analyzed, such as the implementation of an Italian text normalizer and the deployment of the trained model as a microservice. The order of the chapters roughly follows the chronological order of the thesis work, showing the entire development process, from the research to the deployment.

Chapter 2

Background and related work

2.1 Analysis of audio signals

At a high level, a Text-To-Speech system converts an input text into a waveform. As it will be discussed later, a common pipeline is to generate some acoustic features starting from text, then convert the acoustic features into a waveform. One of the most used acoustic representations in neural TTS is the Mel spectrogram. This section will briefly describe how a Mel spectrogram is computed, starting from a speech signal.

2.1.1 Waveform

A signal is a physical quantity that varies over one or more independent variables, such as time or space [12]. An audio signal is a mechanical wave that propagates through a medium, like air. It can be visualized by plotting the changes in air pressure over time.
The variation in the air pressure is represented by the signal's amplitude. The higher the amplitude, the louder the audio signal is.

2.1.2 AD conversion

An audio signal in the physical world is analog, which means that it's continuous both on the time axis as well as the amplitude. In order to be understood by a computer, an analog signal needs to be converted into

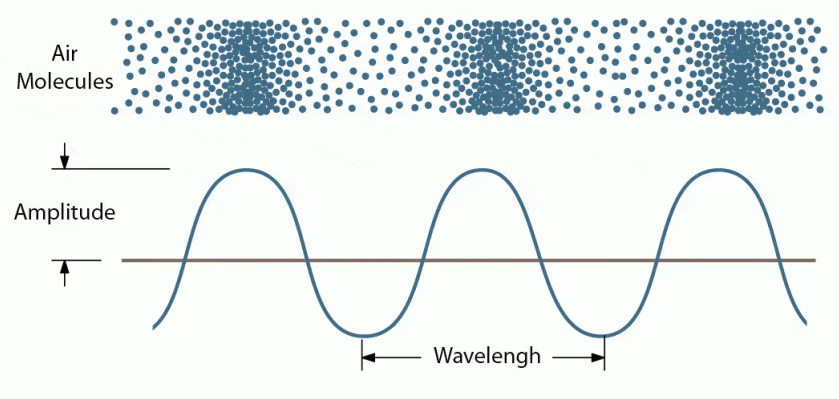


Figure 2.1: The sound wave. Image from [1].

a digital one. This process is called analog-to-digital conversion and it consists of two stages: sampling and quantization.

Sampling

A signal $s(t)$ is discretized over the time axis by taking amplitude points at equidistant time intervals T, where T is the sampling period and $f_s = \frac{1}{T}$ is the sampling frequency. If f_B is the bandwidth of the continuous signal, then it's possible to sample it without loss of information if $f_s > 2f_B$. This is known as the *Nyquist-Shannon sampling theorem* [13]. If the bandwidth f_B of the signal is above the Nyquist frequency $f_N = \frac{f_s}{2}$, then the reconstructed signal is corrupted and contains artifacts with respect to the original signal; this phenomenon is called *aliasing*. In practice, all the frequencies above the Nyquist frequency are shifted down to a lower frequency.

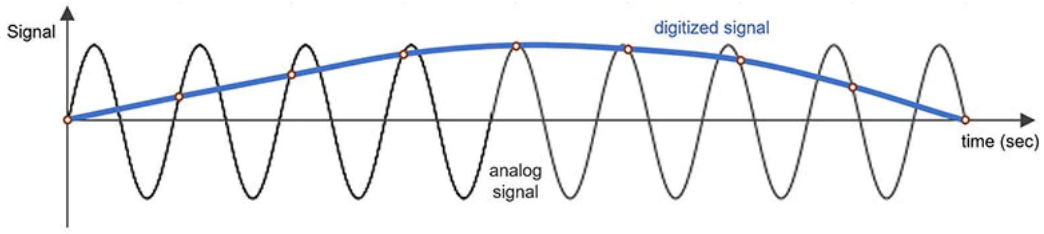


Figure 2.2: A lower frequency emerges in the reconstruction, due to undersampling. Image from [2].

Audio is typically sampled at 44.1 kHz; in fact, this is the sampling rate used by CDs. This is compliant with the Nyquist theorem, considering that the highest frequency that humans can hear is about 20 kHz [14]. In the TTS field though, the most common sampling rate is 22050 Hz. This is the sampling rate of the LJ Speech [15] dataset, which is a public domain dataset commonly used by researchers to compare the performances of TTS models.

Quantization

In this step, the same idea of sampling is applied to the amplitude axis. The continuous amplitude values are mapped to a smaller set with a finite number of elements, since computers don't have an infinite precision. This can be done by either truncating or rounding the amplitudes [3]. The difference with respect to sampling is that this operation is irreversible, since it introduces a quantization error; it's not possible to reconstruct the original signal starting from the quantized signal.

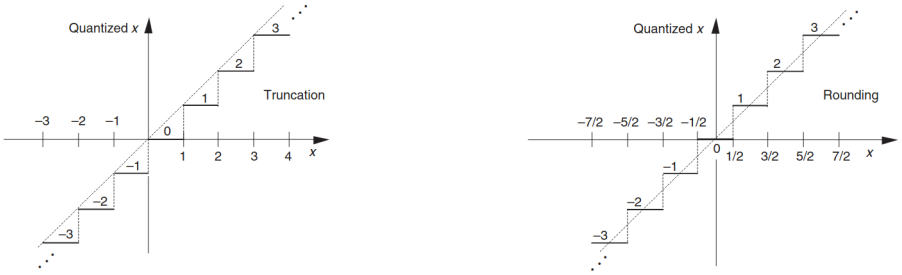


Figure 2.3: Rounding has a lower maximum quantization error w.r.t. truncation. [3]

2.1.3 The Fourier transform

The major mathematical tool to perform signal analysis is the Fourier transform. The goal is to take any signal and decompose it into the individual frequencies that make it up. The continuous Fourier transform is defined as

$$X(f) = \int_{-\infty}^{+\infty} x(t) e^{-i2\pi ft}\, dt \tag{2.1}$$

The transform takes as input a frequency, and returns a complex number that represents the strength of that given frequency in the original signal.

Discrete Fourier transform

The Fourier transform is used for analyzing analog signals; however, as described before, computers deal with digital signals. For this reason, in the digital domain the *Discrete Fourier Transform* (DFT) is used [3]:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-i2\pi nk/N} \tag{2.2}$$

where

- $x(n)$ is the nth discrete input sample;
- N is the total number of input samples as well as the number of output frequencies;
- $X(k)$ is the kth output frequency component.

Given N input samples, the DFT computes the magnitude of N frequency components $f(k) = \frac{kf_s}{N}$, where f_s is the sampling frequency. It's important to note that when the DFT inputs are real numbers, which is always the case with audio signals, the magnitude of the DFT output is symmetric, since $X(N-k)$ is the complex conjugate of $X(k)$. Consequently, to obtain the spectrum of the signal $x(n)$ only the computation of the first $N/2+1$ values of $X(k)$ is needed.

The DFT algorithm has a complexity of $O(N^2)$, which is quite expensive. Therefore, it's never computed in practical applications. What is commonly used is a fast implementation of this algorithm, called *fast Fourier transform* (FFT). There are many variations of the FFT, and the details on how it works are out of the scope of this thesis. However, it's interesting to note that it allows to reduce the complexity of the DFT from $O(N^2)$ to $O(N \log N)$.

Short-time Fourier transform

Despite its importance, the DFT alone is not very useful to analyze audio signals with frequencies that change over time, such as speech signals. This is because the DFT is computed across the entire time interval of the signal, and it returns the frequency components averaged over all the duration. When dealing with a speech signal, it's crucial to know at what time a particular frequency component is present. The *Short-time Fourier*

transform (STFT) is the solution to this problem. The main idea is to divide the signal into time chunks, called frames, and then apply the DFT (in practice, the FFT) to each one of them. The mathematical formulation is [16]

$$S(m,k) = \sum_{n=0}^{N-1} x(n+mH) \cdot w(n) \cdot e^{-i2\pi nk/N} \tag{2.3}$$

where

- $w(n)$ is a window function, which multiplied by the original signal returns a fixed time chunk of that signal;
- m is the frame index;
- N is the total number of samples in a frame;
- H is the *hop size*.

The hop size represents the number of samples that correspond to a window slide. Intuitively, one might think that the hop size is equivalent to the frame size. In reality, the frames overlap, thus the hop size is smaller than the frame size. This is because the commonly used window function is not a rectangular one, but a bell shaped curve. An example is the *Hann function*, formulated as $\frac{1}{2} - \frac{1}{2}\cos\left(\frac{2\pi n}{N}\right)$ [16]. This window function smoothly tapers the signal at the beginning and end of a frame, in order to avoid discontinuities that could lead to spectral leakage [17]. This phenomenon occurs when the processed signal is not an integer number of periods. After applying the Fourier transform, the discontinuities appear as high frequency components which are not present in the original signal. The STFT process is illustrated in Fig. 2.4.

2.1.4 The Mel spectrogram

As it's been discussed, the STFT is a function $S(m,k)$ of both time, represented by the frame index m, and frequency, where k represents a frequency bin. Therefore, the computation of the STFT returns a matrix of complex numbers. Taking the square magnitude of the STFT results in a matrix of real numbers that can be plotted using a heatmap; this plot is called *spectrogram*. Each point on the plot represents the intensity of a given frequency at a particular time. According to the *Weber–Fechner law*, the human ear interprets the intensity of sound logarithmically rather

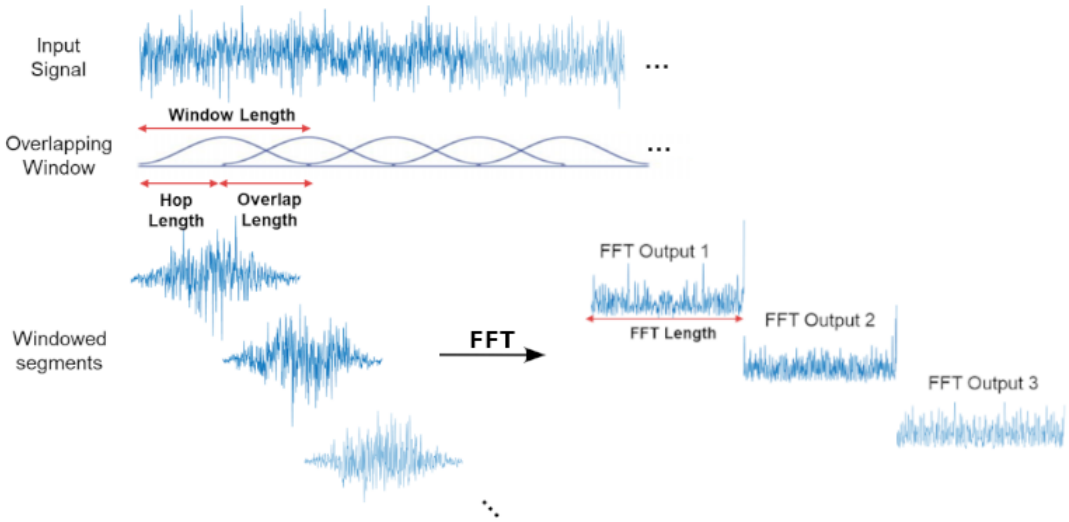


Figure 2.4: FFT applied to overlapping frames of a signal. Image adapted from [4].

than linearly. Hence, when plotting a spectrogram, a logarithmic scale is used for the amplitude, commonly represented in decibels (dB). However, this is not the full story. In fact, also the frequencies are perceived in a logarithmic way. Humans are better at detecting differences in lower frequencies rather than in higher frequencies. For this reason, after conducting a series of experiments on human listeners, the Mel scale has been developed [18] with the following formulation:

$$m = 2595 \log_{10} \left(1 + \frac{f}{700}\right) \tag{2.4}$$

This formula converts a frequency from Hertz to the Mel scale. Frequencies that are equidistant on the Mel scale sound equidistant to us. To generate the Mel spectrogram, the lowest and the highest frequencies are converted to the Mel scale, using the formula above. Then, this range is separated into n evenly spaced point, called Mel bands [19]. Those points are then converted back from Mel to Hertz and rounded to the nearest frequency bin. A triangle filter is created for each Mel band; the lower end of the filter corresponds to the center of the previous Mel band, and the higher end is on the center of the next Mel band. This process generates the entire Mel filter bank, represented in Fig. 2.5.

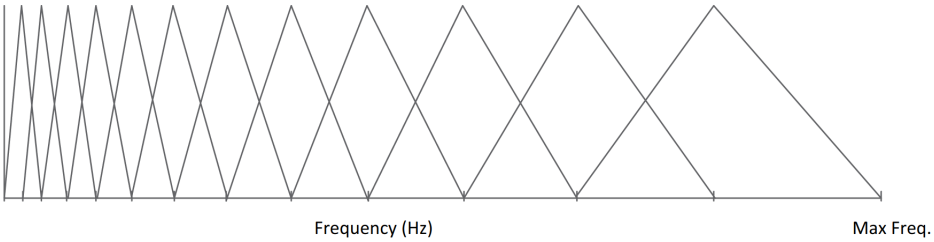


Figure 2.5: The Mel filter bank visualized. It's possible to see how the filters are more spread out as the frequencies get higher, similar to the way humans perceive them.

Finally, the Mel filter bank is applied to the spectrogram, by matrix multiplication of the Mel filter bank matrix and the spectrogram matrix. The end result of these transformations is the Mel spectrogram, which allows to visualize sound as it's perceived by the human ear. This representation is a central point in many of the Text-to-Speech models that will be presented in the following chapters.

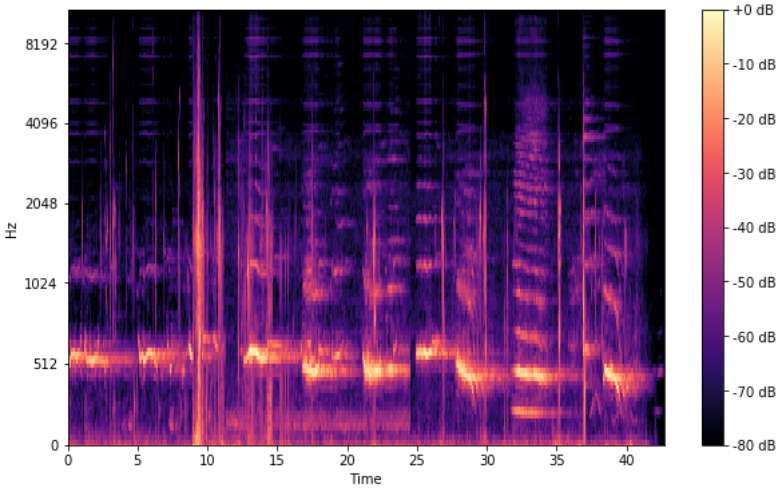


Figure 2.6: A plot of the Mel spectrogram. Image from [5].

2.2 History of TTS

Today, neural networks are the predominant technology for speech synthesis, since they achieve state-of-the-art results that are unmatched by

any other technique. Previously, there were different approaches that can be summarized into four main categories: articulatory synthesis, formant synthesis, concatenative synthesis and statistical parametric speech synthesis. An overview of these techniques is useful to understand what are the main stumbling blocks in TTS, some of which are still not completely resolved. Most of the information presented in this section is gathered from the remarkable paper: "A Survey on Neural Speech Synthesis", by Microsoft Research Asia [6].

2.2.1 Articulatory synthesis

This is the oldest approach to speech synthesis, which consists in a simulation of the human vocal tract. The first example in history is Wolfgang von Kempelen's speaking machine [20]. It was a mechanical machine, played by a person, that could produce vowels and some consonants. Trying to imitate the way humans generate speech doesn't work well in practice, therefore this technique was replaced by formant and concatenative synthesis.

2.2.2 Formant synthesis

This is a rule-based type of synthesis that generates an artificial waveform using additive synthesis, which works by adding sine waves together, and an acoustic model that varies parameters like fundamental frequency, noise and voicing over time. [6]. Formant synthesis produces intelligible speech, even though it doesn't sound natural. The main advantage is that it doesn't require a lot of computational resources, hence it works well in embedded environments that are often limited in terms of memory, processing and power. A popular example is *DECtalk*, the speech synthesizer used by Stephen Hawking [21], based on Dennis Klatt's research [22].

2.2.3 Concatenative synthesis

Concatenative synthesis uses a database of pre-recorded voice segments and the final sentence is generated by concatenation of those segments. The segments can be phonemes, diphonemes (pair of phonemes), triphonemes, syllables, words or even sentences. This approach should result in a natural sounding speech, since it makes direct use of unmodified pieces

of speech recorded from a speaker. However, the concatenation process is problematic since the discontinuities between the fragments can lead to artifacts [23]. Moreover, it requires a large database of recordings.

2.2.4 Statistical parametric speech synthesis

In statistical parametric speech synthesis (SPSS), the waveform is not generated by concatenation of speech segments, but it comes from acoustic parameters learned from the training data. The common pipeline consists of three components [6]: a text analysis module, which performs preprocessing on the input text data, like normalization, grapheme-to-phoneme conversion and tokenization; an acoustic model, usually a Hidden Markov Model (HMM); a vocoder, that synthesizes a waveform starting from the acoustic model parameters. The database, in this case, is used only for training. During synthesis, the acoustic parameters are generated from a statistical model, like a HMM. Those parameters can be modified in order to change speaking style and voice characteristics. The advantages are that it doesn't require a database as big as the one used for concatenative synthesis and the generated audio sounds more natural. The problems are the low intelligibility and the fact that it's still clearly different from natural human speech [6].

2.2.5 Neural synthesis

As the name suggests, neural synthesis employs neural networks to generate linguistic and acoustic features. Over the last decade, this has become the standard for speech synthesis, since it's able to generate natural sounding voices which are almost indistinguishable from human samples. The disadvantage is the cost: training requires big annotated datasets and expensive computational resources; also, the process of generating a dataset is slow and manual, since it requires many hours of audio with the corresponding transcriptions. This is a problem for low resource languages, considering that the dataset samples need to have a high audio quality. A noisy dataset makes training harder and also generates poor sounding speech during inference. These issues make it hard for neural TTS model to be deployed for embedded or IoT devices.
Figure 2.7 shows an overview of the main neural TTS models developed in the latest years. The next section will cover the high level pipeline of

neural TTS systems.

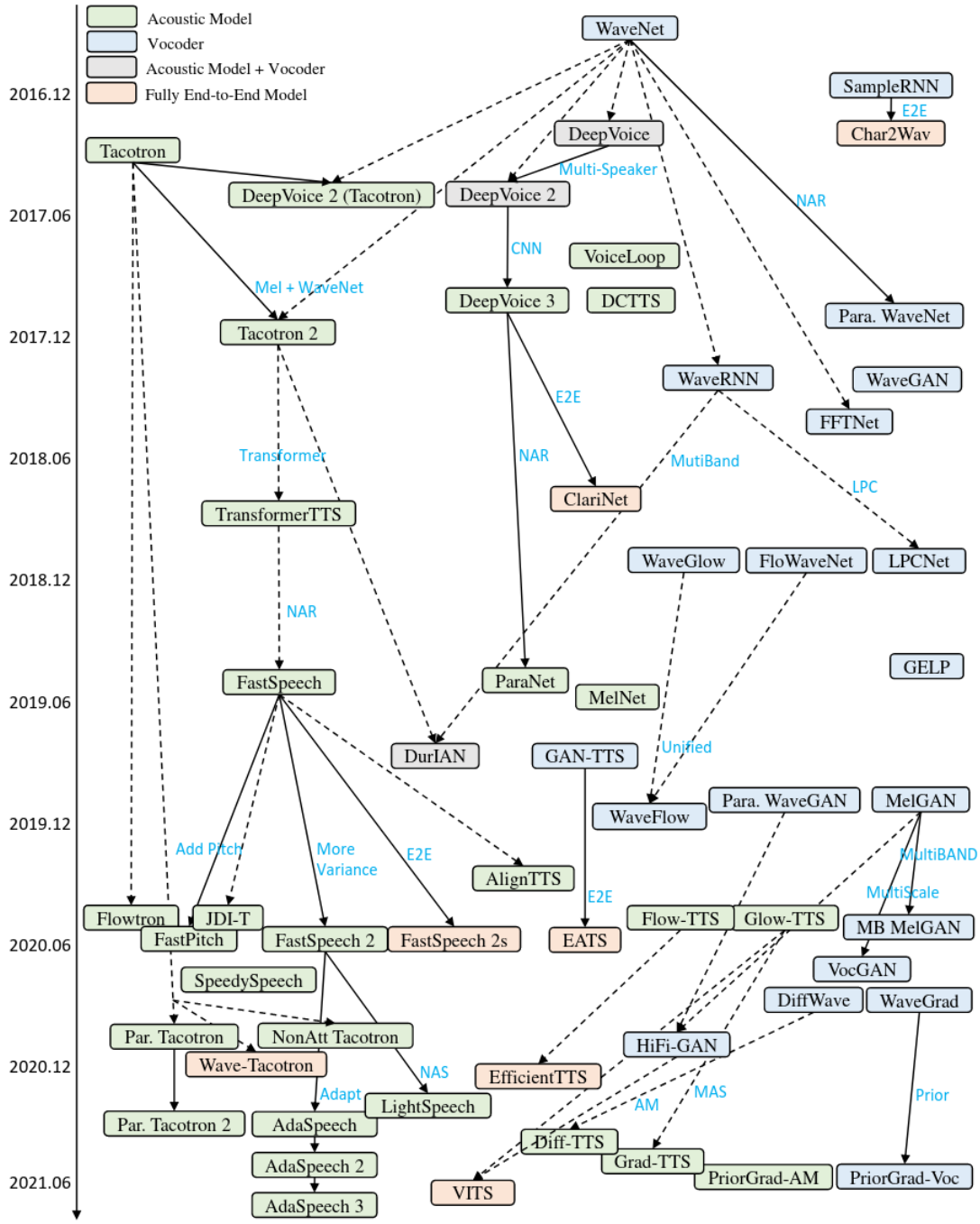


Figure 2.7: "The evolution of neural TTS models." [6]

2.3 Neural TTS

Modern TTS architectures are composed of several modules. The most common pipeline is the following:

- **Text normalization**: the first component in a TTS pipeline performs text preprocessing, converting the raw input text into a representation which is understandable by the model. For example, it converts numbers, symbols and currency into words and it expands abbreviations. It's usually implemented as a rule based system, using, for example, weighted finite-state transducers [24]. The implementation is different for every language.
- **Grapheme to phoneme conversion**: this step is not always performed, but it usually facilitates the learning process, especially for words that are written the same but pronounced differently (for example, in Italian, "ancora" is pronounced with a different accent and has a different meaning depending on the context). The characters (graphemes) are converted into phonemes, the smallest units of sound in a particular language. Today, this step is done using neural networks.
- **Spectrogram synthesis**: an acoustic model synthesizes a Mel spectrogram from the input graphemes/phonemes. This is a sequence-to-sequence problem where inputs and outputs have different lengths, since there are more output spectrogram frames than input characters. Usually, the first module of an acoustic model converts the input graphemes/phonemes into numbers (or vectors of numbers) that are understandable by the neural networks, by generating a character embedding for each character of the alphabet. There are two main families of acoustic models: autoregressive models and parallel models. Autoregressive models, such as Tacotron [25], were the first really good neural TTS models. In autoregressive generation, the output at the current step depends on the outputs generated at the previous steps. This type of sequential generation is usually modeled with recurrent neural networks (RNNs). The downsides of autoregressive models are:

  1. slow training and inference speed, since they can't fully take advantage of parallel GPU architectures;

2. RNNs can't model long term dependencies, due to vanishing gradients;
3. they suffer from robustness issues. Due to their sequential nature, errors tend to propagate in subsequent steps.

Tacotron uses an encoder-decoder architecture with attention. The attention is used by the decoder to determine which characters to consider when generating spectrogram frames. Tacotron predicts a linear spectrogram and then it uses the Griffin-Lim algorithm followed by an inverse short-time Fourier Transform to generate the waveform. Tacotron 2 [26] uses a recurrent sequence-to-sequence feature prediction network with attention to predict a sequence of Mel spectrogram frames, and a modified WaveNet (neural vocoder, see next point) that generates a waveform starting from the Mel spectrogram. Many improved models have since been realeased, starting from the Tacotron architecture, as it can be seen in Fig. 2.7.
Parallel models use an explicit duration predictor, instead of relying on the attention mechanism to implicitly determine the duration, in order to predict spectrogram frames in parallel. An example is the FastSpeech [27] series, based on Transformers. This type of architecture will be analyzed in detail in chapter 3.

- **Waveform generation**: models that generate a waveform starting from acoustic features are called vocoders. There are many different categories, such as autoregressive vocoders, Flow-based vocoders, GAN-based vocoders, VAE-based vocoders and Diffusion-based vocoders [6]. The first neural vocoder was WaveNet [8], based on an autoregressive CNN. The original WaveNet was conditioned on linguistic features, however it can be adapted to condition on Mel spectrograms, as it was done in Tacotron 2 [26]. It was inspired by the PixelCNN architecture [28]. Each audio sample x_t of the waveform $x = x_1, ..., x_T$ is conditioned on the samples at all previous timesteps:

$$p(x) = \prod_{t=1}^{T} p(x_t | x_1, ..., x_{t-1}) \tag{2.5}$$

The conditional probability distribution is modeled by a stack of convolutional layers followed by a softmax. It uses dilated causal convolutions to increase the receptive field of the filter. It uses residual blocks [29] and skip connection to allow the training of much deeper

models. When it was released, it achieved the highest Mean Opinion Score (see chapter 5) in the history of TTS, however it suffers from slow inference speed. In chapter 3 I will present a GAN-based vocoder used in the architecture of the final model.

To conclude this section, it's worth mentioning that the most recent trend in neural TTS is to move towards end-to-end architectures. Instead of the two-stage pipeline described above, the goal is to have a single model that can generate a waveform starting from input text. The most popular examples are FastSpeech 2s [30], EATS [31] and VITS [32].

Chapter 3

Approach and methodology

3.1 Dataset

Dataset quality is one of the most important aspects for a neural speech synthesizer. Feeding noisy samples during training would make the model pick up the noise and reproduce it during synthesis, in the best case. In the worst case, it would make it almost impossible for the network to learn anything at all.
Finding a high quality dataset turned out to be a hard process, due to the lack of available options. The main source comes from the LibriVox [33] project, which is a catalogue of free public domain audiobooks, read by volunteers. The major downsides are that the audio quality of those recordings is often inadequate and the vocabulary is somewhat anachronistic, since public domain books are at least 70 years old [34].
The most popular dataset in TTS research is the LJ Speech [15] dataset, which is derived from LibriVox's catalogue. It's composed of 13100 English audio clips ranging between 1 and 10 seconds, split up based on the silences in the recordings. The sampling rate of the audios is 22050 Hz. Each audio clip has a corresponding transcription, and all the transcriptions are gathered into a file, one record per line, with the following format:

```
<path_to_audio_clip>|<text>|<normalized_text>
```

The normalized text is obtained by expanding abbreviations, as well as

writing numbers and symbols as full words. The normalization process will be examined in depth in chapter 4. Since LJ Speech has become a standard, it can be used as a baseline for comparison, in order to assess the quality of other TTS datasets. It's important to note that even though the LJ Speech is a research standard, it doesn't imply that it's suitable for training a high quality TTS model for a commercial service. Its main advantage, besides being of public domain, is the amount of hours of noise-free speech; moreover, a sampling rate of 22050 Hz provides a good balance between audio quality and training speed. However, if training speed was not a concern, using a sampling rate of 44100 Hz would result in higher quality speech synthesis.
The goal of this work was to generate podcasts in Italian, hence a dataset in the Italian language was needed. There were only two available options: Mozilla Common Voice [35] or the M-AILABS Speech Dataset [36]. The former contained very noisy and low quality audio clips, so it was immediately discarded; the only choice left was the M-AILABS dataset. This is a large multilingual dataset derived from LibriVox, which includes the Italian language. The Italian dataset is divided in three: a multi-speaker dataset, a single-speaker male dataset and a single-speaker female dataset. The male dataset sounded the best, therefore it ended up being the one used for training the TTS models. The recordings are from 5 different audiobooks, and the transcriptions follow the LJ Speech format. However, the sampling rate is 16000 Hz, which means faster training, but also less information.

One useful metric for speech datasets analysis is the signal-to-noise ratio (SNR) of the audio recordings. It's defined as [37]

$$SNR_{dB} = 10 \log_{10} \frac{P_s}{P_n} \tag{3.1}$$

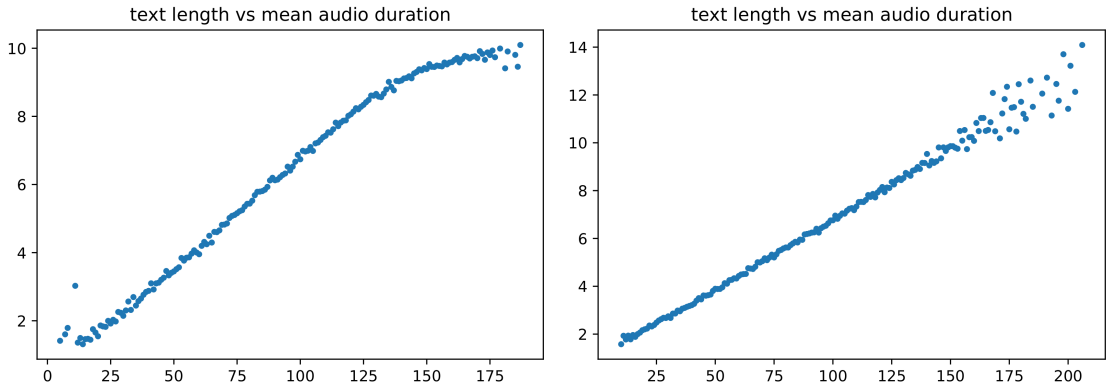
where P_s is the power of the signal and P_n is the power of the noise. As explained before, it's important to make sure that the data has as little background noise as possible. Calculating SNR for each recording can be helpful to identify noisy samples and remove them from the dataset. In addition, it's possible to compare the datasets by comparing the average SNR of the worst samples. In order to do so, I used a tool [38] that implements the WADA-SNR [39] algorithm, which is used to determine the SNR of speech signals. The values are presented in table 3.1.

Dataset	$SNR_{100}(dB)$
LJ Speech	16.20
Common Voice	-2.79
M-AILABS	20.23

Table 3.1: Comparing the average of the 100 worst SNRs, for each dataset.

I computed the average SNR of the worst 100 samples and not of all the samples because I found that, with this algorithm, only low values of SNR were an actual indication of the presence of noise in the recordings. In fact, there are no noisy samples in LJ Speech and M-AILABS, but they have SNRs ranging from 16 dB to 100 dB. However, it correctly detected the noise in the Common Voice recordings, which present a negative value, meaning that the power of the noise is higher than the power of the speech signal.

Mozilla's TTS library [40] offers a notebook for the analysis of speech datasets, which I used to visualize some interesting statistics. In the next figures LJ Speech will be compared with M-AILABS.

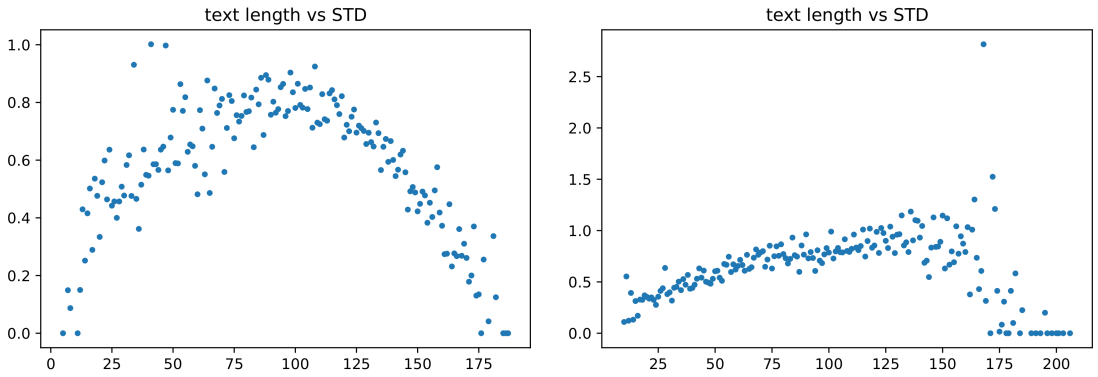


(a) LJ Speech (b) M-AILABS

Figure 3.1: Plots of the character length (x axis) against the average duration in seconds (y axis).

In order to synthesize speech at a consistent speed, there needs to be a linear correlation between the number of characters and the mean audio duration. In figure 3.1b, M-AILABS shows an almost perfectly linear

pattern until 150 characters, which correspond to about 10 seconds of audio. However, it becomes inconsistent with longer sequences. This plot highlighted the presence of a small subset of outliers that were longer than 10 seconds. Consequently, they have been removed from the dataset, in order to reduce the duration inconsistencies and possibly avoid out of memory errors during training.

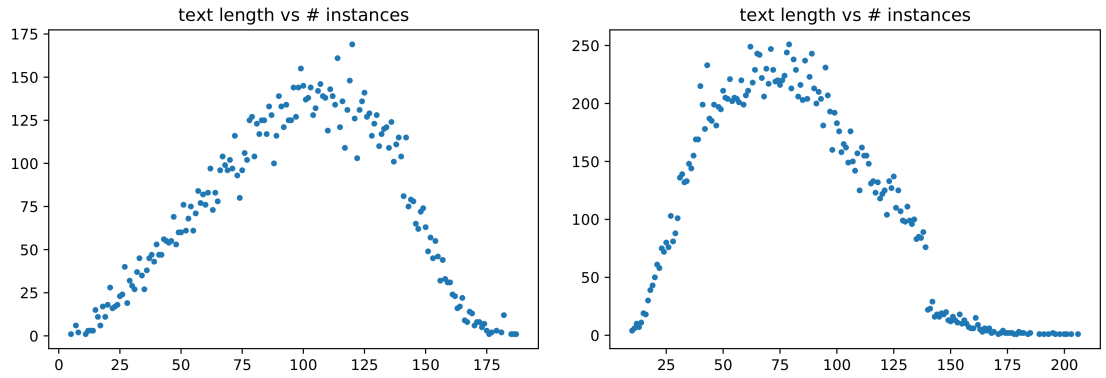


(a) LJ Speech (b) M-AILABS

Figure 3.2: Plots of the character length (x axis) against the std of the duration in seconds (y axis).

To ensure consistent speech, it's also necessary to have a low standard deviation for the duration of the samples. In Fig. 3.2b there is a glaring issue: the point on the top has a length of 168 characters and a standard deviation of 2.8 seconds. Upon further investigation, there are only two samples with 168 characters; one is 9.2 seconds long and the other one is 14.9 seconds. LJ Speech plot looks better in this case, having most of of the points with a standard deviation below 1.0.

Figure 3.3 compares the number of samples for each character length. While the LJ Speech plot follows a smooth Gaussian distribution, the M-AILABS shows the same issue as before with longer characters. Here the outliers are clearly distinguishable, since the plot shows that there are very few samples above 175 characters for the Italian dataset. This was also visible in the previous plot, where those points have a standard deviation of 0, suggesting that there is only one instance of them at that character length.



(a) LJ Speech (b) M-AILABS

Figure 3.3: Plots of the character length (x axis) against the number of corresponding samples (y axis).

3.2 VITS

As discussed previously, training a neural TTS model requires a lot of computational resources. Therefore, I couldn't just take every possible state-of-the-art model and train it until I found the perfect one for the task. I had to make some prior assumptions, excluding some architectures based on the knowledge acquired during the research. I decided to exclude autoregressive models based on RNNs, such as Tacotron 2, because of the slow training and inference speed. Today's GPUs are meant for parallel generation, while autoregressive models are based on sequential generation of the spectrograms, one frame at a time. Since the final objective is to build a TTS API for podcast synthesis, inference speed is one of the priorities. Another problem of autoregressive models is error propagation: due to the sequential nature, an error at a certain step can disrupt the synthesis of the subsequent steps, given that, during inference, the inputs are the previously predicted values.

Traditional neural TTS systems are composed of two main stages: the first one generates a spectrogram from input phonemes/graphemes, the second one generates a waveform starting from the spectrogram. That's done by two separate models. In the last two years, research has started to move towards an end-to-end approach, where a waveform is generated directly from text. The first model that I trained is a parallel end-to-end system, called VITS (Variational Inference with adversarial learning for

end-to-end Text-to-Speech) [32].

3.2.1 Architecture

Overview of Variational Autoencoders

The architecture, as described by its authors [32], is fairly complex. However, at its core it uses a variational autoencoder (VAE). Autoencoders are neural networks that present an encoder-decoder architecture and their objective is to learn a low dimension latent space that can retain as much information about the input data as possible. The encoder is a block that takes an input and compresses it into a latent space; the decoder has to reconstruct the original input starting from the compressed representation. During training, the reconstruction loss between the original input and the decoder output is computed, then the error is backpropagated to the previous layers. In a variational autoencoder, instead of encoding the inputs as points in a latent space, they are encoded as a Normal distribution. The encoder predicts two separate vectors, one representing the mean and the other one the standard deviation. Then, a random point is sampled from this distribution and fed to the decoder. The training loss function of a VAE is described as

$$\mathcal{L}(\theta, \phi) = -\mathbb{E}_{q_\phi(z|x)} \left[\log p_\theta(x|z)\right] + D_{KL}(q_\phi(z|x) \parallel p(z)) \tag{3.2}$$

The first term is the reconstruction loss, while the second term is the Kullback-Leibler divergence between $q_\phi(z|x)$, which is the encoder distribution, and $p(z)$, which is a standard Gaussian. This is a regularisation term that forces the encoder distribution to be as similar as possible to $N(0, 1)$, thus preventing the neural network from turning the standard deviation towards 0 and becoming a standard autoencoder. The KL divergence is a measure of the divergence between two probability distributions; the smaller it is, the more similar the distributions are. Thanks to this term, similar input points end up close to each other in the latent space. The negative of this loss is called Evidence Lower Bound (ELBO); minimizing the loss corresponds to the maximization of the ELBO.

VITS

The final loss of the model is a combination of a VAE and a GAN (Generative Adversarial Network) [41], defined as [32]

$$L_{vae} = L_{recon} + L_{kl} + L_{dur} + L_{adv}(G) + L_{fm}(G) \tag{3.3}$$

The first term is the reconstruction loss between the predicted spectrogram and the ground truth: $L_{recon} = \|x_{mel} - \hat{x}_{mel}\|_1$. The predicted spectrogram $\hat{x}_{mel}$ is obtained by upsampling the latent variables z to the waveform $\hat{y}$, through a decoder, then $\hat{y}$ is transformed to $\hat{x}_{mel}$.
The second term is the KL divergence, computed as

$$L_{kl} = \log q_\phi(z|x_{lin}) - \log p_\theta(z|c_{text}, A) \tag{3.4}$$

The symbol c_{text} expresses the phonemes extracted from the text, while A is the alignment. The alignment is a monotonic matrix with dimensions $|c_{text}| \times |z|$ that represents how long each phoneme is pronounced. The duration d_i of each input token is calculated by summing all the columns in each row of the estimated alignment $\sum_j A_{i,j}$. This is used to train a stochastic duration predictor, whose loss is represented by L_{dur}. During inference, the stochastic duration predictor samples the durations for each phoneme from random noise, then it converts them to integers. This will make the model synthesize speech with different rhythms from the same input text. The last two terms of equation (3.3) pertain the adversarial training process. GANs, in a nutshell, use a neural network as a loss function. During training, a discriminator network has to distinguish when the input comes from the ground truth waveform, as opposed to when it comes from the output generated by the decoder G. The loss of the discriminator is $L_{adv}(D) = \mathbb{E}_{(y,z)}\left[(D(y)-1)^2 + (D(G(z)))^2\right]$; the loss of the generator is $L_{adv}(G) = \mathbb{E}_z\left[(D(G(z))-1)^2\right]$. The discriminator is used only during the training process and dropped when performing inference. $L_{fm}(G)$ is the feature matching loss, which, as written by the authors, *"can be seen as reconstruction loss that is measured in the hidden layers of the discriminator suggested as an alternative to the element-wise reconstruction loss of VAEs"*.

I voluntarily skipped some details, such as how the alignment matrix is estimated and how each module is implemented, in order to avoid going

out of scope and to make this presentation less burdensome. For the interested reader, I suggest referring to the original paper.

3.2.2 Training

I used the PyTorch implementation provided by the authors of the model [42]. The dataset has been splitted into 13775 samples for training, 3444 for validation and 3444 for testing. The grapheme-to-phoneme conversion wasn't embedded into the model, so I used an open source library [43] to convert the transcriptions into phonemes following the IPA (International Phonetic Alphabet) [44] convention. I used the AdamW optimizer with $\beta_1 = 0.8$, $\beta_2 = 0.999$ and $\epsilon = 10^{-9}$, a batch size of 32, and for the FFT I used a standard window size of 1024, hop size of 256 and 80 Mel frequency bins. The learning rate starts at 10^{-3} and is decayed exponentially by a factor of $0.999^{1/8}$ every epoch. The authors trained their model for 800K steps, using 4 NVIDIA V100 GPUs. However, I didn't have access to such resources. I trained this model on Google Colab , which provides a NVIDIA Tesla T4 for free for a maximum of 12 hours. Initially, I wanted to train it for a number of steps comparable to the paper, however, it took around 20 days just to get to 260K steps. To be fair, they were not continuous, since Colab locks GPU access to an account for some hours after using it for a large chunk of time. Nevertheless, by looking at the validation loss and listening to the audio samples generated by the model, most of the learning occurred in the first 50K steps. Afterwards, the loss went down very slowly with minimal improvements on the perceived audio quality. I will analyze the results and discuss the evaluation metrics in chapter 5.

3.3 FastPitch

While VITS (3.2) was training on Google Colab, I decided to experiment with a completely different model. I needed something that was on par with the state-of-the-art, while providing a fast inference time and possibly a faster training speed with respect to VITS. FastSpeech [27] was one of the first models that allowed both parallel training and inference, and, as it can be seen in Figure 2.7, several models based on FastSpeech have been subsequently developed. In the end, I chose FastPitch [45], since it's

well integrated in the NeMo framework [46]. NVIDIA NeMo is an open source framework that allows to easily build speech AI models for Natural Language Processing, Automatic Speech Recognition and Text-to-Speech. It also provides pre-trained models, even though they are only available in the English language. FastPitch is not an end-to-end model, like VITS, but it's an acoustic model. It synthesizes Mel spectrograms from input text, so it takes care only of a part of the pipeline. Another model, a vocoder, will be used to convert FastPitch's output to the final waveform. This model is described in section 3.4. Below, a high level representation of the full pipeline.

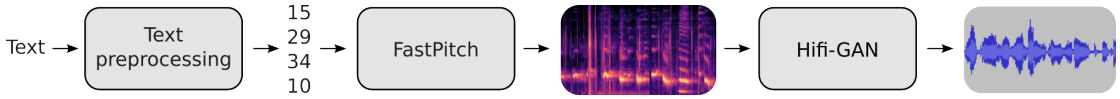


Figure 3.4: Pipeline for converting input text into a waveform.

3.3.1 Architecture

The most important component of the architecture is the Feed-Forward Transformer block (FFT). Transformer models were first introduced in the paper *"Attention is all you need"* [47] and they have replaced LSTM-based models for most applications. Nowadays, they are extensively used in natural language processing models, like BERT [48] and GPT [49]. The Transformer is an encoder-decoder network based on the attention mechanism, which allows the neural network to understand relationships in the input data. The main advantages of Tranformers over Recurrent Neural Networks and LSTMs are their parallel nature and the self-attention mechanism. The fact that they are not sequential means that they can process entire sentences in parallel, making them faster to train, while RNNs and LTSMs process input sentences word by word. The self-attention mechanism allows Transformers to understand the relationships in the inputs and what parts of the sentence to focus on, by using score matrices. Since they don't rely on a hidden state to capture the dependencies with previous words, they don't forget past context like RNNs do.
Coming back to FastPitch, a Feed-Forward Transformer block contains a Multi-Head attention module and a 1D convolution. The Multi-Head attention consists of a stack of parallel self-attention layers, called "heads".

There are two FFT stacks. The first stack is composed of N FFT blocks that operate on the input tokens, while the second stack contains T FFT blocks that operate on the output spectrogram frames. The first FFT stack produces a hidden representation h, used to make predictions about the average duration and pitch of each input phoneme. The hidden representation is fed into the duration predictor and the pitch predictor modules:

$$\begin{aligned} \hat{d} &= DurationPredictor(h), \qquad d \in \mathrm{N}^n \\ \hat{p} &= PitchPredictor(h), \qquad p \in \mathrm{R}^n \end{aligned} \tag{3.5}$$

Then, the pitch is projected to match the dimensionality of the hidden representation $h \in \mathrm{R}^{n \times d}$ and added to h. The resulting sum g is discretely upsampled according to the duration values coming out of the duration predictor and passed to the last FFT stack, which produces the output Mel spectrogram sequence $\hat{y}$:

$$\begin{aligned} g &= h + PitchEmbedding(p) \\ \hat{y} &= FFT([\underbrace{g_1, ..., g_1}_{d_1}, \underbrace{g_2, ..., g_2}_{d_2}, ..., \underbrace{g_n, ..., g_n}_{d_n}]) \end{aligned} \tag{3.6}$$

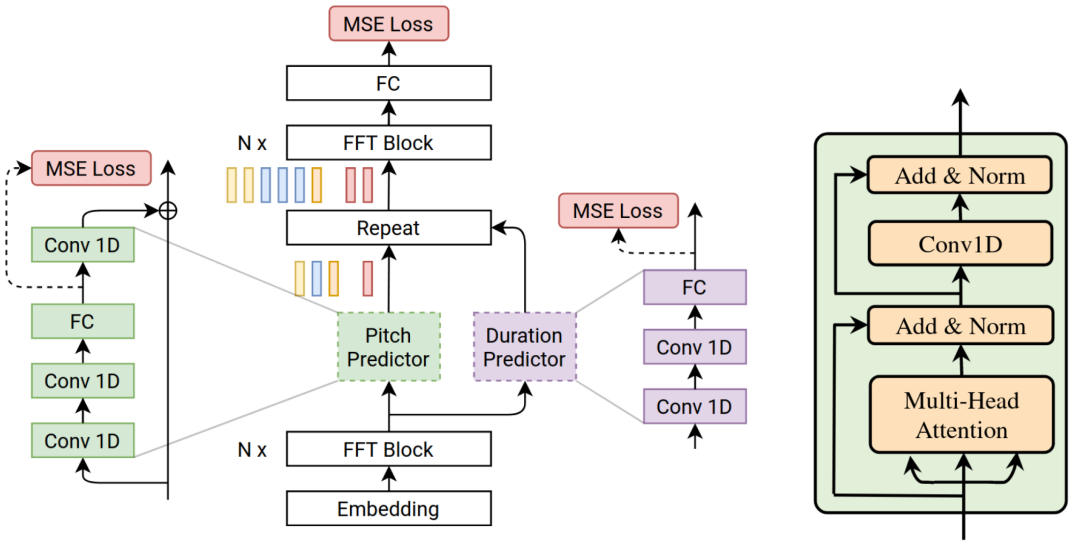
During training, ground truth p and d are used, while the predicted $\hat{p}$ and $\hat{d}$ are used during inference. The final loss is simply a Mean Squared Error between the ground truth values and the predictions:

$$\mathcal{L} = \|\hat{y} - y\|_2^2 + \alpha\|\hat{p} - p\|_2^2 + \gamma\|\hat{d} - d\|_2^2 \tag{3.7}$$

According to the paper, ground truth durations are estimated from the training dataset using a pre-trained Tacotron 2 model, while the ground truth pitch values are obtained through acoustic periodicity detection [50]. Figure 3.5 illustrates the high level architecture of FastPitch and the FFT block.

3.3.2 Training

As previously mentioned, I used the open source NVIDIA NeMo [46] repository, which has an implementation of FastPitch. This model required a different format (JSON) compared to the standard LJ Speech one, so I had to reformat the dataset accordingly. The model also required the



(a) FastPitch [45]. (b) FFT block [27].

Figure 3.5: Architecture of FastPitch.

ground truth duration of each transcript, so I iterated over the entire dataset once and I computed the durations using *librosa* [51], a popular audio processing library. The final format of each transcript is

```
{
    "audio_path": "<audio_path>", "text": "<text>",
    "normalized_text": "<normalized_text>",
    "duration": <duration>, "is_phoneme": <0/1>
}
```

The model was trained on HPC@POLITO [52], a high performance computing cluster of *Politecnico di Torino*. The cluster utilizes the SLURM [53] scheduler to manage the requests coming from different users. A user can submit a job by means of a sbatch script, where he has to specify how long the job will run and the hardware resources required, such as number of CPU cores, amount of memory, how many GPUs and what partition to run the job in. Furthermore, one can also specify a log file and an email in order to receive notifications from SLURM about the status of the submitted job. After the submission, the job is added to a queue with a priority value assigned. When the hardware resources become available and there are no higher priority jobs, the submitted job is finally run.

The cluster uses Intel Xeon Gold 6130 2.10 GHz CPUs and NVIDIA Tesla V100 GPUs. Even though multi-GPU training is supported, I avoided it in order to reduce the queue waiting times. The standard configuration I used was 1 GPU and 12 CPU cores, in order to speed up the data loading by parallelizing it in multiple processes, as supported by PyTorch's DataLoader. Since the NeMo library required some sotware packages that were not installed in the cluster, I had to run my jobs inside a Singularity [54] container. First, I built a Singularity image that encapsulate the OS environment and all the packages required by NeMo. Then, given that an image consists just of a single file, it was sufficient to copy it to my home folder inside the cluster and specify in the sbatch script to run my job inside a container, generated from an instance of that image.
As for the training hyperparameters, I've used the AdamW optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$, a batch size of 32 and the Noam learning rate scheduler, with a starting learning rate of 10^{-3}. The Noam scheduler increases the learning rate linearly for a number of warm-up steps (1000 in this case) and then decreases it proportionally to the inverse square root of the step number [47]. The FFT parameters consisted of a window size of 1024, hop size of 256 and 80 Mel bands. I trained it for more than 4M steps, stopping when the validation loss got stale. Figure 3.6 shows the Tensorboard plot of the total validation loss. Unfortunately, the logs of the first training run have been accidentally deleted, hence the plot starts at around 400K steps. The loss curve has different colors, where each colored segment represents a different run of the training job in the cluster. Since it wasn't possible to know in advance how long the training would have taken, I scheduled most of the jobs to last from 24 to 48 hours. Training checkpoints were saved every 5 epochs if there was an improvement in the validation loss, in order not to lose training progress at the end of each job.

3.4 HiFi-GAN

The model that I've chosen for the vocoder, which takes care of the last stage of the pipeline, is HiFi-GAN [55].

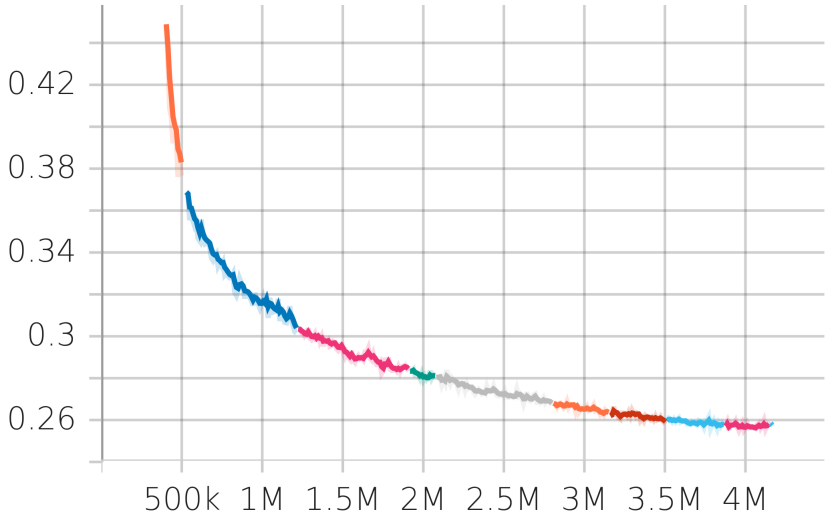


Figure 3.6: Validation loss of FastPitch at each training step.

3.4.1 Architecture

HiFi-GAN is a Generative Adversarial Network that consists of one generator and two discriminators. The generator is a convolutional neural network that takes a Mel spectrogram as input and upsamples it until the length of the output sequence matches the temporal resolution of the raw waveform. The upsampling is done through transposed convolutions. After the convolutional layer, there is a module that the authors called Multi-Receptive Field Fusion (MRF), which contains a stack of parallel residual blocks with different kernel sizes and dilation rates, in order to capture different patterns in the inputs coming from the transposed convolution layer. The MRF module returns the sum of the outputs of the residual blocks. The two discriminators used are called Multi-Period Discriminator (MPD) and Multi-Scale Discriminator (MSD). The Multi-Period Discriminator is composed of several sub-discriminators, each accepting samples of input audio spaced with period p. The periods are set to [2, 3, 5, 7, 11], so that each sub-discriminator captures different implicit structures. The 1D shaped input audio of length T is reshaped into a 2D array of size $(T/p, p)$, to which 2D convolutions are applied. The sub-discriminators are convolutional neural networks with a leaky ReLU activation function. While MPD operates on disjoint samples of raw waveforms, MSD operates on smooth waveforms. The Multi-Scale Discriminator has three sub-discriminators operating on different input

scales: raw audio, x2 average-pooled audio and x4 average-pooled audio. Average pooling is an operation used to downsample the input data; a filter slides over the inputs and computes the average of the values contained in the region covered by the filter. Each sub-discriminator is composed of a stack of strided and grouped convolutional layers with leaky ReLU activation. An illustrations of the sub-discriminators, taken by the original paper, is provided in Figure 3.7.

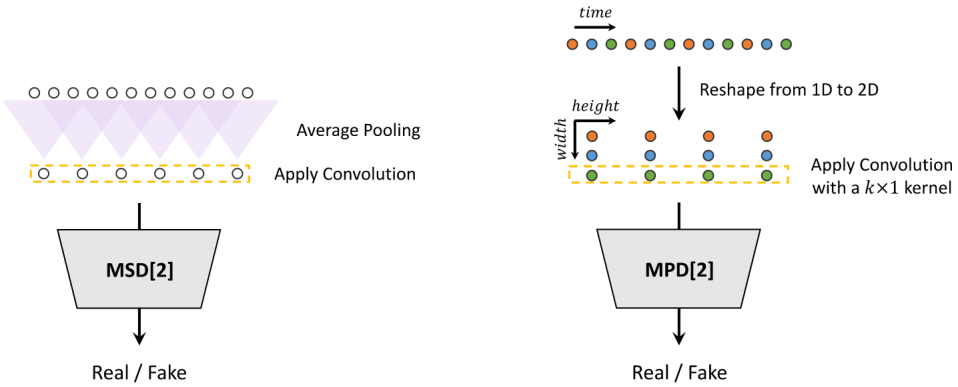


Figure 3.7: MSD on the left, MPD on the right.

The losses for the discriminators and the generator are

$$\begin{aligned} \mathcal{L}_{adv}(D;G) &= \mathbb{E}_{(y,z)}\left[(D(y)-1)^2 + (D(G(z)))^2\right] \\ \mathcal{L}_{adv}(G;D) &= \mathbb{E}_{z}\left[(D(G(z))-1)^2\right] \end{aligned} \tag{3.8}$$

where z is the Mel spectrogram and y is the ground truth waveform. The discriminators are trained to classify with 0 the samples coming from the generator, and with 1 the ground truth samples. The generator is trained to trick the discriminator by making it classify its outputs to be close to 1. In addition, a Mel spectrogram loss and a feature matching loss are added to the generator loss. The Mel spectrogram loss is the L1 distance between the Mel spectrogram of a waveform synthesized by the generator and the spectrogram of the ground truth waveform:

$$\mathcal{L}_{Mel}(G) = \mathbb{E}_{(y,z)}\left[\|\phi(y) - \phi(G(z))\|_1\right] \tag{3.9}$$

where ψ is the function that transforms a waveform into a Mel spectrogram. This loss helps the generator to synthesize more realistic waveforms.

The feature matching loss is the L1 distance between the hidden features of the discriminator when the input is the outcome of the generator as opposed to when it's the ground truth waveform:

$$\mathcal{L}_{FM}(G;D) = \mathbb{E}_{(y,z)}\left[\sum_{i=1}^{T}\frac{1}{N_i}\|D^i(y) - D^i(G(z))\|_1\right] \tag{3.10}$$

where T is the number of hidden layers and N_i the number of features in the i-th layer of the discriminator. The final losses for the generator and the discriminator are written as a summation of the contributions of all the K sub-discriminators:

$$\begin{aligned}\mathcal{L}_G &= \sum_{k=1}^{K}\left[\mathcal{L}_{adv}(G;D_k) + \lambda_{fm}\mathcal{L}_{FM}(G;D_k)\right] + \lambda_{mel}\mathcal{L}_{Mel}(G)\\ \mathcal{L}_D &= \sum_{k=1}^{K}\mathcal{L}_{adv}(D_k;G)\end{aligned} \tag{3.11}$$

3.4.2 Fine-tuning

Since the inputs of a vocoder are Mel spectrograms, it's not necessary to train it from scratch in the target language. I started from a pre-trained checkpoint, provided by NeMo, that was trained on the LJ Speech dataset and I re-trained it on the M-AILABS dataset, with a low learning rate. This process is called fine-tuning. The low learning rate is needed to avoid too large changes in the weights of the pre-trained model. Otherwise, the features learned from the original training would be disrupted.
Before launching the fine-tuning, I had to generate the input Mel spectrograms. That was accomplished by taking the last FastPitch checkpoint and using it to predict synthetic Mel spectrograms for each record in the dataset. However, the FastPitch checkpoint uses ground truth alignment and durations calculated from the dataset records instead of the predicted ones, to make sure that the Mel spectrograms used by Hifi-GAN have the same duration of the true spectrograms. The transcriptions were modified too, so that each one pointed to its corresponding Mel spectrogram:

```
{
    "audio_path": "<audio_path>", "text": "<text>",
    "normalized_text": "<normalized_text>",
```

```
    "duration": <duration>,
    "mel_filepath": "<path_to_mel_spectrogram>"
}
```

The hyperparameters were similar to those used for training FastPitch, except that in this case there was no learning rate scheduler, and the learning rate was set to 10^{-5}. In Figure 3.8 I plotted the total validation loss of the generator (3.11).

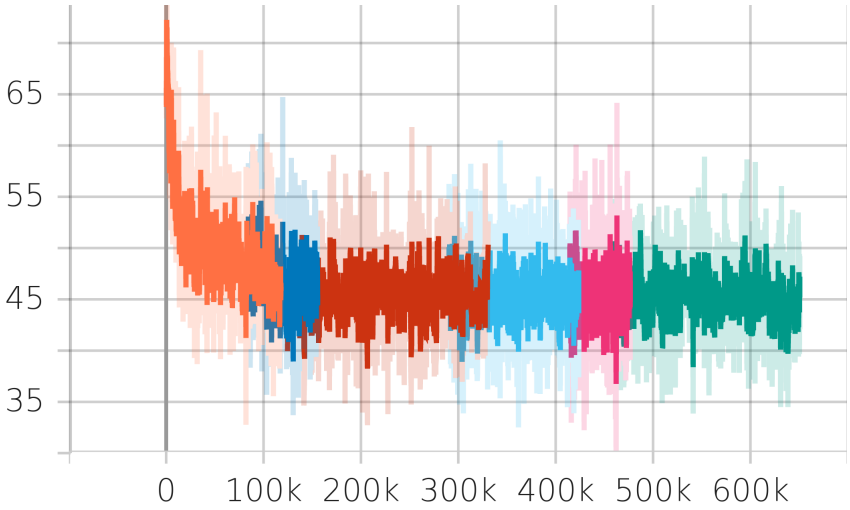


Figure 3.8: Total validation loss of the generator.

I fine-tuned it for around 650K steps. By looking at the plot of the validation loss, it may look like stopping at around 200K would have been enough. However, from a subjective evaluation, the output waveform sounded cleaner after 600K steps, despite the tiny improvements in the loss, while the previous checkpoints had some noticeable artifacts.
In chapter 5 I will compare the results coming out of this pipeline against the end-to-end VITS model (3.2). Before that, I will explain how I fine-tuned FastPitch to a new voice.

3.5 Transfer learning on an automatically generated dataset

Once the models were trained, it was possible to exploit a very powerful concept: transfer learning. It consists of taking a model, pre-trained on a

particular dataset, and re-train it on a new dataset, with the goal of transferring the knowledge learned by that model and apply it to a slightly different task. It's particularly useful when dealing with small datasets, since training a complex network from scratch with very few samples usually leads to severe overfitting. This phenomenon occurs when the network becomes very good at predicting training samples, but performs poorly with unseen data, like a student that has learned a subject by heart, without understanding its actual meaning. Instead, by using a pre-trained model and by lowering the learning rate, the original weights are changed just slightly, or "tuned", thus allowing the network not to lose its generalization capabilities. Another advantage is the decreased training cost, since it usually takes way less steps to fine-tune a pre-trained network on a new dataset rather than training it from scratch.
Unfortunately, in the TTS domain, pre-trained models are rare, with the exception of the English language. This scarcity of pre-trained models stems from a scarcity of datasets. Big tech companies have the resources to build high quality proprietary datasets, and they offer commercial TTS services in almost every language. However, public domain datasets mostly come from public domain audiobooks, sourced by LibriVox. This leads to, at least, three significant disadvantages:

1. absence of high quality datasets for most languages;
2. absence of any kind of dataset for the least common languages, the so-called "low-resource" languages;
3. archaic language of the public domain audiobooks.

By looking at the structure of the LJ Speech dataset, it's possible to see why datasets are such a rare resource. Building them is an expensive procedure. The first step, which can be done automatically, is to split the original audio file into segments of length ranging from 1 to 10 seconds. Then, if the audio file comes from an audiobook, the text has to be segmented into transcripts and aligned to the corresponding audio segments. According to the LJ Speech website [15], the text was matched to the audio manually, which means it was done by humans. Given that it contains 13100 clips, that's a very laborious task that not only requires a team of people and a considerable amount of time, but it's also prone to fatigue errors. An alternative is to start from the transcripts, then use a forced alignment tool [56] in order to determine, for each transcript, the

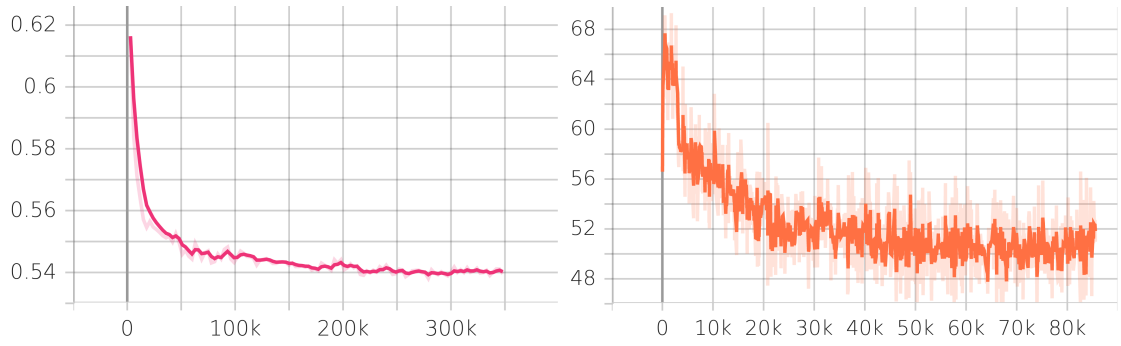
corresponding time interval of where it occurs in the audio. Once that the beginning timestamp and the end timestamp have been computed for each transcript, one could generate the clips by splitting the original audio according to the timestamps. This approach is certainly better than the manual one, but it requires a prior existence of the transcripts. In case one wanted to generate a dataset from a public speech or a podcast, he would still need to manually transcribe the entire recording. Instead, it would be ideal if we could automatically generate transcripts from the recordings.

At this point, I could fill two needs with one deed: the main goal was to fine-tune the models on a female speaker, but I also wanted to do it on an automatically generated dataset. During my research, I came across an automatic speech recognition model that had just been released in September 2022 by OpenAI, called Whisper [57]. It's based on an encoder-decoder Transformer architecture and it was trained on a large (680,000 hours) multilingual dataset. According to the paper, it achieves a human-level transcription accuracy and it's incredibly robust, even in the case of noisy data. OpenAI publicly released the code as well as pre-trained models of five different sizes. The only thing missing was the dataset. I couldn't find recordings of female speakers with decent quality on LibriVox, so I looked for other resources, until I found this [58] audiobook on the Liber Liber portal. In order to test Whisper's capabilities, I tried to generate some transcriptions of the first chapter, using the large model, and I compared them with the actual book. The results were utterly impressive, given that the model was even able to match correct punctuation and capitalization, with minimal mistakes. Finally, I wrote a small script to implement the following pipeline:

1. split the input audio into sentences below 10 seconds, where silences are detected. This was done using an audio manipulation library [59];
2. feed each sentence into Whisper to generate the corresponding transcription;
3. generate the normalized transcriptions, using the text normalizer described in chapter 4;
4. write the results into a file with the LJ Speech format.

I applied the above procedure to the first 10 chapters of the audiobook and I was able to automatically generate a dataset in a matter of minutes,

without any manual intervention; I just did a quality check of the transcriptions.
I fine-tuned the FastPitch model, previously trained on the M-AILABS dataset, on the new dataset using a learning rate of 2×10^{-4}. As it can be seen in Fig. 3.9a, the validation loss stabilized after around 300K training steps. I managed to fine-tune FastPitch on the new voice in a single 18-hour run on the hpc cluster. This clearly shows the benefits of transfer learning, especially when comparing this graph to the 4+ million steps required to train FastPitch from scratch (Fig. 3.6). For fine-tuning the Hifi-Gan model, I followed the exact same approach described in 3.4.2. I let it run for 24 hours, and the trend of the loss ended up looking similar to Fig. 3.8. After fine-tuning both models I was able to successfully generate speech with the new speaker's voice. I will show the comparison of the fine-tuned speech against the original in chapter 5.



(a) FastPitch. (b) Hifi-Gan.

Figure 3.9: Validation losses of FastPitch and Hifi-Gan during fine-tuning.

In this last section I explored the powerful results that can be obtained by combining cutting edge speech AI technologies. I also did some local testing on YouTube videos, and I was able to effortlessly generate datasets out of them. This shows how, as of today, it's possible to clone the voice of anyone who has recordings of their talks available on the internet. Considering that state-of-the-art TTS models generate speech which is almost comparable to human quality, this may raise some ethical issues. One with malicious intents could generate fake conversations of public figures, making them say words that were never actually spoken. Even if the technology itself is neither good nor bad, but the moral is in the way people use it, it's important to remember that with great power comes great

responsibility. A good use of this technique would be to make it easier for low-resource languages to build their own TTS datasets, effectively needing only some hours of recordings since the transcriptions can now be generated automatically, without human intervention.

Chapter 4

Text Normalization

So far I've examined the core elements of a TTS pipeline. After the long training procedure described before I had two models that were able to synthesize small sentences with fairly good results. However, as soon as I tried to fed them some news articles, I quickly realized there was still a lot of work to be done. One of the major issues was the inability to pronounce numbers in the Italian language. The model was using the default normalizer of the NeMo library, which is implemented in English. This led to weird pronunciation issues, since the model was trying to read numbers expanded into English words (e.g. 357 -> *"three hundred and fifty seven"*) with an Italian accent. The only solution was to build a custom Italian text normalizer.
In this chapter, I will present the basic theory and a high level explanation of how the text normalizer works.

4.1 Weighted finite-state transducers

The text normalizer that I implemented is based on weighted finite-state transducers (WFSTs). WFSTs are a special case of finite state machines [60]. A finite state machine is a model described by a finite number of states connected by arcs. The machine can be only in one state at a given time. Arcs represents the transitions between the states. Each transition is associated with an input; when the input matches, the machine transitions into a new state. An example is presented in Figure 4.1. The notation is $input : output$ and the letter ϵ stands for *NULL*.
One of the many applications of finite state machines is to use them to

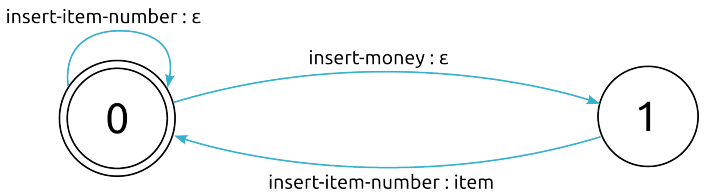


Figure 4.1: Vending machine represented as a finite state machine. Until a user inserts money, the machine is in state 0 and returns nothing.

represent regular expressions. We can build a FSM that is able to accept or reject a given string. Such a machine has no outputs, hence it's usually called finite state acceptor (FSA) [60]. An FSA accepts a given string if there exists a path from a starting state to an end state, and the labels of the arcs traversed by that path correspond to that string [60]. In Figure 4.2 I illustrate a FSA that matches the regex *"^t(ts|o+p)$"*.

A weighted finite-state transducer (WFST) is formally defined [60, p. 14] as a seven-tuple consisting of:

1. a finite set of states Q,
2. a start state $s \in Q$,
3. a semiring $(\mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1})$,
4. a final weight function $\omega \subseteq Q \times \mathbb{K}$,
5. an input alphabet Σ,
6. an output alphabet Φ,
7. a transition relation $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Phi \cup \{\epsilon\}) \times \mathbb{K} \times Q$.

A WFST maps from an input string $x \in \Sigma^*$ to an output string $y \in \Phi^*$ with weight $k \in \mathbb{K}$ as long as complete path with weight k, input x and output y exists.
The superscripted asterisk *, called Kleene star, is the set defined by the infinite union of zero or more concatenation of a string with itself. A complete path begins with a transition from the initial state s to a new state q_1, with input label x_1, output label y_1 and weight k_1 and it ends in a final state.
WFSTs are a generalization of FSTs, and FSTs are a generalization of FSAs. In fact, a FST is a WFST that only has weights of 0 and 1,

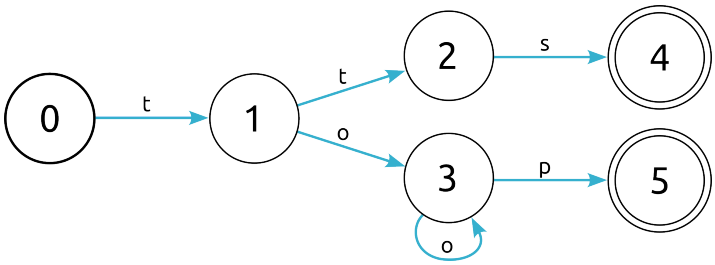


Figure 4.2: FSA for *"^t(ts|o+p)$"*. It accepts $\{tts, top, toop, ...\}$.

while a FSA is a FST that always returns ϵ. An example of a WFST is depicted in Fig. 4.3. The convention is to use a bold circle to represent the starting state, a double circle to represent the final state, and the syntax is *input* : *output*/*weight*.

Figure 4.3: Trivial example of a WFST verbalizing *"good morning"* as *"buongiorno"* with P = 0.8 and as *"buon giorno"* with P = 0.2.

4.2 Implementation

In order to integrate my normalizer in the NeMo framework [61], I had to build two WFSTs:

1. a classifier that assigns a semiotic class to each token;
2. a verbalizer that renders the output of the classifier in a conventional written form.

For example, considering the sentence: *"l'inflazione acquisita è pari al +8,0%"*, the classifier splits it into the following tokens:

```
["l'inflazione", "acquisita", "è", "pari", "al", "+8,0%"].
```

then, it labels each token with a class:

```
tokens { name: "l'inflazione" }
tokens { name: "acquisita" }
tokens { name: "è" }
tokens { name: "pari" }
tokens { name: "al" }
tokens { measure { positive: "true" \
          decimal { integer_part: "otto" \
                    fractional_part: "zero" } \
          units: "percento" } }
```

The verbalizer renders each token in written form:

```
tokens { name: "l'inflazione" } -> l'inflazione
tokens { name: "acquisita" } -> acquisita
tokens { name: "è" } -> è
tokens { name: "pari" } -> pari
tokens { name: "al" } -> al
tokens { measure { positive: "true" \
          decimal { integer_part: "otto" \
                    fractional_part: "zero" } \
          units: "percento" \
        } } -> più otto virgola zero percento
```

finally, the results are combined into the sentence: *"l'inflazione acquisita è pari al più otto virgola zero percento"*.
For the implementation of the WFSTs, the Python library Pynini [62] is used. The class "name" is the default one; in the next sections, I will cover the rest of the classes.

4.2.1 Cardinal

The first class to consider is the "cardinal" class. This one is also the foundation for the following classes; therefore, being able to translate numbers into words is the number one priority. The approach (derived from [63]) is to divide the problem into two tasks: factorization and verbalization. First, the digit string is re-written as a sum of products of powers of 10 (factorization); then, each component of the factorization is verbalized. For example, the digit 357 is factorized as $3 \times 10^2 + 5 \times 10^1 + 7$; then,

the verbalizer maps 3 to "tre", 10^2 to "cento", 5×10^1 to "cinquanta", 7 to "sette".
Implementing this in Pynini is fairly straightforward, but there are a few special cases to handle. The first step is to write the factorizer. With Pynini, it's possible to insert a power of 10 after each input digit. The following example will explain it better than my words:

```
2002100324 -> 2[E9]0[E2]0[E1]2[E6]1[E2]0[E1]0[E3]3[E2]2[E1]4
```

I decided to support numbers up to 10 digits. A string longer than 10 is simply read digit by digit. Shorter digit strings are padded with zeros at the beginning, in order to match the length of 10 digits expected by the factorizer. After the factorization, each power of 10 preceded by a 0 (`0[E*]`) is deleted. However, `0[E3]` is deleted only if it's preceded by `E[6]`, and `0E[6]` is deleted only if preceded by `[E9]`. Then, all the instances of "0" are deleted. So, the string above becomes: `2[E9]2[E6]1[E2][E3]3[E2]2[E1]4`.
To verbalize the string, it's necessary to build a mapping for all the units and the powers of 10:

```
1 -> uno
2 -> due
3 -> tre
...
1[E1] -> dieci
2[E1] -> venti
3[E1] -> trenta
...
1[E1*]1 -> undici
1[E1*]2 -> dodici
1[E1*]3 -> tredici
...
[E2] -> cento
[E3] -> mila
[E6] -> milioni
[E9] -> miliardi
```

The `->` can be read as "is transduced into". All the occurrences of `[E1]` that are preceded by "1" and followed by a digit are converted into `E[1*]`,

since numbers between 11 and 19 have a unique verbalization. Besides that, the language gets redundant, since every number is read as `factor + digit`. There is an exception for the `[E1]` factors followed by 1 and 8, being the only digits that, in Italian, begin with a vowel. They are read without the desinence of the factor; for example, `21` should be written as "ventuno" and not as "ventiuno". Other exceptions to take care of are the translations from plural to singular of "mila", "milioni" and "miliardi" when they are preceded by "uno", but not when "uno" is preceded by "cento", like in "centouno mila". All of these corner cases can be handled in Pynini by adding rewrite rules. It's possible to write a series of rules and then compose them together. The output coming out of each rewrite rule is the input of the next rule.
If the cardinal number is preceded by a sign, the classifier takes care of it by adding "negative: 'true'" or "positive: 'true'", depending on the sign ("-" or "+"). The verbalizer transduces "negative: 'true'" into "meno" and "positive: 'true'" into "più". To summarize, here is an example of the entire process:

1. **input**: -128;
2. **classifier**: cardinal { negative: "true" integer: "centoventotto" };
3. **verbalizer**: meno centoventotto.

4.2.2 Ordinal

The classifier of ordinal numbers is derived from the cardinal classifier. Ordinal numbers are used to represent positions. In Italian, with the expection of the first 10 digits, which have unique names, to turn a cardinal number into an ordinal the desinence of the cardinal is removed, then "esimo" is appended to the end (it can be also "esima", "esimi" or "esime" depending on the context). It's sufficient to append this rule to the cardinal classifier and to correctly map the first ten digits to the corresponding ordinal names. For example:

1. **input**: 1º;
2. **classifier**: ordinal { integer: "primo" };
3. **verbalizer**: primo.

4.2.3 Decimal

Just like the ordinal class, also the decimal class derives from cardinal. When the classifier sees a number containing a comma, it uses the cardinal classifier to classify the number before the comma as "integer_part", and the number after the comma as "fractional_part". The verbalizer inserts " virgola " between the verbalization of the integer and the fractional part. Example:

1. **input**: 12,1;
2. **classifier**: decimal { integer_part: "dodici" fractional_part: "uno" };
3. **verbalizer**: dodici virgola uno.

4.2.4 Electronic

This class handles URLs and email addresses. If the input is an email, it's split into a "username" and a "domain" class. If it's an URL, only the "domain" class is used. If the email or URL contains numbers, those are read digit by digit. The URLs require additional care; for example "http" and "www" are converted to "acca ti ti pi" and "vu vu vu". The verbalizer takes care of expanding punctuation marks into words, as well as inserting "chiocciola" (@) between the username and the domain. Example:

1. **input**: nome@email.it;
2. **classifier**: electronic { username: "nome" domain: "email.it" };
3. **verbalizer**: nome chiocciola email punto it.

4.2.5 Measure

This class makes direct use of the cardinal and decimal classes as subclasses. The mappings between the units of measurement and the expanded words are written in a TSV (tab-separated values) file that is loaded as a FST. Example:

1. **input**: 12,1m;
2. **classifier**: measure { decimal { integer_part: "dodici" fractional_part: "uno" } units: "metri" };
3. **verbalizer**: dodici virgola uno metri.

4.2.6 Money

This class handles currency. If the value is a decimal number, the integer part is associated to the major currency (e.g., dollars) while the fractional part is associated to the minor currency (e.g., cents). The major and minor currencies are written in TSV files loaded at runtime. If the fractional part contains a single digit, it has to be padded with a 0 in order to be correctly transduced. If the integer part is "1" or the fractional part is "01" it's also necessary to use the singular version of the currency. If the integer part is "0", it shouldn't be read. Example:

1. **input**: 12,1€;
2. **classifier**: money { integer_part: "dodici" currency_maj: "euro" fractional_part: "dieci" currency_min: "centesimi" };
3. **verbalizer**: dodici euro e dieci centesimi.

4.2.7 Time

This class could be implemented in a number of different ways, since the way we prefer to read time is subjective. Some prefer a 12 or 24 hour base, or use the convention of counting backwards from the hour. For the sake of clarity, I avoided this convention and I chose a 24 hour base. Common patterns, such as "un quarto" to mean "15" and "mezzogiorno" have been properly handled. The classifier uses the cardinal class to read the numbers before ":" as hours and the numbers after ":" as minutes. The verbalizer takes care of translating the numbers into the common patterns mentioned above, as well as inserting "e" between hours and minutes. Example:

1. **input**: 12:30;
2. **classifier**: time { hours: "dodici" minutes: "trenta" };
3. **verbalizer**: mezzogiorno e mezza.

4.2.8 Whitelist

This class handles all the special cases that don't have a standard category. These are written in a big TSV file, used by the classifier to map an input string into its corresponding expansion. Basically, it works as a

dictionary. Since they have no category, the tokens are assigned to the class "name". In the TSV file I also added some loanwords, which are common words borrowed from the English language, with their corresponding pronunciation (e.g., "computer" -> "compiuter"). Example:

1. **input**: sr.;
2. **classifier**: name: "signor";
3. **verbalizer**: signor.

The whitelist dictionary is still incomplete, since there are probably many corner cases I didn't think about. The only way to improve it is to find new words that need to be normalized through extensive testing.

The final classifier is built by performing a union operation over all the classifiers (WFSTs) described above. A weight is assigned to each classifier, and the shortest path policy is followed. Hence, the more general classes, such as cardinal, have a larger weight than the more specific classes which should have the priority. Similarly, the final verbalizer is obtained from a union of all the verbalizers.

Chapter 5

Evaluation

5.1 MOS

The most common metric to evaluate Text-to-Speech models is the Mean Opinion Score (MOS) [64]. It's a subjective metric that comes from the telecommunications field. A group of subjects has to rate the quality of the synthesized speech on a scale ranging from 1 to 5, where 1 is bad and 5 is excellent. The MOS is just the arithmetic mean of all people's ratings:

$$MOS = \frac{1}{N} \sum_{i=0}^{N-1} R_i \tag{5.1}$$

where R_i is the i-th rating and N is the number of total ratings.

5.2 Method

In order to compare the models that I had previously trained, I created a survey with the following structure:

- 6 ground truth sentences;
- 6 sentences generated from the VITS model;
- 6 sentences from FastPitch + Hifi-GAN;
- 6 sentences from FastPitch + Hifi-GAN fine-tuned on the female speaker.

The sentences were taken from the test dataset and randomized. Each subject had to evaluate the quality of each sentence independently. All

the audios have been normalized in order to make the volume even. I also collected age, gender and audio source (PC speakers, smartphone speakers, headphones, earbuds) in order to see if there were some factors influencing the ratings. I sent the survey to my acquaintances and I received a total of 40 answers.

5.3 Results

Since each sentence has 40 ratings and there are 6 sentences for each model, there are a total of 240 ratings per model. The MOSs are showed in table 5.1.

Model (dataset)	**MOS**
Ground Truth (M-AILABS)	4.38 ± 0.09
VITS (M-AILABS)	3.03 ± 0.14
FastPitch + Hifi-GAN (M-AILABS)	3.98 ± 0.12
FastPitch + Hifi-GAN (woman [58])	3.09 ± 0.13
Ground Truth (LJ Speech) [32]	4.46 ± 0.06
VITS (LJ Speech) [32]	4.43 ± 0.06
FastPitch (LJ Speech) [45]	4.08 ± 0.13
Hifi-GAN (LJ Speech) [55]	4.36 ± 0.07

Table 5.1: MOS computed with 95% confidence interval.

In the upper half of the table there are the models that I trained, while in the lower half there are the MOSs taken from the original papers. First, by comparing the datasets, it appears that M-AILABS has a lower perceived quality with respect to LJ Speech. This could be due to an issue present in some clips of M-AILABS, where the final part of the audio is sometimes cut off too early, resulting in a truncation of the last spoken word. Another reason is the presence of two outliers, as shown in Fig. 5.1, that are pulling down the MOS.
The VITS model received the lowest score with the M-AILABS dataset, while its paper reported the highest score among all the models. By listening to some sentences synthesized by VITS, the voice sounds very natural

but it has a tendency to mispronounce some phonemes, lowering its intelligibility. To be fair, I trained it for 260K steps (3.2.2), while the paper trained it for 800K steps, so it's possible that training it for longer would have fixed that issue; however, the validation loss was already stalling after around 50K steps.
FastPitch and Hifi-GAN achieved the best results on M-AILABS, ending up very close to the score of the FastPitch paper. To be precise, the FastPitch paper calculated its score using WaveGlow for the vocoder [45]. Hifi-GAN, instead, was tested with ground truth spectrograms as inputs [55].
The model fine-tuned on the female speaker scored higher than VITS, but lower than the original. The reason could be due to the voice sounding not as natural as the M-AILABS model. Since there were no mispronunciations, I believe that the issue was in the vocoder. Since the fine-tuning was done on a smaller dataset, adding more data would probably be enough to improve the results and make the voice sound smoother.
Given this results, I decided to use FastPitch and Hifi-GAN as the final models for the podcast generation task. However, as explained in the Deployment chapter (6), I still made sure to support different models as well as different voices, in order to simplify future changes in the architecture.

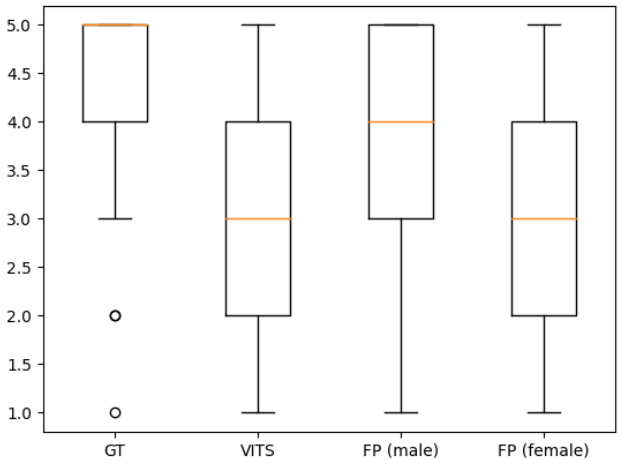
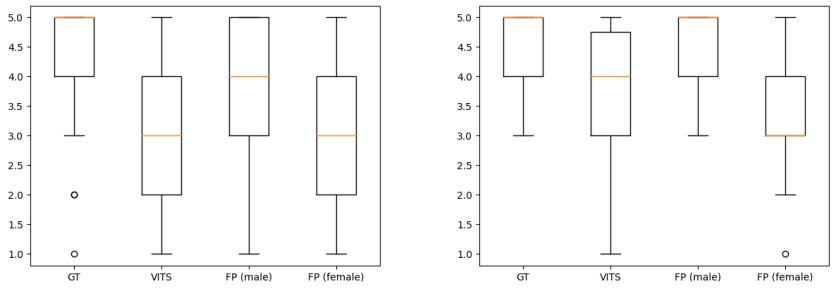


Figure 5.1: Boxplots of the scores of each model.

It's also possible to find out if there is any correlation between factors (age, gender, audio source) and ratings by plotting the distribution of the

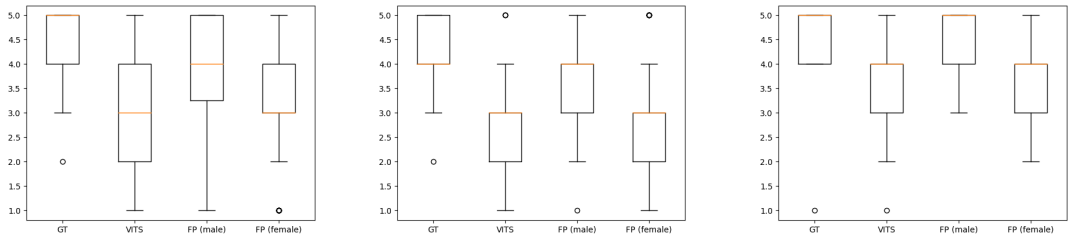
ratings for each factor.



(a) Men. (b) Women.

Figure 5.2: Boxplots for each gender.

By comparing men and women ratings (Fig. 5.2), it appears that women gave higher scores on average, independently of the model. Before drawing any strong conclusion, it's important to specify that only 5 surveys out of 40 were from women, so this might just be a coincidence.



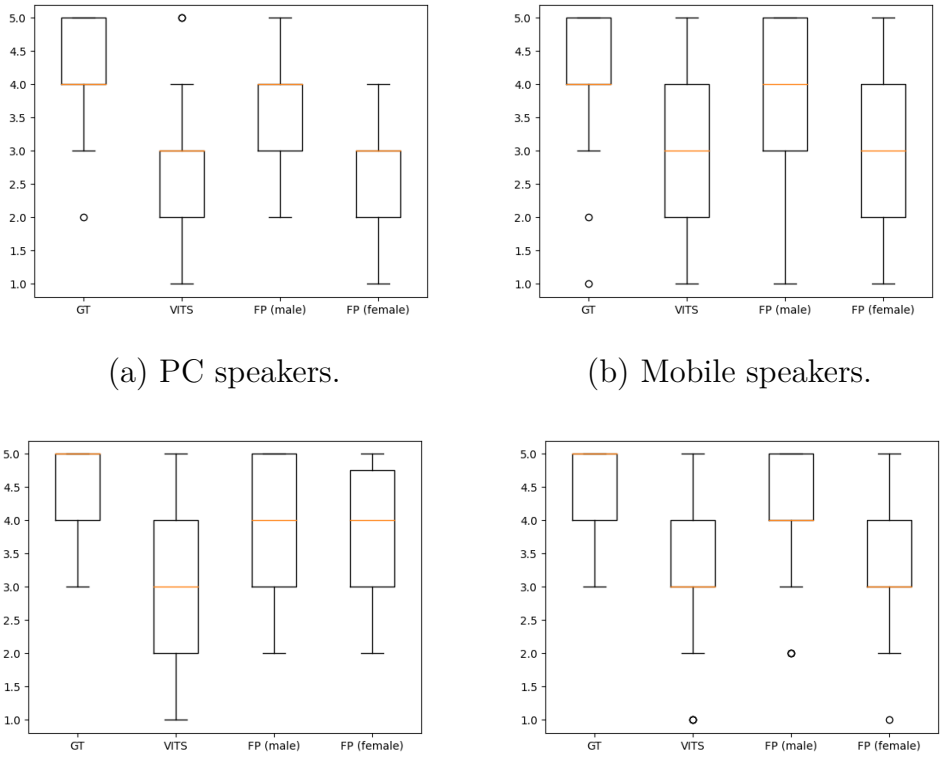
(a) 18-24. (b) 25-44. (c) 45-64.

Figure 5.3: Boxplots for each age group.

In Fig. 5.3 there are the plots for the different age groups. Apparently, the group 25-44 has stricter scores, with a lower spread. The oldest group (45-64) has the highest medians, suggesting a difference in the way speech is perceived compared to younger people.

Finally, looking at the ratings from different audio sources (Fig. 5.4), it seems that people who listened to the audios using headphones or earbuds gave higher scores compared to people that used PC or smartphone speakers. This could just mean that using headphones or earbuds improved the

listening experience, increasing the perceived quality of the synthesized speech.



(a) PC speakers.

(b) Mobile speakers.

(c) Headphones.

(d) Earbuds.

Figure 5.4: Boxplots for each audio source.

Chapter 6

Deployment

Once the models had been trained and evaluated, the last task was to wrap them inside a microservice. As explained in the introduction (1), the end goal is to create podcasts. At this moment, though, the models are only able to synthesize short sentences of around 10 seconds; hence, an additional infrastructure is needed.
At a high level, the microservice should manage the following workflow:

1. a user selects a list of articles he wants to listen to;
2. the unique identifiers of the articles are sent to the microservice;
3. given the article ids, the service grabs the full text of each article from the database;
4. the full text is split into sentences that are sent to the ML models to perform inference;
5. the resulting audios are concatenated to obtain the full spoken version of the article;
6. the audios of the articles are joined together to form the final podcast, with a jingle played in between each article;
7. the podcast is sent to the user.

Obviously, it would be undesirable to hang the user while performing inference, therefore the service will return a podcast identifier that the client can poll to check the request status.

6.1 Technologies

The service is composed of a total of 4 Docker containers, managed by Docker Compose. A Docker [65] container is a process isolated from the rest of the system. A container is an instance of an image, which is a file that packages everything needed to run the application, including the environment and the dependencies. The main advantage is portability, since a docker container is going to run in the same way on every machine, provided that the host kernel is compatible. An image is built from a file containing a set of instructions, called Dockerfile. Docker Compose is an orchestration tool that allows to easily run multi-container applications and it creates a network that enables containers to communicate with each other. It uses a YAML configuration file to setup and configure the containers.
The first container runs a FastAPI application. FastAPI [66] is a web framework for building Python APIs. The FastAPI container interacts with two other containers. One container runs mongod, which is the MongoDB server. The other container runs RabbitMQ. RabbitMQ [67] is a message broker based on the Advanced Message Queuing Protocol (AMQP), responsible of managing the communication between FastAPI and the service that performs inference. Its role will be explained more clearly in the next section. The last container runs the Python application that performs inference using the trained models.
In the Docker Compose configuration file I added options to tune the number of inference processes and to allow the selection of different TTS models. Since we currently perform inference on the CPU and not on the GPU, I relied on a multi-processing solution. Each process is a consumer waiting for RabbitMQ messages. A high level schema of the full architecture and the interactions between the containers is illustrated in Fig. 6.1.

6.2 Functionality

The FastAPI service exposes 3 endpoints:

1. The first endpoint allows the client to submit the list of articles that will constitute the podcast and the voice of the speaker he prefers.

Then, a new document is added to the Podcast collection, containing the podcast identifier, the request status (NotStarted, Running, Succeeded, Failed), the list of article identifiers submitted and some other minor information. Afterwards, the FastAPI service (producer) forwards the podcast id to the task queue managed by RabbitMQ and returns the podcast identifier to the client. RabbitMQ pushes the message to the first available consumer. The inference process that receives the message updates the podcast status to "Running", queries MongoDB to retrieve the list of article identifiers given the podcast id and it retrieves the body of the corresponding articles. Each article is sent to the synthesizer, which is a wrapper that splits the full text into sentences, runs inference on each sentence using the TTS model specified in the Docker Compose configuration file, and returns the audio resulting from the concatenation of all the synthesized sentences. Finally, the podcast is composed by concatenating the audio of each article with a jingle and it's saved on the disk as a mp3 file. In the end, the podcast file path is updated and the status is set to "Succeeded". If everything goes as described, the consumer sends an acknowledge to RabbitMQ to signal that the message can be deleted. Otherwise, if the inference process fails before sending the acknowledge, RabbitMQ re-queues the message. Since the inference process is expensive and the number of articles is finite, a small optimization that was implemented is the caching of the audios of the articles. In this way, if more users request the same article, it won't be generated from scratch every time, but only for the first user that requests it.

2. The second endpoint is used by the client to retrieve information about his podcast request, such as the inference status, by using the podcast identifier retrieved from the response of the first endpoint. This endpoint can be polled until the status is "Succeeded", which indicates that the podcast file is ready to be downloaded.
3. The third endpoint allows the client to download a podcast, given the podcast id.

As a last note, there was an interesting issue with the synthesis of the sentences. The model had the tendency to sometimes skip the last word of a sentence. This was resolved by adding some extra padding text at the end of each sentence and then cutting the padding part from the synthesized audio. By listening to some ground truth samples from the

training dataset, I noticed that in some of them the audio was cut off too early, while the speaker was still pronouncing the last word. My hypothesis is that the model learned to mimic a similar behaviour during training. This could be an example of how defects on the training data can be picked up by a neural network, sometimes in unpredictable ways.

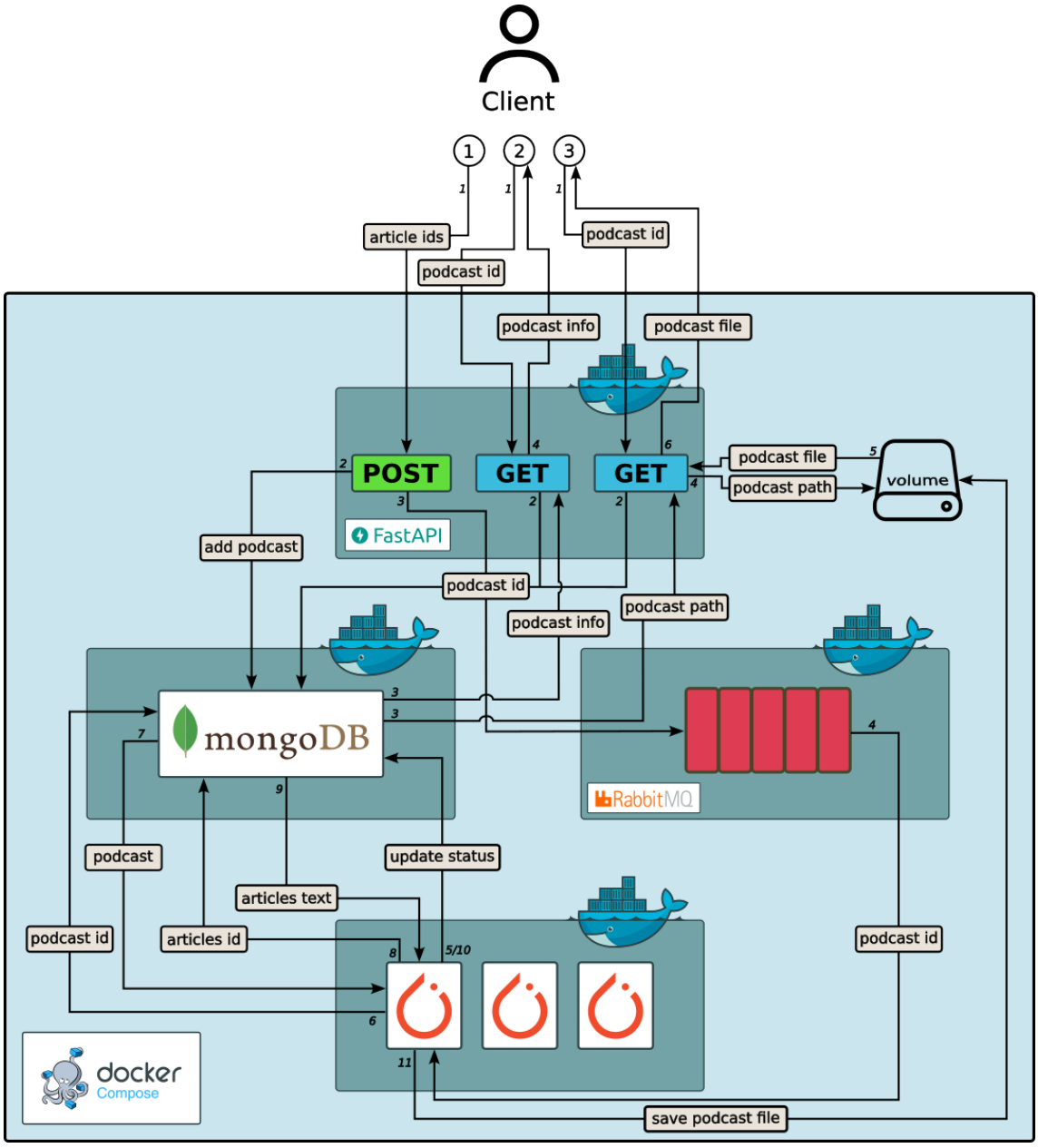


Figure 6.1: The TTS microservice.

Chapter 7

Conclusions and future work

In this work I presented a novel application of Text-to-Speech systems to the task of automated generation of podcasts of news articles. I discussed the current state of the TTS field and the criteria that I followed to select which models were worth training. Then, I described in detail the training process, as well as my attempt to fine-tune the trained models on a new speaker's dataset. This allowed me to highlight one of the biggest problems for open source TTS, which is the lack of public domain datasets. In fact, there are many languages which don't have any public datasets and are forced to use commercial TTS services, built with proprietary datasets. I presented a way to mitigate this problem, thanks to the recent progresses in the automatic speech recognition field, which enabled me to automate the creation of the transcriptions of the speech recordings; with this technique I built the dataset used for fine-tuning. Afterwards, I explained in detail how I implemented an Italian text normalizer that could handle most of the corner cases presented in news articles, such as numbers, units of measure, currencies and loanwords. Then, I discussed the evaluation method, based on the MOS, and I compared the results of my trained models against the state-of-the-art. Despite the limitations, both in terms of dataset quality and computational resources, the models achieved decent results on short sentences, with one model not too far from the state-of-the-art. In the end, I presented the architecture of the microservice that was built around the TTS models, discussing the complete pipeline for synthesizing news articles and combining them into a

podcast.
I believe that there are several areas of improvement. The first issue is that, even if I chose parallel TTS models, the training and inference cost is still significant. This prevented me from experimenting with custom architectures or comparing the performances of more models, since that would have required an extensive amount of time. There is a need for more lightweight neural TTS models.
Another critical aspect concerns the robustness and generalization capabilities. I trained the models on a public domain dataset which made extensive use of words that today would be considered archaic. The domain of the training dataset is very different from the domain of the articles, hence there are some combination of phonemes and words which result in a poor synthesis. This is especially true for loanwords. The approach of handling them in the text normalizer, hard coding their pronunciation in a dictionary, works most of the times. However, it's not perfect, since the pronunciation of some English words is impossible to get right. In the future, it would be interesting to experiment with a multilingual model, trained on a dataset composed of different languages. This might also improve the generalization capabilities of the model.
To conclude, I hope that I've shed some light on the current state of TTS research, exposing some of the problems that are still unsolved. I think that, in the near future, it will be possible to generate podcasts which will be indistinguishable from human speech, or maybe even better, seeing how quickly the machine learning world is progressing. My wish is that some of the aspects discussed in this work will be helpful for future research.

Bibliography

[1] "What is sound?" https://www.soundproofingcompany.com/soundproofing_101/what-is-sound, The Soundproofing Company, 2019.

[2] "Sampling rate and aliasing effect," https://www.kistler.com/en/glossary/term/sampling-rate-and-aliasing-effect/, Kistler Group, 2022.

[3] R. G. Lyons, *Understanding digital signal processing, 3/E*. Pearson Education India, 1997.

[4] "Short-time fft," https://it.mathworks.com/help/dsp/ref/dsp.stft.html, The MathWorks, Inc., 2022.

[5] D. Gartzman, "Getting to know the mel spectrogram," https://towardsdatascience.com/getting-to-know-the-mel-spectrogram-31bca3e2d9d0, 2019.

[6] X. Tan, T. Qin, F. Soong, and T.-Y. Liu, "A survey on neural speech synthesis," *arXiv preprint arXiv:2106.15561*, 2021.

[7] H. Dudley and T. H. Tarnoczy, "The speaking machine of wolfgang von kempelen," *The Journal of the Acoustical Society of America*, vol. 22, no. 2, pp. 151–166, 1950.

[8] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv preprint arXiv:1609.03499*, 2016.

[9] X. Tan, J. Chen, H. Liu, J. Cong, C. Zhang, Y. Liu, X. Wang, Y. Leng, Y. Yi, L. He *et al.*, "Naturalspeech: End-to-end text to speech synthesis with human-level quality," *arXiv preprint arXiv:2205.04421*, 2022.

[10] "Infinite dial 2022," http://www.edisonresearch.com/wp-content/

uploads/2022/03/Infinite-Dial-2022-Webinar-revised.pdf, Edison Research, 2022.

[11] Podcast., *Cambridge Academic Content Dictionary*. Cambridge University Press, 2009.

[12] J. G. Proakis, *Digital signal processing: principles algorithms and applications*. Pearson Education India, 2001.

[13] U. Zölzer, *Digital audio signal processing*. John Wiley & Sons, 2022.

[14] S. Rosen, "For auditory signals and human listeners the accepted range is 20hz to 20khz the limits of human hearing," in *Signals and Systems for Speech and Hearing*. BRILL, 2011, p. 163.

[15] K. Ito and L. Johnson, "The lj speech dataset," https://keithito.com/LJ-Speech-Dataset/, 2017.

[16] J. O. Smith, *Spectral Audio Signal Processing*. `http:`http://ccrma.stanford.edu/ jos/sasp//-`ccrma.stanford.edu/~jos/sasp/`, accessed <date>, online book, 2011 edition.

[17] D. A. Lyon, "The discrete fourier transform, part 4: spectral leakage," *Journal of object technology*, vol. 8, no. 7, 2009.

[18] S. S. Stevens, J. Volkmann, and E. B. Newman, "A scale for the measurement of the psychological magnitude pitch," *The journal of the acoustical society of america*, vol. 8, no. 3, pp. 185–190, 1937.

[19] F. F. Li and T. J. Cox, *Digital signal processing in audio and acoustical engineering*. CRC Press, 2019.

[20] P. Taylor, *Text-to-speech synthesis*. Cambridge university press, 2009.

[21] L. Greenemeier, "Getting back the gift of gab: next-gen handheld computers allow the mute to converse," *Scientific American*, vol. 10, 2009.

[22] D. H. Klatt, "Software for a cascade/parallel formant synthesizer," *the Journal of the Acoustical Society of America*, vol. 67, no. 3, pp. 971–995, 1980.

[23] X. Huang, A. Acero, and H.-W. Hon, "Spoken language processing: guide to algorithms and system development," 2001.

[24] M. Mohri, "Weighted finite-state transducer algorithms. an overview," *Formal Languages and Applications*, pp. 551–563, 2004.

[25] Y. Wang, R. Skerry-Ryan, D. Stanton, Y. Wu, R. J. Weiss, N. Jaitly, Z. Yang, Y. Xiao, Z. Chen, S. Bengio *et al.*, "Tacotron: Towards end-to-end speech synthesis," *arXiv preprint arXiv:1703.10135*, 2017.

[26] J. Shen, R. Pang, R. J. Weiss, M. Schuster, N. Jaitly, Z. Yang, Z. Chen, Y. Zhang, Y. Wang, R. Skerrv-Ryan *et al.*, "Natural tts synthesis by conditioning wavenet on mel spectrogram predictions," in *2018 IEEE international conference on acoustics, speech and signal processing (ICASSP)*. IEEE, 2018, pp. 4779–4783.

[27] Y. Ren, Y. Ruan, X. Tan, T. Qin, S. Zhao, Z. Zhao, and T.-Y. Liu, "Fastspeech: Fast, robust and controllable text to speech," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[28] A. Van Den Oord, N. Kalchbrenner, and K. Kavukcuoglu, "Pixel recurrent neural networks," in *International conference on machine learning*. PMLR, 2016, pp. 1747–1756.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[30] Y. Ren, C. Hu, X. Tan, T. Qin, S. Zhao, Z. Zhao, and T.-Y. Liu, "Fastspeech 2: Fast and high-quality end-to-end text to speech," *arXiv preprint arXiv:2006.04558*, 2020.

[31] J. Donahue, S. Dieleman, M. Bińkowski, E. Elsen, and K. Simonyan, "End-to-end adversarial text-to-speech," *arXiv preprint arXiv:2006.03575*, 2020.

[32] J. Kim, J. Kong, and J. Son, "Conditional variational autoencoder with adversarial learning for end-to-end text-to-speech," in *International Conference on Machine Learning*. PMLR, 2021, pp. 5530–5540.

[33] H. McGuire, "Librivox, free public domain audiobooks," https://librivox.org/, 2005.

[34] "Copyright," https://europa.eu/youreurope/business/running-business/intellectual-property/copyright/index_en.htm, Your Europe, 2022.

[35] R. Ardila, M. Branson, K. Davis, M. Henretty, M. Kohler, J. Meyer, R. Morais, L. Saunders, F. M. Tyers, and G. Weber, "Common voice: A massively-multilingual speech corpus," *arXiv preprint arXiv:1912.06670*, 2019.

[36] "The m-ailabs speech dataset," https://www.caito.de/2019/01/03/the-m-ailabs-speech-dataset/, Munich Artificial Intelligence Laboratories GmbH, 2019.

[37] D. H. Johnson, "Signal-to-noise ratio," *Scholarpedia*, vol. 1, no. 12, p.

2088, 2006.

[38] C. Kim and R. M. Stern, "Robust signal-to-noise ratio estimation based on waveform amplitude distribution analysis," http://www.cs.cmu.edu/~robust/archive/algorithms/WADA_SNR_IS_2008/, 2008.

[39] ——, "Robust signal-to-noise ratio estimation based on waveform amplitude distribution analysis," in *Ninth Annual Conference of the International Speech Communication Association*, 2008.

[40] "Tts: Text-to-speech for all." https://github.com/mozilla/TTS, Mozilla, 2022.

[41] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *Communications of the ACM*, vol. 63, no. 11, pp. 139–144, 2020.

[42] "Vits: Conditional variational autoencoder with adversarial learning for end-to-end text-to-speech." https://github.com/jaywalnut310/vits, 2021.

[43] M. Bernard and H. Titeux, "Phonemizer: Text to phones transcription for multiple languages in python," *Journal of Open Source Software*, vol. 6, no. 68, p. 3958, 2021. [Online]. Available: https://doi.org/10.21105/joss.03958

[44] I. P. Association, *Handbook of the International Phonetic Association: A guide to the use of the International Phonetic Alphabet*. Cambridge University Press, 1999.

[45] A. Łańcucki, "Fastpitch: Parallel text-to-speech with pitch prediction," in *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2021, pp. 6588–6592.

[46] O. Kuchaiev, J. Li, H. Nguyen, O. Hrinchuk, R. Leary, B. Ginsburg, S. Kriman, S. Beliaev, V. Lavrukhin, J. Cook *et al.*, "Nemo: a toolkit for building ai applications using neural modules," *arXiv preprint arXiv:1909.09577*, 2019.

[47] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[48] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[49] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[50] P. Boersma *et al.*, "Accurate short-term analysis of the fundamental frequency and the harmonics-to-noise ratio of a sampled sound," in *Proceedings of the institute of phonetic sciences*, vol. 17, no. 1193. Amsterdam, 1993, pp. 97–110.

[51] B. McFee, A. Metsai, M. McVicar, S. Balke, C. Thomé, C. Raffel, F. Zalkow, A. Malek, Dana, K. Lee, O. Nieto, D. Ellis, J. Mason, E. Battenberg, S. Seyfarth, R. Yamamoto, viktorandreevichmorozov, K. Choi, J. Moore, R. Bittner, S. Hidaka, Z. Wei, nullmightybofo, A. Weiss, D. Hereñú, F.-R. Stöter, L. Nickel, P. Friesch, M. Vollrath, and T. Kim, "librosa/librosa: 0.9.2," Jun. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6759664

[52] "Hpc@polito," https://www.hpc.polito.it/, The DAUIN HPC Initiative (C), 2022.

[53] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Workshop on job scheduling strategies for parallel processing*. Springer, 2003, pp. 44–60.

[54] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PloS one*, vol. 12, no. 5, p. e0177459, 2017.

[55] J. Kong, J. Kim, and J. Bae, "Hifi-gan: Generative adversarial networks for efficient and high fidelity speech synthesis," *Advances in Neural Information Processing Systems*, vol. 33, pp. 17 022–17 033, 2020.

[56] A. Pettarin, "forced-alignment-tools," https://github.com/pettarin/forced-alignment-tools, 2018.

[57] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, "Robust speech recognition via large-scale weak supervision," Tech. Rep., OpenAI, Tech. Rep., 2022.

[58] M. C. Silvia Cecchini, "Le avventure di nicola nickleby [audiolibro]," https://www.liberliber.it/online/autori/autori-d/charles-

dickens/le-avventure-di-nicola-nickleby-audiolibro/, 2008.
[59] J. Robert, "Pydub," https://github.com/jiaaro/pydub/, 2011.
[60] K. Gorman and R. Sproat, "Finite-state text processing," *Synthesis Lectures onSynthesis Lectures on Human Language Technologies*, vol. 14, no. 2, pp. 1–158, 2021.
[61] "Wfst tutorial." https://colab.research.google.com/github/NVIDIA/NeMo/blob/stable/tutorials/text_processing/WFST_Tutorial.ipynb, NVIDIA, 2022.
[62] K. Gorman, "Pynini: A python library for weighted finite-state grammar compilation," in *Proceedings of the SIGFSM Workshop on Statistical NLP and Weighted Automata*, 2016, pp. 75–80.
[63] R. Sproat, "Multilingual text analysis for text-to-speech synthesis," *Natural Language Engineering*, vol. 2, no. 4, pp. 369–380, 1996.
[64] Wikipedia contributors, "Mean opinion score — Wikipedia, the free encyclopedia," https://en.wikipedia.org/wiki/Mean_opinion_score, 2022.
[65] "Overview | docker documentation," https://docs.docker.com/get-started/, Docker Inc., 2022.
[66] S. Ramírez, "Fastapi," https://fastapi.tiangolo.com/, 2022.
[67] G. M. Roy, "Rabbitmq," https://www.rabbitmq.com/, 2022.