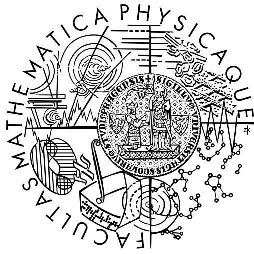

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Jan Vainer

Efficient neural speech synthesis

Institute of Formal and Applied Linguistics

Supervisor of the master thesis: Mgr. et Mgr. Ondřej Dušek, Ph.D.
Study programme: Theoretical Computer Science
Study branch: Artificial Intelligence

Prague 2020

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date .... signature of the author

I am grateful especially to Mgr. et Mgr. Ondřej Dušek, Ph.D. for his patient supervision. His advice, ideas and support helped me greatly to finish this thesis. I am also grateful to my significant other Alena Jarolímová for her kind support. But most importantly, I want to thank my family for their lasting support throughout my studies.

Title: Efficient neural speech synthesis

Author: Jan Vainer

Department: Institute of Formal and Applied Linguistics

Supervisor: Mgr. et Mgr. Ondřej Dušek, Ph.D., Institute of Formal and Applied Linguistics

Abstract: While recent neural sequence-to-sequence models have greatly improved the quality of speech synthesis, there has not been a system capable of fast training, fast inference and high-quality audio synthesis at the same time. In this thesis, we present a neural speech synthesis system capable of high-quality faster-than-real-time spectrogram synthesis, with low requirements on computational resources and fast training time. Our system consists of a teacher and a student network. The teacher model is used to extract alignment between the text to synthesize and the corresponding spectrogram. The student uses the alignments from the teacher model to synthesize mel-scale spectrograms from a phonemic representation of the input text efficiently. Both systems utilize simple convolutional layers. We train both systems on the english LJSpeech dataset. The quality of samples synthesized by our model was rated significantly higher than baseline models. Our model can be efficiently trained on a single GPU and can run in real time even on a CPU.

Keywords: Speech synthesis, efficiency, real-time synthesis, convolutional neural networks

Contents

List of Abbreviations **3**

1 Introduction **4**
1.1 Background . . . 4
1.2 Goal of this thesis . . . 5
1.3 Thesis structure . . . 5

2 Theoretical background **6**
2.1 Text-to-speech synthesis . . . 6
2.2 Human voice . . . 6
2.3 Signal processing . . . 8
2.4 Probabilistic modelling . . . 11
2.4.1 Maximum likelihood estimation . . . 11
2.4.2 Stochastic gradient descent . . . 12
2.4.3 Chain rule factorization . . . 12
2.4.4 Loss functions . . . 12
2.5 Neural networks . . . 13
2.5.1 Convolutional and recurrent networks . . . 14
2.5.2 Attention . . . 17
2.5.3 Positional encoding . . . 19
2.5.4 Residual networks . . . 20
2.5.5 Batch normalization . . . 21

3 Literature review **22**
3.1 Non-neural approaches . . . 22
3.2 Neural approaches . . . 23
3.2.1 Synthesis scope . . . 24
3.2.2 Alignment mechanism . . . 24
3.2.3 Training and inference: Performance issues . . . 26
3.3 Models related to our work . . . 28

4 Our TTS architecture **31**
4.1 Design principles . . . 31
4.2 Architecture overview . . . 31
4.3 Duration extraction – Teacher network . . . 32
4.3.1 Network structure . . . 32
4.3.2 Attention preconditioning . . . 34
4.3.3 Training . . . 36
4.3.4 Duration Extraction . . . 37
4.4 Spectrogram synthesis – Student network . . . 39
4.4.1 Network structure . . . 40
4.4.2 Positional encoding for student model . . . 41
4.4.3 Training . . . 41
4.5 Vocoding . . . 43

5 Experiments 44
5.1 Data set and data pre-processing . . . 44
5.2 Duration extraction . . . 44
5.2.1 Training stability . . . 44
5.2.2 Alignment learning . . . 45
5.2.3 Error propagation in sequential inference . . . 46
5.2.4 Inference speed . . . 47
5.2.5 Location masking in duration extraction . . . 47
5.2.6 Model validation . . . 48
5.3 Spectrogram synthesis . . . 49
5.3.1 Input phoneme durations accuracy . . . 49
5.3.2 Batch normalization and dropout . . . 49
5.3.3 Learning phoneme durations versus spectrograms . . . 50
5.3.4 Using logarithmic phoneme durations . . . 50
5.3.5 Learning rates . . . 51
5.3.6 Normalizing spectrograms . . . 53
5.3.7 SSIM . . . 53
5.3.8 Padding and masking . . . 53
5.3.9 Homogeneus spectrogram segments . . . 53
5.3.10 Positional encodings . . . 53
5.4 Persisting problems and challenges . . . 54

6 Evaluation 55
6.1 Voice quality . . . 55
6.1.1 Survey interface . . . 55
6.1.2 Compared setups . . . 56
6.1.3 Results . . . 57
6.2 Inference speed . . . 58
6.3 Training time . . . 60
6.4 Directions for future work . . . 61

7 Conclusion 64

Conclusion 64

Bibliography 65

A Appendix 72

List of Figures 75

List of Tables 78

List of Abbreviations

The next list describes several symbols that are used within the body of the document.

$\mathbf{X}$	A sequence of random column vectors $\{\vec{X}_i\}_{i=1}^n$. Can be seen as a random matrix $\mathbf{X} \in \mathbb{R}^{k \times n}$
$\mathbf{x}$	A sequence of column vectors $\{\vec{x}_i\}_{i=1}^n$. Can be seen as a matrix $\mathbf{x} \in \mathbb{R}^{k \times n}$
$\mathbf{x}[i, j]$	The j-th element of the i-th vector. $\mathbf{x}[i, j] = \vec{x}_i[j]$
$\vec{x}$	A column vector of real numbers $\vec{x} = (x_1, \ldots x_k)^T$
$\vec{x}[i]$	An i-th element of a column vector, i.e. a scalar $x = \vec{x}[i] \in \mathbb{R}$
$\vec{x}_i$	An i-th column vector from $\mathbf{x}$, $\vec{x}_i \in \mathbb{R}^k$
x	A scalar
DTW	Dynamic time warping
GPU	Graphics processing unit
iSTFT	inverse Short-time fourier transform
MLE	Maximum likelihood estimation
SGD	Stochastic gradient descent
STFT	Short-time fourier transform
TTS	Text-to-speech or Speech synthesis

1. Introduction

1.1 Background

Voice assistants such as Alexa, Google assistant or Siri are becoming more popular every year and have become a tool that many people use on a daily basis. Text-to-speech synthesis (TTS) is an integral part of such systems. It provides a natural interface between the user and the machine. There is a growing number of uses in other areas of human lives too. TTS can be used to automatically generate spoken messages for the public transport announcers. Blind people can utilise TTS for screen reading including reading of emails or other personal messages and for internet browsing. Synthetic speech also enables vocally handicapped people to verbally communicate with others. A famous example of such use would be the Stephen Hawking's speech synthesis. High quality synthesis systems could be also used for automatic e-book reading. Recently, the research areas of voice cloning, voice editing and speech enhancement have been gaining traction. For example, it is possible to overdub parts of already recorded speech while keeping the characteristics and prosody of the speaker. This allows easier editing of already recorded podcasts, audio books but also voiced movie scenes. Sampling and generation of speech for random voices is also possible (Jia et al., 2018). This would for example allow creating new voice lines for PC games without the necessity to record all the character scripts, which could be useful especially in open world games with thousands of lines of dialogues.

Speech synthesis can be used as a helper module in other systems. For example, Aeneas[1] uses TTS to extract alignment between an audio recording and the corresponding transcript. The Resulting alignments can be used for automatic subtitle synchronisation as well as for creation of new audio datasets for automatic speech recognition systems.

Deployment of speech synthesis systems in real world applications brings many challenges. Speech quality is one of the main factors determining the overall usefulness of the system. The synthesized voice should be intelligible, noiseless and should sound human-like. The pronunciation and prosody of the voice also play a big role in perceived voice quality. The ability to change the pitch or the color of the voice can also be useful.

Since the system may be deployed on relatively low-resource devices such as smartphones, it may be necessary to consider CPU/GPU requirements of the TTS system. Well-sounding, but computationally heavy systems may be of limited use. Synthesis speed can also play an important role in certain scenarios. For example, dialogue systems typically require real-time speech synthesis since delayed machine responses degrade the naturalness of the dialogue.

Recent TTS systems (Shen et al., 2018; Oord et al., 2016)(see Section 3) are often based on neural approaches and require training of a neural network model on a dataset of audio recordings and corresponding transcripts. The training can take anywhere between tens of hours to tens of days depending on the model and on the available hardware. Deploying such system requires the developer to either use a pretrained model, or train a custom model. The former option is typically

[1] `https://github.com/readbeyond/aeneas`

not available for other languages than English as publicly available pretrained models are mostly trained on English datasets. The latter option may require access to significantly large computational resources that are not accessible to everyone.

1.2 Goal of this thesis

The goal of this thesis is to provide a single-speaker speech synthesis system capable of fast training times and speedy inference, while keeping high quality of synthesized speech and requiring low computational resources. Speech Synthesis systems based on neural networks have reached a high quality of synthesized audio. Thus, we also employ neural networks in our TTS system. The speed of training of a neural model is influenced by two factors–size of the model (number of parameters) and parallelizability of the computations inside the model. The number of parameters of the model has an effect on the capacity of the model. Too few parameters can cause lower quality of the synthesized speech, while too many parameters prolong training and slow down inference. Being able to perform as many operations inside the model as possible in parallel enables efficient use of hardware designed for parallel computing such as *graphics processing units* (GPUs). We utilize Convolutional neural networks and attention mechanisms and cast the synthesis problem as a probabilistic modelling framework. Our system consists of two parts. The first part is responsible for extracting character-level alignments between audio and text and is used only during training. The second part uses the extracted alignments for speech synthesis and is used during training and inference.

1.3 Thesis structure

The rest of this thesis is organised as follows. First, we describe theory and techniques relevant to our system in Section 2. Secondly, we review the literature connected to our work and make a comprehensive comparison of state-of-the-art techniques applied to speech synthesis in Section 3. Thirdly, we provide an in-depth description of our methodology including the model architecture, training procedure, and other tricks that helped to improve the model performance in Section 4. Finally, we evaluate our results and suggest further research directions in Section 5. The thesis ends with a short summary and concluding remarks in Section 7. We also provide our source code and synthesized audio samples in our repository. All source code used for experiments in this thesis as well as synthesised audio samples are included in the archive file attached to this thesis. We also publish our source code[2] and audio samples on github.

[2] `https://github.com/janvainer/speedyspeech`

2. Theoretical background

In this chapter, we first define a test-to-speech system. Then, we describe the basic characteristics of human voice in Section 2.2. Next, we discuss signal processing techniques used in this thesis in Section 2.3. Finally, basic concepts from probabilistic modelling are introduced and necessary concepts from deep learning and neural networks are reviewed in Sections 2.4 and 2.5.

2.1 Text-to-speech synthesis

Text-to-speech synthesis is a process of converting written language into audio waves. A speech synthesis system usually consists of several subsystems with different responsibilities:

1. Typically, it is necessary to parse the input text into smaller units such as sentences.

2. Words containing numeric and other symbols are rewritten with alphabetic characters. Abbreviations are expanded into full expressions.

3. Next, a linguistic analysis of the text is conducted in order to extract useful features such as phonematic representations of the words, phoneme durations, or intonation (see Sections 2.2, 3.1).

4. The linguistic features can be used to generate spectral representation of audio such as a spectrogram (see Section 2.3) that is further decoded into raw waveforms or to generate the waveforms directly.

Steps 1–3 are considered as preprocessing steps; In this thesis, we concentrate on the spectrogram generation (Step 4). There are various aproaches to generation of waveforms from phonemic representations. We describe some of the more traditional methods in Section 3.1. Modern probabilistic approaches are described in Section 2.5.

2.2 Human voice

We include a very brief summary of voice production in humans; please refer to (Taylor, 2009, p. 147–170) for more details.

Human voice is a sound produced by human vocal organs. The sounds are mostly created by a flow of air from the lungs flowing through the vocal organs. Different vocal organs are responsible for different kinds of sounds. For example, vocal folds are responsible for the melody of the voice. The air passing through the vocal folds creates vibrations that result in a periodic sound similar to musical tone. The frequency of the sound is called the *fundamental frequency* (F0) also termed as *pitch*. Together with F0, the periodic signal resonates at multiples of the fundamental frequency called *harmonic frequencies*. The harmonic frequencies give a characteristic color to the voice of the speaker. Similarly, different musical instruments such as the piano or the guitar can play the same note, but

the sound of both instruments at the given note has a different color (also called timbre). The range of frequencies that can be generated with vocal folds differs from speaker to speaker and is determined by the length of the vocal folds.

Speech sounds can be divided into vowels and consonants. When vocal folds vibrate, they generate a *voiced* sound. Vowels are voiced and are mostly responsible for speech prosody. If the vocal folds are released, the voiced sound disappears and non-periodic noise is produced. For example, try saying and whispering “ah” with your fingers placed on your neck. In the first case, you should feel vibrations while in the second case, the vocal folds should not vibrate. The noise can be modulated by lips, tongue and teeth to create unvoiced consonants such as “s” or “f”. A combination of voiced and noisy sounds is also possible. For example, when “z” is pronounced, the vocal folds vibrate, but the sound is further modulated by teeth. You can try to transfer from saying “s” to saying “z”. The vocal folds will start to vibrate at some point.

The vocal organs could be divided into a sound source and sound filters. For example, the vocal folds are a source of the sound, while articulators such as oral and nasal cavities inside the vocal tract can be seen as filters or modulators of the sound. The articulators typically do not affect the pitch of the incoming sound, but boost some of the harmonic frequencies while suppressing the other. For example, the source of the sound of all vowels are the vocal folds, but the oral cavity is responsible for what vowel is spoken. It is possible to keep the same pitch and vary the spoken vowels at the same time by changing the position of the jaw, lips and tongue. Similarly, varying consonants can be produced by varying the positioning of the tongue and teeth and by adding either a voiced sound or noise. Importantly, consonants and vowels are determined by the configuration of the vocal tract.

There is a correspondence between the configuration of the vocal tract and presence and absence of various harmonic frequencies in the sound. As already indicated, some frequencies in the source signal are suppressed and some are amplified. The amplified harmonics are said to *resonate* and are called *formants*. A spectrum of the vowels “a” and “e” is visualised in Figure 2.1. The spectrum of a sound has a local and a global structure. The global structure can be captured with a *spectral envelope*. The spectral envelope contains peaks and valleys, where the peaks correspond to high concentration of acoustic energy around the given frequency. The peak frequency regions are the formants. Usually, the first two or three formants are sufficient to identify a vowel. Formant analysis is useful in formant speech synthesis. Please see Section 3.1 for more information.

Often, there is not a one-to-one correspondence between the orthographic representation[1] of the spoken language and the sounds of the language. For example, the word “experience” is rather pronounced like “ikspirients”. This can be problematic, because in TTS synthesis, we would like to be able to uniquely map written characters to corresponding speech sounds. Fortunately, phonetics provides a specialized symbolic representation of the speech sounds. Any distinguishable sound which is a part of speech is called a *phone*. Any speech utterance consists of a sequence of phones. Multiple phones may correspond to the same meaning. For example, two people can speak English with different accents and pronounce words in a slightly different way, but both can still understand each

[1]How the spoken language is written.

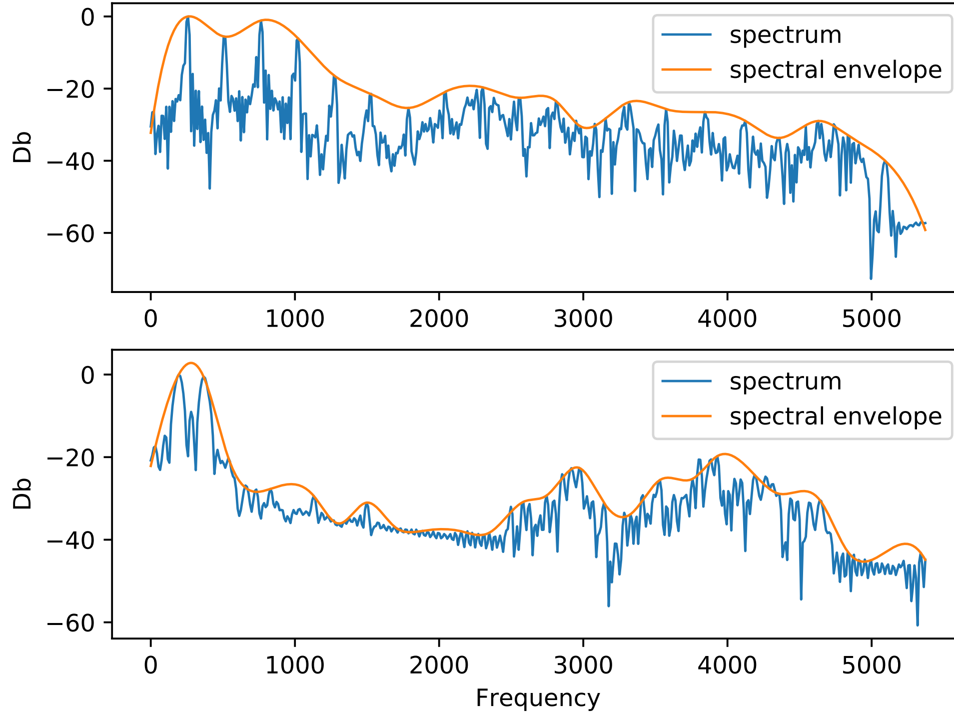

Figure 2.1: The spectra and approximate spectral envelopes for vowels “a” on the top and “e” on the bottom. The peaks correspond to resonating harmonics, while the valleys correspond to suppressed harmonics.

other. Sounds with distinguishable meaning are called *phonemes* and can be seen as basic units of sound. Phonemes are usually represented by phonetic symbols described by the International Phonetic Alphabet (IPA). For example, the phonematic representation of a word *explanation* would be [ˌɛkspləˈneɪʃən]. Phonematic word representation can be found in most language dictionaries.

Since phonemes correspond to distinguishable speech sounds, it can be convenient to first convert text for speech synthesis to phonemes, because there will probably be better sound-to-character correspondence. This conversion is called grapheme-to-phoneme conversion. First, numbers, dates and times written in numerc form are spelled out. For example, "the 4$^{\text{th}}$ of May" would be converted to "the fourth of May". Then, the graphemes are converted to phonemes either based on pronunciation dictionaries or predefined rules. Pronunciation can depend on context and further analysis may be necessary. This process is language-dependent. Phonemes can then be directly used as input to a TTS system.

2.3 Signal processing

Models in speech synthesis make heavy use of various preprocessing of the audio signal. This section provides a basic overview of short-time Fourier transform and further audio transformations that are utilized in our model. The definition of a short-time Fourier transform is as follows (Kehtarnavaz, 2008).

Definition 1. *Let $x(t)$ be a signal at time t and let f be a frequency. Let w be a window function. The continuous short-time Fourier transform of frequency f*

at time τ is expressed as follows:

$$STFT(\tau, f) = \int_{-\infty}^{\infty} x(t)w(t-\tau)e^{-ift}dt \tag{2.1}$$

The discrete short-time Fourier transform of frequency f at time step m is expressed as follows:

$$DSTFT(m, f) = \sum_{n=-\infty}^{\infty} x[n]w[n-m]e^{-ifn}dt \tag{2.2}$$

We further refer to the DSTFT as to STFT. The window function can be for example a Gaussian or Hann window function (Harris, 1978). In practice, the window function is truncated to be zero outside a specified interval around zero and the sum in the STFT calculation runs over a finite number of elements. In the discrete case, STFT can be seen as a table of complex numbers representing the phase and magnitude of each given frequency at a given time step of the signal. Short-time Fourier transform can be inverted back to the original signal domain with the so-called inverse STFT or simply iSTFT (Allen and Rabiner, 1977).

Definition 2. *A spectrogram of a signal is a squared magnitude of STFT.*

$$spectrogram(m, f) = |DSTFT(m, f)|^2 \in \mathbb{R} \tag{2.3}$$

where $|.|$ denotes absolute value. In case of complex numbers, $|.|$ can be viewed as the size of the complex number.

A spectrogram can be seen as a table where each row represents a discrete time step and each column represents a frequency bin. Among other things, spectrograms can be useful for displaying harmonic frequencies of a signal. Spectrograms can also be used to visualize human intuition behind a melody. In speech synthesis, spectrograms reduce the time dimension of the signal, which simplifies modelling of long-distance dependencies typical for spoken language such as intonation. A log-magnitude spectrogram is visualized in Figure 2.2.

Stevens et al. (1937) have shown that humans do not perceive frequency pitch on a linear scale. If we take frequencies f_0, $f_1 = 2f_0$ and $f_2 = 2f_1$, the perceived distance between f_0 and f_1 will be the same as between f_1 and f_2. If we choose for example $f_0 = 220$ Hz, then $f_1 - f_0 = 220$ Hz and $f_2 - f_1 = 440$ Hz. The absolute frequency distances must increase exponentially with frequency for humans to perceive the frequency changes as equal. This phenomenon can be used to further reduce the dimensionality of raw audio. The perceptual scale is called the *mel scale*. A spectrogram in Hertz scale can be transformed into mel scale with the following formula (Taylor, 2009, Chap 12, p. 360):

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

In practice, a mel-filter matrix is constructed from the above formula and the linear-scale spectrogram is multiplied by that matrix. We can extract a spectrogram and apply frequency filters with high detail in low frequencies and low detail in high frequencies. This process reduces high frequency precision, but shrinks the number of frequency bins. If the shrinking is not too drastic,

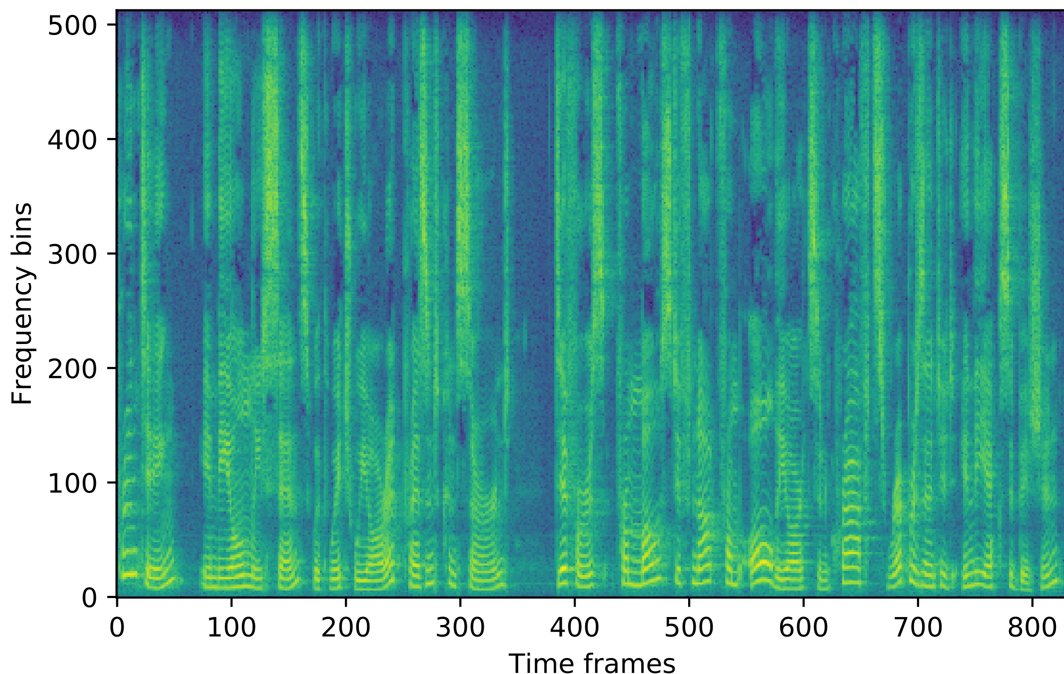

Figure 2.2: Log-magnitude spectogram with 512 frequency bins of a sentence *"Printing, in the only sense with which we are at present concerned, differs from most if not from all the arts and crafts represented in the Exhibition"*.

the reconstructed voice quality is still acceptable. This form of dimensionality reduction is useful because it allows us to train and use smaller models for speech synthesis. A log-magnitude mel spectrogram is visualized in Figure 2.3.

Exact spectrogram reconstruction from a mel spectrogram is impossible because the mel transformation discards information. However, a pseudoinverse of the mel transformation can be calculated to get approximate spectrograms. For example, we can find an approximate solution to the non-negative least squares (NNLS) problem (Chen and Plemmons, 2009) in order to reconstruct the linear-scale spectrograms. In other words, we want to solve $argmin_{\mathbf{x}}||\mathbf{ax} - \mathbf{y}||_2$ given $\mathbf{x} \geq \mathbf{0}$, where $\mathbf{x}$ is the spectrogram reconstruction we are looking for, $\mathbf{a}$ is the mel transformation matrix and $\mathbf{y}$ is the mel-spectrogram.

To invert a spectrogram back to the original signal, the phase of the STFT-

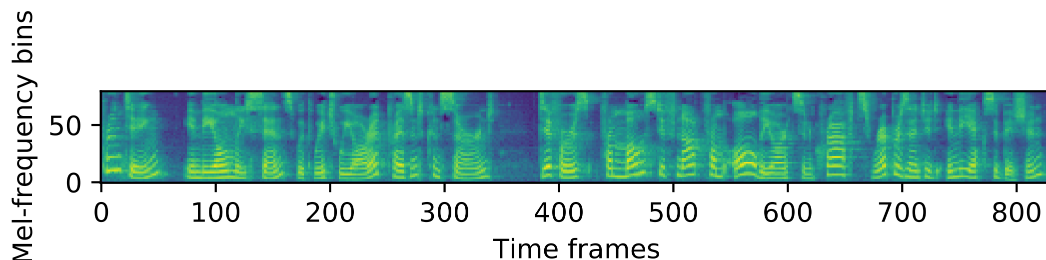

Figure 2.3: Log-magnitude mel spectogram with 80 mel frequency bins of a sentence *"Printing, in the only sense with which we are at present concerned, differs from most if not from all the arts and crafts represented in the Exhibition"*.

Algorithm 1: Griffin-Lim

Data: Linear scale magnitude spectrogram, number of iterations n
Result: Estimated signal
$phase \leftarrow$ RandomPhase;
$signal \leftarrow \text{iSTFT}(spectrogram, phase)$;
for $i \leftarrow 1$ **to** n **do**
    $real, phase \leftarrow \text{STFT}(signal)$;
    $signal \leftarrow \text{iSTFT}(spectrogram, phase)$;
end
return $signal$;

processed signal must be estimated in order to use iSTFT. Various phase estimation algorithms have been proposed, for example (Griffin and Lim, 1984) or (Le Roux et al., 2010). We use a slightly modified version of the Griffin-Lim algorithm depicted in Algorithm 1. The *phase* corresponds to the imaginary part of STFT. The spectrogram and a phase are inverted by iSTFT to form a signal. Then, the signal is processed by STFT. The imaginary part of the output is used to form a new estimate of the phase.

2.4 Probabilistic modelling

Conditional audio synthesis can be formulated in a probabilistic setting. Specifically, the dependence of the audio wave sequence on spoken text can be formulated as a conditional probability distribution

$$P(\mathbf{Y}|\mathbf{X}) = P(\vec{Y}_1, \ldots, \vec{Y}_n | \vec{X}_1, \ldots \vec{X}_k) \tag{2.5}$$

where $\mathbf{X}$ is a sequence of discrete symbols, such as text characters or phonemes and $\mathbf{Y}$ is a sequence of spectrogram or raw waveform frames.

2.4.1 Maximum likelihood estimation

Given a dataset of independent, identically distributed sequence pairs $(\mathbf{y}, \mathbf{x}) \in \mathcal{D}$, such distributions can be estimated from data with maximum likelihood estimation (Bishop, 2006, p. 27), which means solving the following problem:

Definition 3. *Let* $P(\mathbf{Y}|\mathbf{X})$ *be the true distribution and* $\hat{P}(\mathbf{Y}|\mathbf{X}, \vec{\theta})$ *be from a family of distributions* $\mathcal{P}_{\vec{\theta}}$*. The maximum log-likelihood estimate of* P *is*

$$\arg\max_{\vec{\theta}} \sum_{(\mathbf{y},\mathbf{x}) \in \mathcal{D}} \log(\hat{P}(\mathbf{y}|\mathbf{x}, \vec{\theta})) \tag{2.6}$$

The independence of the samples allows us to express the joint distribution over the entire dataset as a product of distributions over each sample. The logarithm is a monotonically increasing transformation. Therefore, it does not change the resulting $\vec{\theta}$ and allows us to optimize a sum instead of a product. In conclusion, a MLE estimate is an estimate such that the probability of observing the data under the estimated distribution is maximized.

2.4.2 Stochastic gradient descent

Typically, $\mathcal{P}_{\vec{\theta}}$ is selected so that $\hat{P}(\mathbf{y}|\mathbf{x}, \vec{\theta})$ is differentiable w.r.t. $\vec{\theta}$. Then, the optimal parameters $\vec{\theta}^*$ can be iteratively estimated with Monte-Carlo optimization techniques such as *stochastic gradient descent* (SGD) or some of its variants such as Adam (Goodfellow et al., 2016; Kingma and Ba, 2015).

Definition 4. *One iteration of SGD performs the following update*

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \rho_t \frac{1}{|\mathcal{D}|} \sum_{i=1}^{|\mathcal{D}|} \nabla_{\vec{\theta}} \log(\hat{P}(\mathbf{y}|\mathbf{x}, \vec{\theta})), \tag{2.7}$$

where ρ_t is a learning rate at step t satisfying the Robense-Monro convergence conditions (Robbins and Monro, 1951).

It is common that negative log-likelihood is minimized instead. In practice, the gradient update is not calculated across the entire dataset. The datasets used in Deep learning applications are often too large to efficiently calculate the gradient with respect to the entire dataset. Instead, batch updates are used. A batch is a random subset of the dataset and a batch update is a gradient update calculated with respect to the batch. It would be possible to run the SGD with respect to a single item from the dataset. However, the convergence in such case typically becomes less stable. The gradient calculation in neural networks (see Section 2.5) is calculated by means*backpropagation* (Rumelhart et al., 1986).

2.4.3 Chain rule factorization

If we define $\mathbf{Y}_{j<i} = (\vec{Y}_1, \ldots, \vec{Y}_{i-1})$ for $i > 1$, the chain rule can be used to factorize the distribution in Equation 2.5 into the following form:

$$P(\vec{Y}_1, \ldots, \vec{Y}_n | \vec{X}_1, \ldots \vec{X}_k) = \prod_{i=1}^{n} P(\vec{Y}_i | \vec{Y}_{j<i}, \mathbf{X}). \tag{2.8}$$

The factorized distribution is often easier to model. It allows parameter sharing – the model can use the same parameters to express each time step of the target series conditioned on already expressed time steps. The choice of distribution family $\mathcal{P}$ can have a large impact on the resulting estimate. If $\mathcal{P}$ contains the true distribution, the MLE estimate has the following suitable properties:

1. The estimate is consistent, i.e., it converges in probability to the true value of the estimated parameter as the data set size increases.

2. The estimate is efficient, i.e., its variance is lowest possible among unbiased estimators.

2.4.4 Loss functions

To model complex distributions, one needs highly expressive distribution families to get close to the true distribution. Let us assume that the sequence lengths of $\mathbf{X}$ and $\mathbf{Y}$ are equal. By restricting the the distribution family to a certain group of distributions, it is often possible to express the minimization of the negative

log-likelihood as a minimization of a corresponding loss function. It is common to take some family of parametric distributions such as the Gaussian family and use functions $\mu_{\vec{\theta}_1} : \mathbf{X} \rightarrow \mathbb{R}^{k \times n}$ and $\sigma_{\vec{\theta}_2} : \mathbf{X} \rightarrow \mathbb{R}^{+k \times n}$ to characterize the distribution mean and variance (or other parameters of the given distribution). Then the target variable is modeled as follows.

$$\mathbf{Y}|\mathbf{x} \sim \mathcal{N}(\mu_{\vec{\theta}_1}(\mathbf{x}), \sigma_{\vec{\theta}_2}(\mathbf{x})) \tag{2.9}$$

If we set $\sigma_{\vec{\theta}_2}(\mathbf{x}) = \mathbf{1}$, plugging the Normal distribution into negative log-likelihood results in minimizing the L_2 loss (Bishop, 2006, p. 27). In other words, we can directly minimize the mean squared error (MSE):

Definition 5. *The mean squared error for all pairs* $\mathbf{x}, \mathbf{y} \in \mathbb{R}^{k \times n}$ *in the dataset* $\mathcal{D}$ *is defined as:*

$$\frac{1}{|\mathcal{D}|} \sum_{\mathbf{x},\mathbf{y} \in \mathcal{D}} \frac{||\mu_{\vec{\theta_1}}(\mathbf{x}) - \mathbf{y}||_2^2}{kn}, \tag{2.10}$$

where $||\mathbf{z}||_2 = \sqrt{\sum_{i=1}^{k} \sum_{j=1}^{n} \mathbf{z}[i,j]^2}$.

Similarly, if we model $\mathbf{Y}|\mathbf{x}$ with the Laplacian distribution, we can directly minimize the negative log-likelihood by minimizing the L_1 loss, which results in minimizing the mean absolute error (MAE):

Definition 6. *The mean absolute error for all pairs* $\mathbf{x}, \mathbf{y} \in \mathbb{R}^{k \times n}$ *in the dataset* $\mathcal{D}$ *is defined as:*

$$\frac{1}{\mathcal{D}|} \sum_{\mathbf{x},\mathbf{y} \in \mathcal{D}} \frac{||\mu_{\vec{\theta_1}}(\mathbf{x}) - \mathbf{y}||_1}{kn}, \tag{2.11}$$

where $||\mathbf{z}||_1 = \sum_{i=1}^{k} \sum_{j=1}^{n} |\mathbf{z}[i,j]|$ *is a sum of absolute values of the matrix* $\mathbf{z}$.

The distribution parameters μ are often parameterized with differentiable models such as the neural networks.

2.5 Neural networks

Neural networks are seen as differentiable universal function approximators (Hornik et al., 1989). A typical neural network combines affine transformations of the input with non-linear activation functions, such as ReLU, hyperbolic tangens or sigmoid. For example, the ReLU activation is defined as $f(x) = \max(0, x)$. Modern architectures can also include normalization (Ioffe and Szegedy, 2015), pooling layers (Scherer et al., 2010), residual connections (He et al., 2016), attention mechanisms (Bahdanau et al., 2015) and so on. In the following, we will briefly describe common types of currently used neural network architecture components, including:

- convolutional layers that make use of local dependencies in the data,
- recurrent layers that maintain a hidden state which allows them to model longer-distance dependencies in some data sequence,
- attention mechanism that allow the network to access any part of the modeled sequence,

- residual skip connections that enable training of very deep neural networks,
- batch normalization that allows more stable training and faster training convergence.

Parts of these architectures will then be used in our models described in Chapter 4. For a comprehensive overview of neural architectures, we refer the reader to Goodfellow et al. (2016).

2.5.1 Convolutional and recurrent networks

Both recurrent and convolutional neural networks can be used to model Equation 2.8. Convolutional networks consist of several convolutional layers. The layers are sequentially ordered and each layer processes the outputs of the previous layer. We work with temporal input sequences that are usually multidimensional; each point in the sequence has several dimensions, sometimes called *channels*. A convolutional layer uses the same set of parameters to transform different parts of the input sequence. Each layer has its own parameter set. The trainable parameters consist of a set of matrix filters that are also called kernels. Since we work with temporal data, the convolution layers in our models use 1D (temporal) convolutions. The 1D convolution kernel set is usually represented as a fixed-sized three-dimensional array of size (n, m, k), where n is the number of output channels, m is the number of input channels and k is the kernel size. We can also interpret the kernel set as a list of n matrices of size $(m \times k)$. The convolutional transformation is similar to a sliding window filter. The part of the input sequence under the sliding window is element-wise multiplied by each matrix in the convolutional kernel. For each matrix multiplication, the resulting scalars are summed and the summands are concatenated to form an output vector with n items. The result is typically transformed by some non-linear function such as ReLU. The convolutional operation is depicted in Figure 2.4.

The convolutional filters slide across the input sequence with some predefined step size (also called stride). The length of the output sequence depends on the step size, kernel size and dilation. The dilation factor determines the size of the gaps between columns of the filters. A dilation of 1 corresponds to no gaps between the filter columns (see Figure 2.5). Increased dilation, step size and kernel size decrease the length of the output sequence. To keep the output sequence length the same as the input sequence length, the input sequence can be zero-padded at the beginning and at the end. A dilated convolution with padding is displayed in Figure 2.5.

Due to the sliding window approach, convolutional networks use a restricted receptive field to model the sequence. Receptive field is a segment of input sequence that corresponds to one item in the output sequence. For example, in Figure 2.4, the receptive field contains 3 input tokens. Thus, the size of the receptive field is 3. A restricted receptive field means that $\mathbf{Y}_{j<i}$ contains only a limited number of recent timesteps and not necessarily the whole sequence $\{\vec{Y}_1, \dots \vec{Y}_{i-1}\}$. This may limit the ability of the model to capture long-term dependencies. Increasing the kernel size and the dilation factor can be used to extend the receptive field. Larger receptive field allows the network to extract

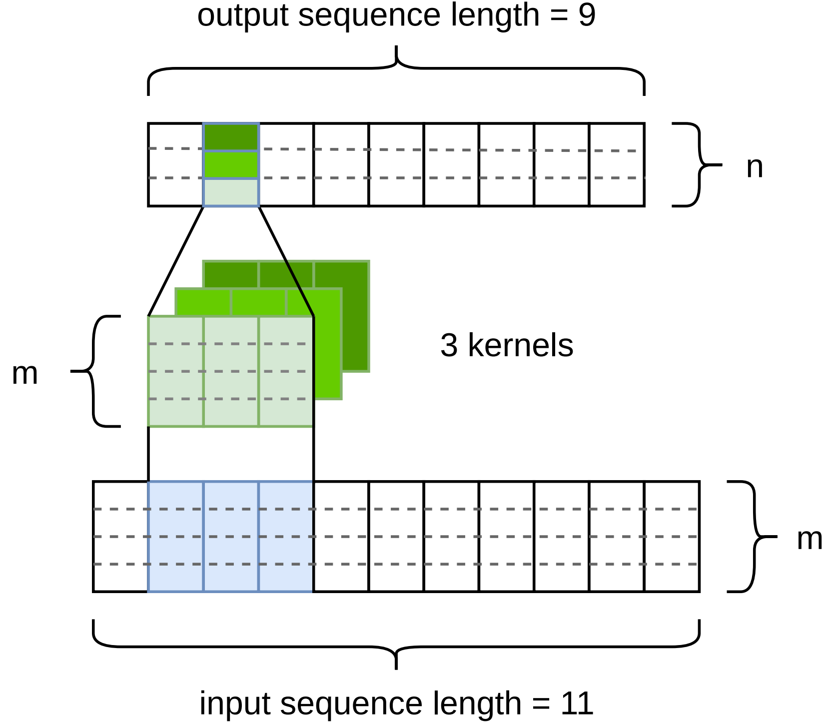

Figure 2.4: One-dimensional non-causal convolution with four input channels, three output channels and three kernels. The green filters transform the input sequence (bottom) to output sequence (top). The length of the output sequence is decreased due to kernel size larger than 1.

features across longer distances and can capture long-term dependencies such as voice intonation.

To model the Equation 2.8 with convolutional networks, it is necessary to carefully design the receptive field of the network in order to prevent the network from looking at the future timesteps. This can be achieved by padding the beginning of the input sequence with a sequence of zeros, where the length of the zero sequence equals the size of the receptive field - 1. Convolutions with a receptive field shifted in this way are called *causal*. A causal convolution is displayed in Figure 2.6.

Recurrent neural networks capture the past time steps $\mathbf{Y}_{j<i}$ in a finite compressed representation called the *hidden state*. The distribution becomes:

$$P(\vec{Y}_1, \ldots, \vec{Y}_n | \vec{X}_1, \ldots \vec{X}_k) = \prod_{i=1}^{n} P(\vec{Y}_i | \vec{Y}_{i-1}, \vec{H}_{i-1}, \mathbf{X}) \tag{2.12}$$

The hidden state vector $\vec{H}_i$ is internally updated at every time step and serves as a memory of the network.

During training, the target sequence $\mathbf{y}$ is known for each data point. We can use the sequence $\mathbf{y}$ as the input to the model and use the same sequence on the output, but with each token shifted one position to the right. For convolutional networks this means that for each member of the product in Equation 2.8 we know both inputs and outputs and each member can be evaluated independently and in parallel.

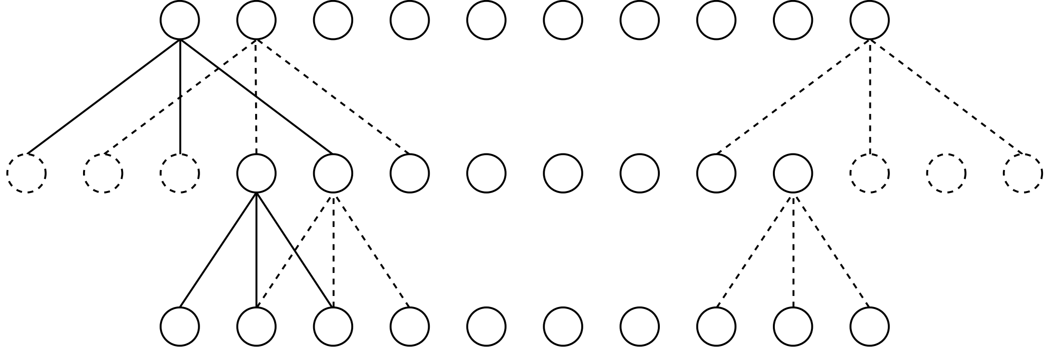

Figure 2.5: Dilated temporal non-causal convolutions with dilations 1 (bottom layer) and 3 (top layer), kernel size 3 and stride 1. Dashed circles represent zero padding necessary to keep the same output size in each layer. For visualisations of dilated temporal causal convolutions, see (Oord et al., 2016).

During sampling from the model conditional on some $\mathbf{X}$, the sequence $\mathbf{Y}$ is unknown and has to be iteratively generated. We start by sampling $\vec{y}_1$ from $P(\vec{Y}_1|\mathbf{x})$. Then we sample $\vec{y}_2$ from $P(\vec{Y}_2|\vec{y}_1, \mathbf{x})$ and so on. Models that iteratively generate the output are called *autoregressive*.

Figure 2.6 shows a computational graph for a convolutional neural network with a receptive field of size $k-1$, i.e., each output token is generated based on the previous $k-1$ tokens.

A computational graph for a recurrent neural network can be seen in Figure 2.7. The hidden state $\vec{h}$ is updated at each time step. The sequence of hidden states is not known beforehand and has to be sequentially calculated even during training, which hinders parallelization across time.

The way the model conditions on $\mathbf{X}$ is one of the most important distinctions between various architectures. For the task of speech synthesis, we assume that each token in $\mathbf{X}$ corresponds to a consecutive sequence of frames in $\mathbf{Y}$. The number of frames corresponding to a token is called the token *duration*.

Example. Let $D = \{d_i\}_{i=1}^k$, $d_i \in \mathbb{N}$ be a sequence of durations corresponding to tokens in $\mathbf{X}$ such that $\sum_{i=1}^k d_i = |\mathbf{Y}|$. We can create a new sequence of tokens:

$$\mathbf{X}' = \{\underbrace{\vec{X}_1, \ldots, \vec{X}_1}_{d_1 \text{ times}}, \underbrace{\vec{X}_2, \ldots, \vec{X}_2}_{d_2 \text{ times}}, \ldots\} \tag{2.13}$$

I.e., $\vec{X}_1$ is copied d_1 times and so on, so that $|\mathbf{X}'| = |\mathbf{Y}|$.

If the token durations are available, we can simply associate the elements of $\mathbf{X}'$ with corresponding frames of $\mathbf{Y}$. This is visualised in Figure 2.8 for a recurrent network. This can be done analogously for a CNN.

This approach is taken for example in the WaveNet TTS model (Oord et al., 2016), where other linguistic features are used in addition to input orthographical characters (see Chapter 3 for details). WaveNet uses dilated 1-D convolutions as its building block and uses a computational graph similar to Figure 2.6.

Token durations can be extracted with publicly available force-aligners such as the Montreal Forced Aligner (McAuliffe et al., 2017). Such aligners either use a pretrained speech recognition system or a combination of a pretrained speech synthesis system and a dynamic time warping algorithm (Salvador and Chan, 2004).

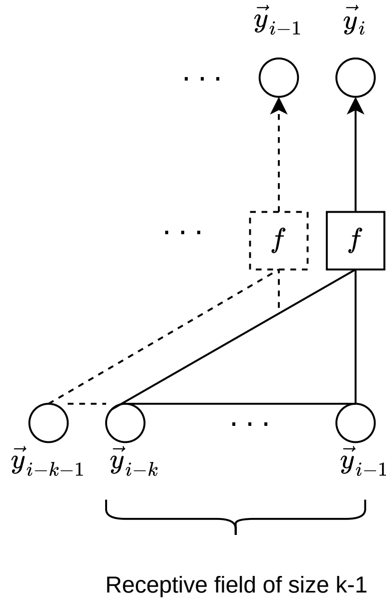

Figure 2.6: A computational graph for a temporal convolutional network with causal convolutions. The function f maps a window of input elements to an output element. For example, the function f can be composed of a convolution operation, normalization and non-linear activation such as ReLU. We omit the dependence on $\mathbf{X}$ for simplicity.

2.5.2 Attention

Attention (Bahdanau et al., 2015) can be used to automatically align the text tokens with audio frames. The concept can be understood in the context of recurrent neural networks but can be easily generalized for other architectures as well.

Let us consider that we want to generate frame $\vec{y}_i$ from $\vec{h}_{i-1}$, $\vec{y}_{i-1}$ and $\mathbf{x}$. To incorporate the information from the whole sequence $\mathbf{x}$, we would like to select a certain frame $\vec{x}$ (dubbed *context vector*) that would provide relevant information about $\vec{y}_i$. For example, $\vec{x}$ could be the token/character pronounced in frame $\vec{y}_i$. To do so, we can define a score function $score(\vec{h}_{i-1}, \vec{x}_j) \in \mathbb{R}$ and calculate the score of each $\vec{x}_j \in \mathbf{x}$ for given h_{i-1}. Ideally, we would select the $\vec{x}$ with the highest score as the context vector. Unfortunately, the operation is not differentiable and gradient is not able to flow through, which hinders the use of SGD. Instead, we can define a sequence of distributions $\mathbf{a} = \{\vec{a}_1, \dots \vec{a}_{|X|}\}$ over the elements of $\mathbf{x}$ with respect to the calculated scores. A softmax function can be used to normalize the scores into a probability distribution.

$$\mathbf{a}[i,j] = \frac{\exp\left(\text{score}(\vec{h}_{i-1}, \vec{x}_j)\right)}{\sum_j \exp\left(\text{score}(\vec{h}_{i-1}, \vec{x}_j)\right)} \tag{2.14}$$

Each column of the matrix $\mathbf{a}$ is a distribution over columns of $\mathbf{x}$. We can extract the context vector from $\mathbf{x}$ as a weighted average of $\mathbf{x}$.

$$\vec{c}_i = \vec{a}_i \cdot x = \sum_j \mathbf{a}[i,j] \cdot \vec{x}_j \tag{2.15}$$

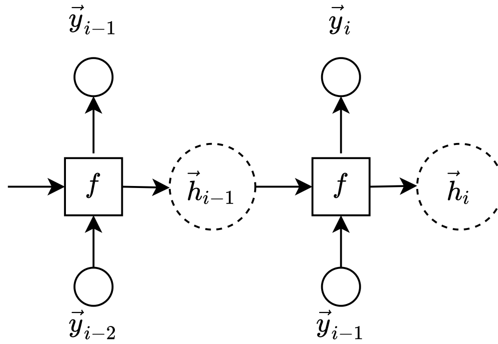

Figure 2.7: A computational graph for a recurrent network. The variable h_i represents the hidden state carried between timesteps. The function f maps the hidden state from the previous timestep $\vec{h}_{i-1}$ and the input $\vec{y}_{i-1}$ at current timestep to output $\vec{y}_i$. We omit the dependence on $\mathbf{X}$ for simplicity.

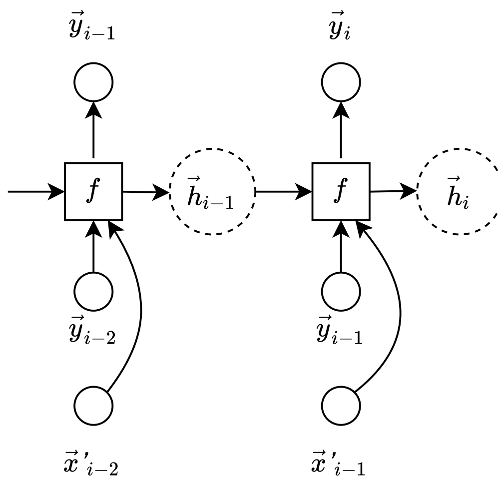

Figure 2.8: A computational graph for a conditioned recurrent network.

This operation is already differentiable and we can use the resulting vector as a context vector for generation of y_i. There are various score functions. In this thesis, we use the scaled dot-product attention (Vaswani et al., 2017):

Definition 7. *The scaled dot-product attention score is calculated as:*

$$\text{score}(\vec{h}_{i-1}, \vec{x}) = \frac{\vec{h}_{i-1} \cdot \vec{x}}{\sqrt{d}} \tag{2.16}$$

where d is the size of vector x.

The elements in the numerator are vectors, the denominator is a scalar and its value equals the number of dimensions of the vectors. The scaled dot-product attention has several advantages. It is easy to calculate, easy to parallelize and can be generalized for non-autoregressive models.

The attention mechanism can be seen as a dictionary with keys and values. We can imagine $\vec{h}$ to be a query. We also have a set of keys $\mathbf{x}$; the keys in $\mathbf{x}$ have values associated to them in theory, but in practice, $\mathbf{x}$ itself is used as values, i.e., keys = values. We compare the query with the available keys and extract the

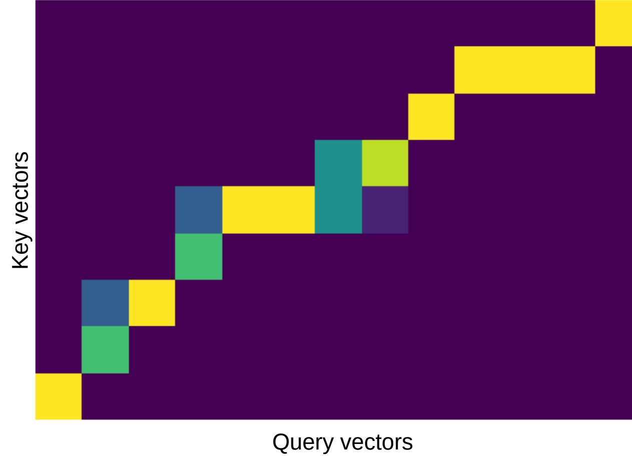

Figure 2.9: An attention distribution for each timestep. The horizontal axis represents the query sequence. The vertical axis represents the key/value sequence. Each column forms the attention distribution over keys for the given query. Dark color is close to zero, light color is close to 1. We can extract the alignment of keys and queries by taking the argmax of each column.

values whose keys were most compatible with the query. If multiple queries are available, we can technically compute attention for each of the queries in parallel.

In a special case where keys = queries = values, the attention becomes so called *self-attention*, because for each element in the sequence, we are looking for compatible elements in the same sequence. This principle is used in the Transformer model (Vaswani et al., 2017) and in this thesis.

Scaled dot-product attention can be described as a matrix product:

Definition 8. *Let* $\mathbf{q} = [\vec{q}_1, \ldots, \vec{q}_m]$*,* $\mathbf{k} = [\vec{k}_1, \ldots, \vec{k}_n]$*,* $\mathbf{v} = [\vec{v}_1, \ldots, \vec{v}_n]$ *be matrices of queries, keys and values. The context matrix* $\mathbf{c}$ *is calculated as follows:*

$$\mathbf{c} = \mathbf{v} \cdot \mathrm{softmax}\left(\frac{\mathbf{k}^T \mathbf{q}}{\sqrt{d}}\right) \tag{2.17}$$

where d *is the size of vectors in* $\mathbf{k}$*. The result of the softmax operation is a matrix of size* $n \times m$*. The softmax is applied on each column of the matrix* $\mathbf{k}^T\mathbf{q}$*. Therefore, each column is normalized and can be treated as a probability distribution.*

An example attention distribution is visualised in Figure 2.9. We can extract the alignment between keys and queries by taking the argmax of each column. This gives us a key index for each query, which can be considered an alignment of keys to queries. If the key indices stay the same or increase, but never decrease with the query index (the alignment is non-decreasing), we can extract the duration for each key by counting the number of occurrences of each key index in the alignment.

2.5.3 Positional encoding

The attention layers are not able to distinguish relative locations of the input. For example, it is hard for an attention layer to "look" 2 positions to the left

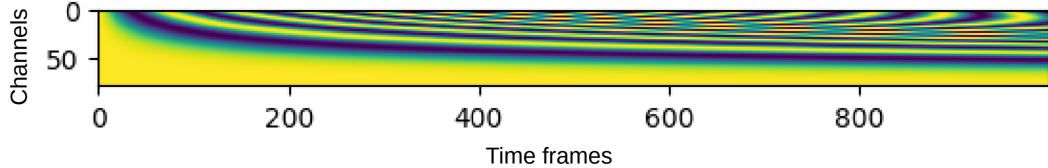

Figure 2.10: The positional encoding matrix. The horizontal axis represents time. The vertical axis represents dimensionality of the encoding.

from a given input vector because there is no location information encoded in the input vectors and the attention operation is non-local. The positional encodings Vaswani et al. (2017) were designed to solve this issue by adding a unique positional timestamp to each input vector.

Definition 9. *The positional encoding is a sequence of vectors defined as follows.*

$$pe_{(\mathrm{pos},2i)} = \sin\left(\frac{\mathrm{pos}}{c^{2i/\mathrm{channels}}}\right) \tag{2.18}$$

$$pe_{(\mathrm{pos},2i+1)} = \cos\left(\frac{\mathrm{pos}}{c^{2i/\mathrm{channels}}}\right) \tag{2.19}$$

The first subscript is a position of the vector in a sequence and the second argument is the item position inside the vector. The whole sequence can be seen as a matrix. The second subscript in pe corresponds to the dimensionality of the encoding. We use as many dimensions as there are channels in our models so that the encoding can be added to the layer inputs. The constant c is set to 10,000.

The positional encoding matrix is visualised in Figure 2.10. Vaswani et al. (2017) hypothesized that accessing relative positions in the sequence could be easy to learn, because $pe_{pos+k,2i}$ can be expressed as a linear function of $pe_{pos,2i}$. The positional encoding could also be learned by the network, but the learned encoding would have a fixed size and could not be used for sequences longer than the sequences seen during training. In fact, Vaswani et al. (2017) decided to use the variant in Definition 9 for that purpose.

2.5.4 Residual networks

Stochastic gradient descent requires gradient computation for all trainable parameters in the model. In the case of neural networks, the gradient is calculated by means of *backpropagation* (Rumelhart et al., 1986). As the depth of the network increases, the backpropagation algorithm becomes less effective at propagating the gradient through the network, which results in slower training. To allow training of deeper neural networks, the so-called *residual connections* between consecutive layers are used. The resulting networks are called Residual networks. We use a variant of a residual network in the spectorgram synthesis part of our model (see Section 4.4). Residual neural networks were first introduced by He et al. (2016) and are technically a simplified version of highway networks (Srivastava et al., 2015). Instead of directly transforming the input

$$\mathbf{x}_n = \mathcal{F}(\mathbf{x}_{n-1}) \tag{2.20}$$

a residual block calculates a residual

$$\mathbf{x}_n = \mathcal{F}(\mathbf{x}_{n-1}) + \mathbf{x}_{n-1} \tag{2.21}$$

The index n of the matrix $\mathbf{x}_n$ denotes that $\mathbf{x}_n$ is the hidden representation of the n-th layer of the residual network. The mapping $\mathcal{F}$ can be composed of various combinations of layers such as convolutional layers, normalizations and activation functions. The residual connection allows the gradient to flow to deeper layers of the residual network during training, which allows training of very deep networks. Residual blocks can be stacked on top of each other.

2.5.5 Batch normalization

Neural networks learn an output distribution based on an input distribution. Similarly, the network layers learn some output distribution given the input from the previous layer. While the distribution of the network inputs usually does not change, the outputs of the network layers change during the learning process. This means that a layer inside the network has to learn some output distribution based on non-stationary input, due to the changes in the previous layer. This effect is called an *internal covariance shift*. This is problematic because learning an output distribution based on non-stationary input can be comparatively harder than learning the output distribution when the input distribution is constant. Various techniques have been developed to decrease the effect of input non-stationarity. One of the most popular approaches is *batch normalization* (Ioffe and Szegedy, 2015).

This technique normalizes the layer inputs to have zero mean and unit variance. The goal of batch normalization is to keep the mean and variance of the input distribution constant to enable faster and more stable learning. However, input normalization can decrease expressiveness of the network. To restore the network expressiveness, the normalized inputs are transformed by an affine transformation. The parameters of the affine transformation are learned during training. Please see Ioffe and Szegedy (2015) for details.

We work with mel spectrograms (see Section 2.3), i.e., 3D arrays of the following dimensions: training batch, time, mel-frequency bins. Similarly, the intermediate layer outputs in our networks have three dimensions: batch, time, channels (i.e., results of convolutions on the preceding layer). Therefore, we use a 1D batch normalization for sequences (temporal batch normalization) applied to tensors with the batch, time and channels dimensions. In this case, the normalization is applied over the (batch, time) dimensions for each channel. In our case, this amounts to normalizing each frequency bin across time and batch.

3. Literature review

There is a long history of attempts to synthesize human voice. Early attempts usually tried to mechanically simulate the human vocal tract. One such device was invented by Wolfgang von Kempelen in 18th century (Dudley and Tarnoczy, 1950). A more recent example is a so called Vocoder invented by Homer Dudley at Bell labs in 1938 (Dudley, 1938). Vocoder originally served as a voice encoder-decoder and allowed transmission of encrypted human voice via radio communication. Later it was demonstrated that with a human-operated keyboard, the encoder part of Vocoder–so called Voder–was capable of speech synthesis. In the last five decades, speech synthesis made a significant progress, especially in naturalness and intelligibility of the outputs – two main characteristics of a TTS system's quality. The methods used to reach high quality synthesis can be divided into non-neural and neural approaches. We mainly focus on neural approaches and describe the most common techniques in Section 3.2. We also describe non-neural approaches in Section 3.1 mainly for historical context.

3.1 Non-neural approaches

In this section, we discuss concatenative synthesis, formant-based synthesis and synthesis based on hidden Markov models.

Concatenative synthesis (Sagisaka, 1988) is a synthesis method that works with snippets of recorded speech. A human speaker is recorded saying various sentences. The recorded sentences are divided into predefined speech units and saved. In this way, a database of sounds and corresponding phonemes is created. During synthesis, the text to be synthesized is transformed into phonemes. Next, corresponding sounds in the sound database are retrieved and concatenated to form the synthesized audio (Schwarz, 2005). This method can require relatively large sound database depending on the granularity of the speech units. The longer the speech utterances are, the more utterances are required to cover a reasonable set of spoken sentences. On the other hand, the shorter the speech utterances are, the more concatenation points are created in the synthesized speech which may lead to lower quality. Usually diphones (transitions between two consecutive phones) or longer segments involving multiple consecutive phones are used. Single phonemes are rarely used as the phoneme transitions are more noticeable in the final audio. It is beneficial to also measure the fundamental frequency and loudness of each utterance to be able to select combinations of samples that better match on the audio borders. The festival project[1] contains engines for concatenative synthesis.

Formant synthesis, or rule-based synthesis (Klatt, 1980), does not use a database of human audio samples, but instead generates the waveform by adding sine waves of different frequencies and amplitudes and a non-periodic noise together and passing the resulting signal through series of filters that model the

[1] `http://www.cstr.ed.ac.uk/projects/festival/`

vocal tract (Klatt, 1980, 1987). It is necessary to extract formants and other features for each phoneme from human speakers to form a phoneme-formant table. Usually, the first three formants are extracted, but more formants can be also used. The extracted formants are used to set the parameters of the filters. For each synthesized phoneme, the filters are adjusted so that the formants of the filtered signal correspond to the extracted formants of the given phoneme. Unlike in concatenative synthesis, a formant synthesizer can in theory synthesize any sound. This allows the system to synthesize arbitrary words. However, the voice may sound unnatural due to simplifications in the synthesis process. On the other hand, such synthesizer usually has low computational requirements and can be deployed on embedded devices. An example formant synthesis system is espeak-ng[2]. Espeak is an open source speech synthesis project and can be deployed on all major platforms.

Hidden Markov Models: Another popular method for speech synthesis is based on hidden Markov models (HMM) (Bishop, 2006) called statictical parametric speech synthesis (Zen et al., 2009). Hidden Markov models are graphical models designed to model sequential data. In context of speech synthesis, the hidden states of an HMM can represent phonemes corresponding to the synthesized sentence, pitch accent and lexical stress. The observed variables are usually the fundamental frequency and spectral parameters such as mel frequency cepstral coefficients (MFCC)–a discrete cosine transform of mel-scale log spectrogram (see Section 2.3). Even though we know the phonemes corresponding to the audio, the alignment of the phonemes and the spectral parameters is unknown. We can define possible phoneme-to-phoneme transitions, but we need to infer the corresponding transition probabilities from data. Since the hidden state visited by the HMM at time t corresponds to a phoneme, it can be interpreted as if the model was reading the given phoneme at time t. The same phoneme can be read consecutively several times depending on how long it takes to pronounce the given phoneme. The phoneme durations can be extracted based on how many timesteps the model looked at the given phoneme. In more advanced systems the phoneme duration is explicitly modeled with Gaussian distributions (Zen et al., 2004). The spectral parameters, fundamental frequency and other features are vocoded by e.g. a MLSA filter into synthesized audio (Imai, 1983). This synthesis method allows better control over the speaking style and speaker emotions and has been the state of the art in speech synthesis before the use of deep neural networks. Please see Tokuda et al. (2013) for a comprehensive review of the technique. An example system is Flite[3].

3.2 Neural approaches

Models utilizing deep neural networks started to emerge around the year 2016 starting with WaveNet (Oord et al., 2016). Given sufficient amount of data and computational resources, some of the neural models are able to synthesize high quality audio, overcoming some of the problems of the earlier approaches. The

[2] `https://github.com/espeak-ng/espeak-ng`
[3] `http://flite-hts-engine.sp.nitech.ac.jp/index.php`

neural models take advantage of various network architectures and work with different kinds of data, each of which have their advantages and disadvantages with respect to different use cases.

This section serves as a detailed review of the most influential neural synthesis architectures, focusing on approaches related to our own work. We can divide the models by synthesis scope, alignment mechanism and inference and training mechanism.

3.2.1 Synthesis scope

Synthesis can be done on various scopes. Different models usually implement different parts of the speech synthesis pipeline described in Section 2.1. We divide the models into three groups:

- End-to-end (text-to-waveform)
- text-to-spectrogram
- spectrogram-to-waveform

Models from the second group can be combined with models from the third group to form an end-to-end system. The models from the first group are standalone end-to-end systems that transform phonemes or ortographic characters(text) directly along with external features such as phoneme durations to raw waveforms. An example of an end-to-end system is WaveNet (Oord et al., 2016). The models from the second group predict spectrograms from phonemes or ortographic characters. For example, the Tacotron 2 model (Shen et al., 2018) accepts text as input and outputs mel-scale spectrograms. The third group contains vocoder-like models such as WaveGlow (Prenger et al., 2019). Vocoders usually accept spectral features such as mel-scale spectrograms on the input and output speech as a raw waveform. A categorization of prominent recent neural TTS models based on synthesis scope is shown in Table 3.1.

Note that the same model can have multiple uses, e.g., WaveNet modified with upsampling layers can also be used as a vocoder and is used as such in conjunction with Tacotron 2. There are also TTS models not covered by this scheme, such as systems of Sheng and Pavlovskiy (2019) or Neekhara et al. (2019) taylored to improve quality of already generated spectrograms. Finally, there are also models aimed at unsupervised audio synthesis (no conditional input provided), such as WaveGan (Donahue et al., 2019).

3.2.2 Alignment mechanism

Transforming a sequence of characters into a sequence of either spectrogram or waveform frames requires a mapping from a shorter to a longer sequence. To generate the output frame sequence, it is beneficial to know how many frames on the output correspond to each character on the input.

Attention-based approaches: One option of achieving this knowledge is to use some form of the attention mechanism (Bahdanau et al., 2015) (see Section 2.4) to learn the alignments from character-spectrogram pairs. This approach

End-to-end	WaveNet (Oord et al., 2016) Parallel WaveNet (Van Den Oord et al., 2018) WaveRNN (Kalchbrenner et al., 2018) ClariNet (Ping et al., 2019) GAN-TTS (Bińkowski et al., 2019) Char2Wav (Sotelo et al., 2017)
text -to-spectrogram	Tacotron 2 (Shen et al., 2018) Deep Voice 3 (Ping et al., 2018) FastSpeech (Ren et al., 2019) Efficiently trainable TTS (Tachibana et al., 2018)
spectrogram-to-waveform	MelGAN (Kumar et al., 2019) WaveGlow (Prenger et al., 2019) WaveNet (Oord et al., 2016) * WaveRNN (Kalchbrenner et al., 2018) *

Table 3.1: A list of selected neural TTS models divided into categories based on their synthesis scope. The systems marked with "*" were originally designed as end-to-end but with slight modifications can be used to synthesize audio from spectrograms.

is used for example by Shen et al. (2018), Ping et al. (2018) and Tachibana et al. (2018). It has the advantage that the data does not need to contain explicitly annotated alignments. Speech synthesis in essence simulates reading aloud. One does not need to re-read characters they have already seen. This intuition tells us that the learned alignments should be monotonic. Unfortunately, naive uses of the attention mechanism often lead to word repetitions and word skipping. Therefore, the attention-based systems usually use some sort of modified attention that ensures monotonicity of the alignment, either already at train time or only during inference. Tacotron 2 Shen et al. (2018) benefits from the so-called location-based attention. This form of attention does not allow the model to attend to already processed characters and enforces monotonic alignment. However, it requires the model to be sequential at both train and inference time, which usually slows down the operation of the model.

Deep Voice 3 Ping et al. (2018) uses positional encoding (see Sections 2.5.3 and 4.3.2) from (Vaswani et al., 2017) as a preconditioner for the attention during training. This allows faster convergence of the model and does not require sequential pass during training time. During inference, heuristic location-based attention is used.

Tachibana et al. (2018) use guided attention for training. This involves adding an extra loss function that penalizes attention alignments too far from the diagonal and consequently also non-monotonic alignments. Similarly to Deep Voice 3, this model does not require a sequential pass during training. However, same as Tacotron 2 and Deep Voice 3, this model generates output sequentially during inference.

FastSpeech (Ren et al., 2019) is able to generate output in parallel during both training and inference. First, a teacher network is trained, which is parallel during training but sequential during inference. Then, alignments produced by the teacher network are extracted and saved. Finally, the second feed-forward

model is trained. The second model learns to encode the input characters. Based on the encoding, the duration for each character is predicted. Then each encoding is copied to the decoder input multiple times according to the duration prediction. Thus, the decoder input has the same length as the required output sequence and can be processed in parallel.

Approaches based on linguistic features: Most of the end-to-end models do not use an attention mechanism for alignment and rely on linguistic features instead. Linguistic features primarily include phoneme durations extracted from a HMM-based force-aligner or a pretrained ASR system, and a prediction of the fundamental frequency F0, which helps with intonation. Basically, those models require ground-truth alignments to be part of the training data. Knowing the character durations allows the models to simply upsample the input to the required length and then transform the input sequence to the output sequence of the same length. Models using linguistic features are for example WaveNet (Oord et al., 2016) and GAN-TTS (Bińkowski et al., 2019).

Vocoders/spectrogram enhancers: Models that serve as vocoders or spectrogram enhancers do not require knowledge of character durations. When transforming spectrograms to waveforms, it is possible to calculate the required audio waveform length based on window size and hop length of the STFT used. Spectrogram enhancement usually does not change the time dimension of the spectrogram; it only adjusts the frequency bins.

3.2.3 Training and inference: Performance issues

We have already touched upon training and inference strategies of some models with regard to the attention mechanism. The training strategy determines computational requirements and training time of the model, which determines whether it is possible to train the model in restricted conditions. The inference strategy determines the usability of the model as a part of a real-time speech synthesis system or a system designed to answer a high volume of requests in a short time. The former is required in real-time dialogue systems, the latter may be required for large-scale batch synthesis.

Sequential models usually model the conditional distribution of the next timestep with recurrent neural networks (see Section 2.5.1), where the history of generated frames is implicitly represented in the hidden state of the network. Such models are typically sequential at both training and inference time and produce one spectrogram/audio frame per timestep, which results in longer training time due to limited parallelizability of the computations. The model size also often requires multiple GPUs to reach sufficient batch size (Shen et al., 2018). During inference, the models are again sequential and the speed of inference depends on the form of the predicted audio. Tacotron 2 (Shen et al., 2018) predicts mel-scale spectrograms, which typically contain around 1,000 time frames per sentence. With some hardware optimizations, it is possible to reach near-real-time inference speed (Zhang and Karch, 2019). On the other hand, WaveRNN (Kalchbrenner et al., 2018) generates raw audio, which requires generating 22,050 frames per

second in case of the usual 22,050 Hz audio sampling rate. Thus, it can take a prohibitively long time to generate one sentence of audio with WaveRNN.

Models such as Deep Voice 3 (Ping et al., 2018) and EfficientTTS (Tachibana et al., 2018) drop the use of a hidden state and model the conditional distribution with dilated temporal 1D convolutions (see Section 2.5.1) instead. Such models are still sequential at inference. However, since the hidden state from previous time step is not needed and the entire input and output are known during training, these models can generate all output frames in parallel.

Models formulated as autoregressive flows also allow better parallelization at training time, but require the input and output sequences to have the same length (Kobyzev et al., 2019). WaveNet (Oord et al., 2016) and the teacher part of ClariNet (Ping et al., 2019) fall in this category. During training, the entire input sequence is known and the model can use a sequence of transformations to transform the input into required output. This process does not necessarily require recurrent networks. Indeed, most models in this class use temporal convolutions that are easily parallelizable on modern hardware. Nevertheless, both mentioned models generate raw audio and relatively large number of model parameters is required to get reasonable results, which in turn requires multiple GPUs to train the model in reasonable time.

At inference, autoregressive flows require sequential generation, which is usually slower than with recurrent networks that are optimized for performance in sequential setting. To overcome slow inference times, Parallel WaveNet (Van Den Oord et al., 2018) and ClariNet (Ping et al., 2019) first train an autoregressive flow teacher network and then train an inverse autoregressive student network. The inverse autoregressive flow is easily parallelizable for output generation, but is slow during likelihood evaluation, which is required for training. The teacher network is able to evaluate the likelihood of a sequence quickly, but is slow during output sampling. The teacher can be used to calculate approximate likelihood of the student's output samples, which can be used for faster training of the student. This process is called teacher distillation. The student network is typically able to synthesize speech in real time. The autoregressive flow framework can be used to train real-time high-quality speech synthesis systems. Unfortunately, it is only applicable if our input and out has the same time dimensionality, which is usually not the case for phoneme-to-audio conversion. Thus, autoregressive flows are mostly used in raw audio domain in combination with previously extracted linguistic features. The former again means that the models require quite a considerable number of parameters to provide acceptable results, which may substantially increase training time on limited hardware.

Models producing raw audio based on linguistic features do not require teacher distillation. One such model is GAN-TTS (Bińkowski et al., 2019). This model samples a sequence of random numbers from some simple (typically Gaussian) distribution and upsamples the sequence into raw audio. The transformation is conditioned on the linguistic features. A set of adversarial and style losses is used to get a reasonably sounding speech. Again, modelling raw audio directly requires relatively large number of parameters for reasonable performance.

Finally, a few composite models are parallel at training and inference and do not require linguistic features. As already mentioned, FastSpeech (Ren et al., 2019) uses the Transformer architecture from (Vaswani et al., 2017) at training

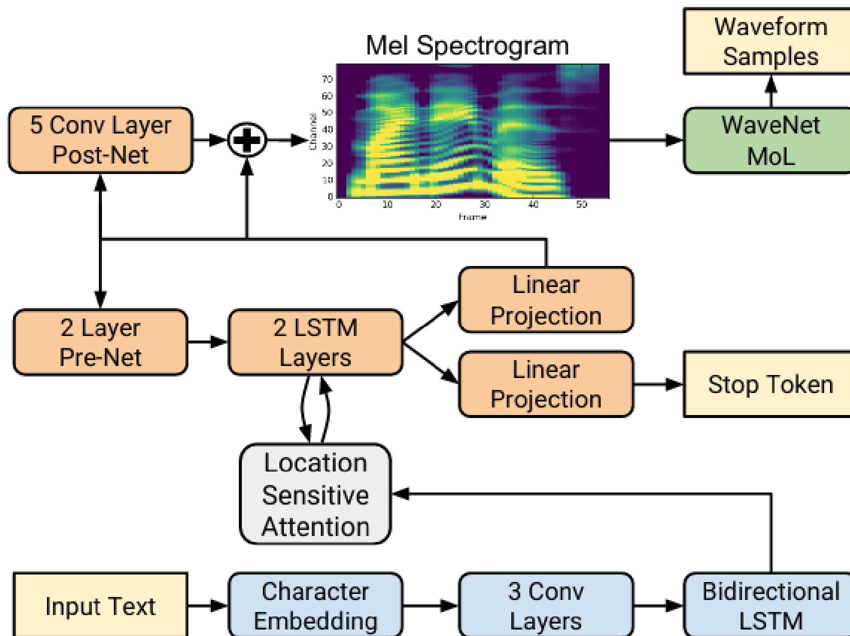

Figure 3.1: The full architecture of Tacotron 2. Source: Shen et al. (2018)

time to transform phonemes into mel-spectrogram frames using the attention mechanism for phoneme-spectrogram alignment. Then, the alignments are extracted and a new model learns to predict how many frames of a spectrogram correspond to a given phoneme. This can be seen as a form of teacher distillation, but unlike in flow-based models, the teacher is not used to evaluate output likelihood of the student, but to provide learned alignments.

3.3 Models related to our work

In this subsection, we describe architectures tightly related to our work in more detail: Tacotron 2, Deep Voice 3 and FastSpeech. We build on the described models and use some of the models for comparison in the evaluation of our system (see Section 6).

Tacotron 2 (Shen et al., 2018) is a sequence-to-sequence model with attention capable of producing high quality mel-scale and linear-scale spectrograms. Together with a WaveNet vocoder, the model is able to synthesize audio from text without requiring linguistic features. The architecture looks as follows (see Figure 3.1): The input characters are first normalized and encoded with stacks of convolutional layers. Then a a bidirectional LSTM is applied to further encode temporal dependencies among the input text. An autoregressive decoder network made of LSTM cells uses previously generated spectrogram frames concatenated with an attention vector from the encoder to generate a spetrogram for the next timestep. We use speech samples generated by Tacotron 2 as comparison samples in our speech quality evaluation (see Chapter 5). Please refer to (Shen et al., 2018) for further architecture details.

Deep Voice 3 Ping et al. (2018) is similar to Tacotron 2 in that it also uses an encoder for the input sequence and an autoregressive decoder with attention for spectrogram frame generation. However, the Deep Voice 3 model uses temporal convolutions interleaved with scaled-dot attention layers instead of recurrent neural networks in both the encoder and the decoder. The absence of the hidden state in the recurrent network cells allows to decode all spectrogram frames in parallel during training, which speeds up training time. Our own achitecture (see Chapter 4) is similar to Deep Voice 3 in that we also use temporal convolutions in the first part of our system. But unlike Deep Voice 3, we only use one attention layer to create alignment between the encoder and decoder and do not use attention layers between the convolutional layers in the rest of the network. We also use attention masking and attention preconditioning similar to Deep Voice 3. We use samples generated by Deep Voice 3 for comparison in our quality evaluation in Chapter 5.

Tachibana et al. (2018) present an architecture similar to Deep Voice 3. Unlike (Ping et al., 2018), this model uses only a single attention layer to align phonemes and spectrograms and does not include synthesis for multiple voices. Remaining layers are composed of temporally dilated convolutions. We use the Guided attention loss from Tachibana et al. (2018) to speed up convergence of the alignments in our model. For more information, see the original paper.

FastSpeech (Ren et al., 2019) (Figure 3.2) is a speech synthesis approach based on two models: The teacher network and the student network. Both networks are based on the Transformer architecture (Vaswani et al., 2017) (see also Section 2.5.2). The teacher learns the alignment between spectrogram frames and input characters. The alignments are extracted and used for training of the student model to predict input character durations. The student model consists of an encoder and a decoder, where both modules use stacks of attention layers. The student network encodes the inputs, predicts how many frames in the spectrogram correspond to each input token, copies the encoding of each token that many times and decodes the result into a spectrogram. We use a similar approach in our work (see Chapter 4), but instead of utilizing the Transformer architecture, we use temporal convolutional layers for our teacher network similar to Ping et al. (2018) or (Tachibana et al., 2018). Our student network does not use attention at all and instead uses a simple residual network blocks.

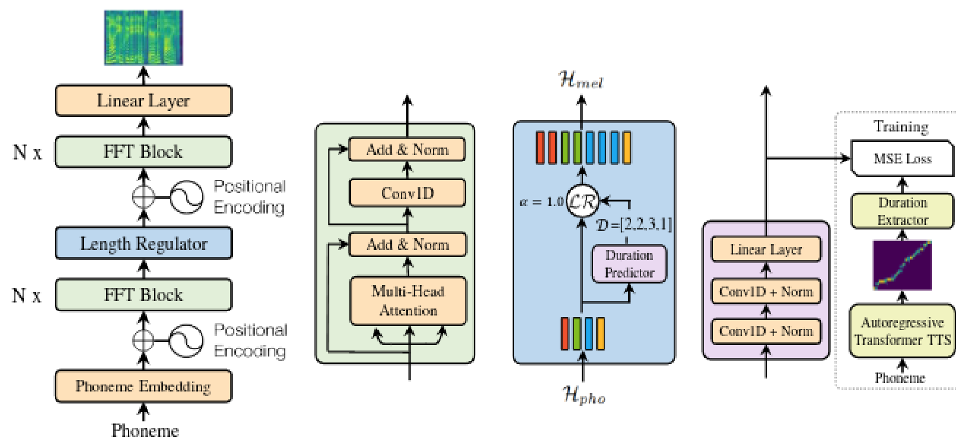

Figure 3.2: The components of the FastSpeech architecture. The full model (left) consists of an encoder, a length regulator and a decoder. The encoder and the decoder consist of stacks of FFT blocks (second from the left). The encoder produces contextualized phoneme embeddings. The length regulator (second from the right) expands the embeddings based on predicted phoneme durations. The decoder synthesizes a spectrogram from the expanded embeddings. The durations are predicted by a duration prediction module (right). Source: Ren et al. (2019)

4. Our TTS architecture

We first introduce our overall approach (see Section 4.1) and make a short comparison to corresponding literature listed in Chapter 3. Then, an in-depth description of all parts of our system is provided. We mainly focus on explaining the principles of our implementation. Architecture overview is provided in Section 4.2. The teacher and student models are described in sections 4.3 and 4.4 respectively. Implementation details such as the number of layers, kernel size and so on can be viewed in Appendix A.

4.1 Design principles

Our goal is to design a model capable of synthesyzing high quality speech, trainable in reasonable time on a single GPU and speedy during inference. We decided to operate on mel-scale spectrograms, which should simplify modelling of more distant relationships in the audio and improve speech qualities such as intonation compared to modelling raw waveforms. Moreover, the model has fewer parameters than if it operated in the raw audio domain.

To achieve high synthesis speed during inference and fast training, our model should be able to leverage parallelization at both training and inference time. However, most models satisfying this requirement operate in the raw audio domain with one notable exception–FastSpeech (Ren et al., 2019). FastSpeech is based on the Transformer architecture from (Vaswani et al., 2017) and consists of multiple multi-head attention layers. Unfortunately, Transformers can be quite difficult to train and require careful hyperparameter selection (Popel and Bojar, 2018). Therefore, we decided to use an approach similar to FastSpeech, but instead of the transformer attention blocks, we leverage dilated convolutional layers.

4.2 Architecture overview

Our system consists of two networks. The first network is a teacher network with an architecture based on (Ping et al., 2018) and (Tachibana et al., 2018) (see Section 3.2). The purpose of the teacher network is to learn a reliable temporal alignment between phonemes and spectrogram frames. We assume that the input text is already converted to phonemes. We leverage dilated temporal convolutional blocks with gated activations similar to WaveNet (Oord et al., 2016) along with a single attention layer that captures alignment between phonemes and spetrogram frames. The model is trained to predict spectrogram frames given frames at previous timesteps. The prediction is conditioned on the corresponding phonemes. The mathematical operations in each layer of the network are parallelizable and each frame can be predicted in parallel during training by conditioning on the ground truth spectrogram frames instead of the generated ones. After the model is trained, we use the alignment of each utterance in the dataset and extract phoneme durations. The durations are then used in the second model.

The second network in our system is a student network. The purpose of this network is to quickly generate a high quality spectrogram from a given sentence. The model first encodes the input phonemes. Then, a duration predictor similar to FastSpeech (Ren et al., 2019) predicts the duration for each phoneme based on the phoneme encodings. Next, each phoneme encoding is copied as many times as there were predicted frames. Finally, the encodings are decoded into spectrogram frames. The duration predictor is trained to fit the durations extracted with the teacher network. The model is not sequential and operations in each layer are parallelizable during both training and inference. Unlike FastSpeech, our network does not use any attention layers and utilises dilated convolutions, batch normalization and residual connections instead, which reduces training time (see Section 4.4).

For vocoding, we use a pretrained MelGAN vocoder (Kumar et al., 2019), which allows faster than real-time inference.

4.3 Duration extraction – Teacher network

To extract durations from the data, we build and train a model based on Deep Voice 3 (Ping et al., 2018) and (Tachibana et al., 2018).

4.3.1 Network structure

This model has four main parts – phoneme encoder, spectrogram encoder, attention and decoder. The full architecture is depicted in Figure 4.2.

Phoneme encoder: The phoneme encoder encodes a one-hot representation of phonemes into keys and values used in the attention layer (see Section 2.5.2). First of all, the phonemes are multiplied by an embedding matrix. Next, a fully connected layer and ReLU activation are applied. Then, several gated residual blocks with progressively more dilated temporal non-causal convolutions are used. An example gated residual block is displayed in Figure 4.1. The skip connection in the residual block accumulates transformations caused by the convolutional tanh and sigmoid gates. The skip connection from the final layer of residual blocks is used as the keys in the attention layer. The values for the attention layer are formed by summing the outputs of the phoneme embedding matrix with the keys, similar to (Ping et al., 2018).

The residual block is derived from the building block in WaveNet (Oord et al., 2016), but we do not use causal convolutions for the phoneme encoder to allow parallelization over time. Unlike (Tachibana et al., 2018), we do not use highway blocks (Srivastava et al., 2015) because the residual block is simpler and it is slightly more straightforward to implement a faster inference algorithm (more on that later in Section 4.3) for the gated residual block than for a full highway block without significant performance drop.

Spectrogram encoder: The spectrogram encoder is used to extract attention queries from spectrograms. First, a fully connected layer and ReLU are applied to each frame of the input spectrogram. Then, several gated residual blocks with

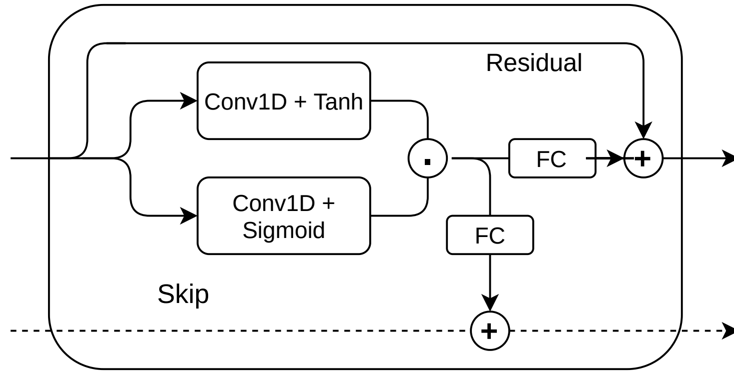

Figure 4.1: A gated residual block. The convolutions are progressively dilated to increase the receptive field. The blocks' skip connection sums outputs from all layers to form the network output. The "dot" symbol represents element-wise multiplication and the "plus" symbol represents element-wise addition. The "FC" block stands for a fully connected layer.

progressively more dilated gated temporal causal convolutions are used. Because the whole system is supposed to predict a spectrogram frame conditioned on frames in the previous timesteps, future spectrograms must be ignored during training and causal convolutions are used. Both phonemes and spectrograms are properly zero-padded to ensure that there are as many keys and queries as there are phonemes and spectrogram frames.

Attention: The keys and queries are preconditioned by adding the positional encoding matrix (see Sections 4.3.2 2.5.2) to bias the attention towards monotonicity. Next, an identical linear layer is applied to keys and queries. Then keys, queries and values are inserted into a scaled dot-attention module that outputs the attention scores. The attention scores are weighted averages of the value vectors according to how much the values match a given query. This way, the model can learn to select characters relevant for prediction of the next spectrogram frame. Because the goal of this model is to extract correct alignments among phonemes and spetrogram frames, the model should ideally attend to input text characters representing phonemes that are pronounced in the predicted spectrogram frame. The queries are added to the result in order to create a direct link between the spectrogram encoder and the decoder in the network computational graph. Similar to residual networks (see Section 2.5.4) the link allows better flow of the gradient during backpropagation through the attention layer from the decoder to the spectrogram encoder.

Decoder: Finally, the decoder decodes the resulting vectors into predicted spectrograms. The architecture of the decoder follows the same scheme like the spectrogram encoder except for the final block that consists of several convolutional layers with ReLU activation. The final block transforms the outputs into the correct spectrogram dimension by decreasing the number of the convolutional output channels. The schematic representation of the whole model can be seen in Figure 4.2.

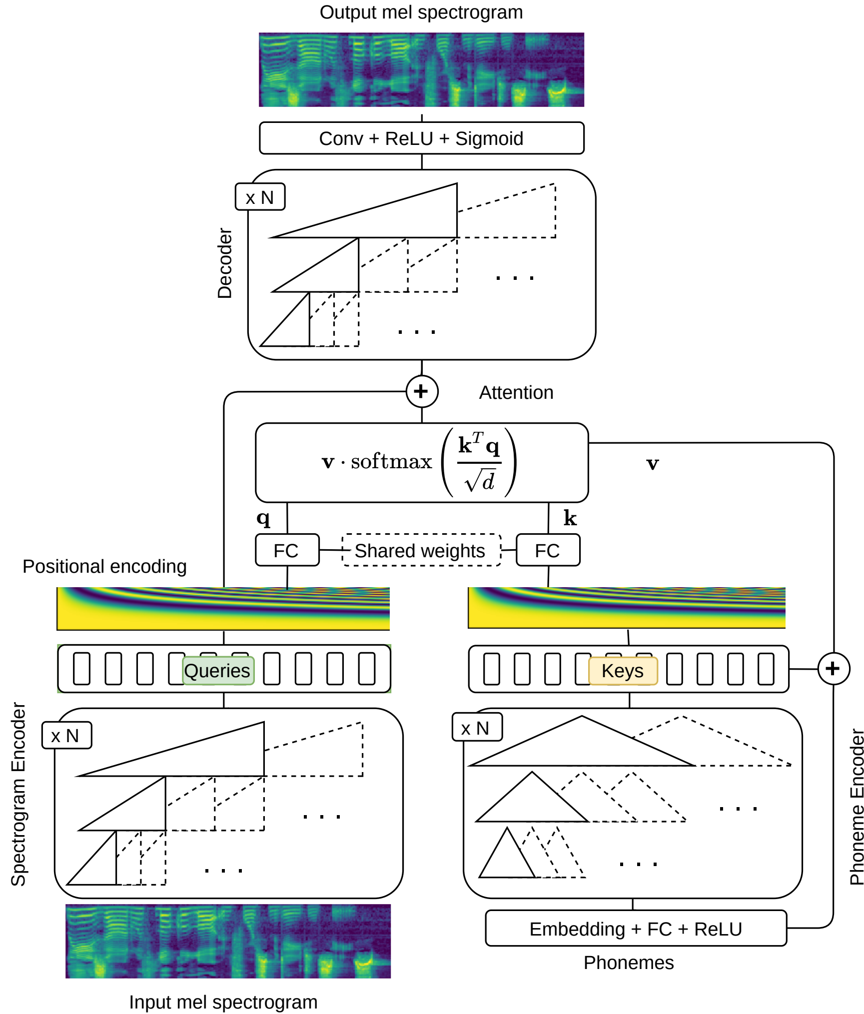

Figure 4.2: The network used for duration extraction from data. The positional encoding is element-wise added to the queries and a fully connected layer is applied on each column. The fully connected layers for keys and queries share weights. The gates inside the residual blocks are not visualized for better readability and dilated convolutions are displayed instead.

4.3.2 Attention preconditioning

The positional encodings (see Definition 9) have the following convenient property: First, we define positional vectors at k-th position as

$$\mathbf{p_k} = (pe_{(k,1)}, pe_{(k,2)}, \dots, pe_{(k,N)})$$

. We assume that N, the number of channels, is even. By taking the inner product of positional encoding vectors at different positions, we get the following result.

$$\mathbf{p_k} \cdot \mathbf{p_l} = \sum_{\substack{i \in (1,N), \\ \text{i odd}}} \cos\left(\frac{k}{c^{2i/N}}\right) \cos\left(\frac{l}{c^{2i/N}}\right) \tag{4.1}$$

$$+ \sum_{\substack{j \in (1,N), \\ \text{j even}}} \sin\left(\frac{k}{c^{2j/N}}\right) \sin\left(\frac{l}{c^{2j/N}}\right) \tag{4.2}$$

If $k = l$, we get $\mathbf{p_k} \cdot \mathbf{p_l} = N/2$ by the equality $\cos^2(x) + \sin^2(x) = 1$. It is not possible to get a higher value for any combination of k and l. A finite combination of periodic functions is again periodic. This means that if we define a function $f(|k-l|) = \mathbf{p_k} \cdot \mathbf{p_l}$, then f would be periodic with respect to the distance between positions. By combining many sine and cosine functions with different periods, it is possible to create a function with a very long period. Figure 4.3 contains comparison of period length for channel size 4, 8 and 80.

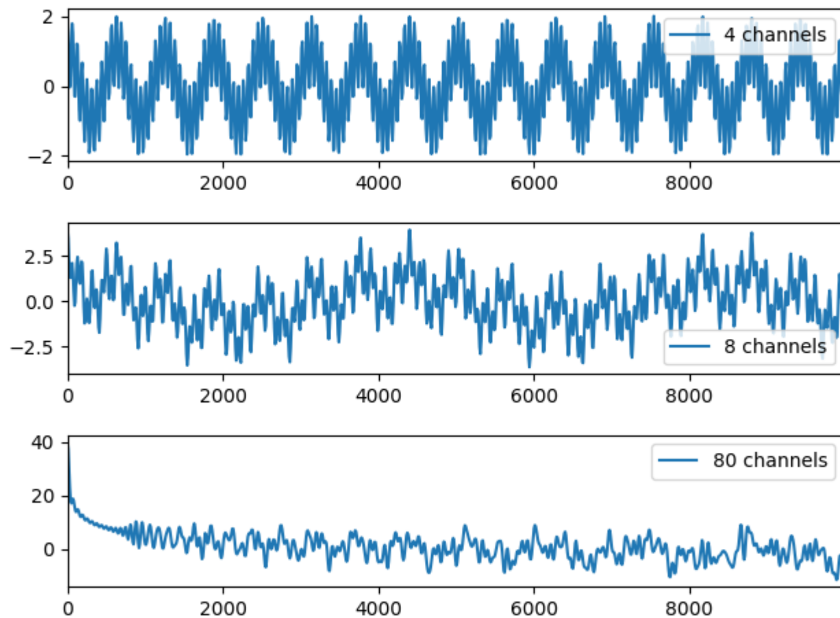

Figure 4.3: The value of $\mathbf{p_k} \cdot \mathbf{p_l}$ as k and l become more distant. The horizontal axis represents the distance $|k - l|$. The vertical axis represents the value of the dot product. The more channels are added to the positional encoding, the longer the period is.

As a consequence, if we use enough channels and we calculate the dot product of positional encoding vectors at various pairs of positions, we get the highest values for the same positions if the two input positions are identical. We can stack the encoding vectors into a matrix and calculate a matrix product with itself to extract $\mathbf{p_k} \cdot \mathbf{p_l}$ for all required k and l. The resulting encodings using 4, 8 and 80 channels are depicted in Figure 4.4. We can see that with a sufficient number of channels, we only get high dot product values around the diagonal.

Normalization of columns with softmax returns a multinomial distribution centered around the diagonal. We use the resulting matrix as an initial attention distribution between the keys and queries of the audio and text encoder in the attention module discussed in Section 4.3.1. It enforces approximately diagonal attention early on during training and if the learning rate is carefully selected, the duration extraction model converges faster. The detailed implementation is done as follows: Phonemes and spectrograms usually have different time dimensionality. There are typically more spectrogram frames than there are characters. Thus, there are more queries derived from spectrogram frames than keys derived from characters. To get an attention discribution centered around the diagonal, it is necessary to scale the positional encoding for queries. We extract the average

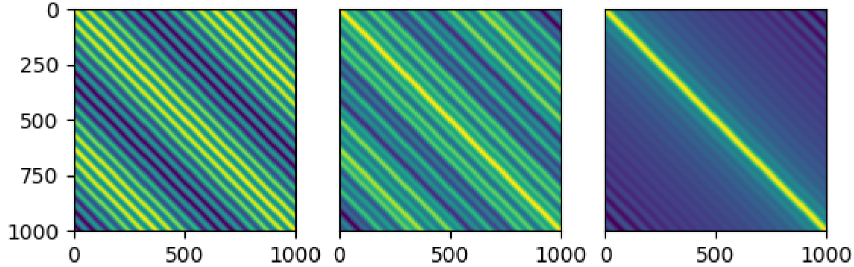

Figure 4.4: Dot product of positional encoding vectors for combinations of positions between 0 to 1000. A point (m, n) in each of the sub-figures represents the value of the dot product of $\mathbf{p}_m$ and $\mathbf{p}_n$. Positional encoding in the sub-figures has 4, 8 and 80 channels. As the number of channels increases, the periodicity also increases and for 80 channels, values outside the diagonal are relatively small compared to the diagonal.

ratio of character and spectrogram lengths in the dataset. There are on average approximately 7 spectrogram frames per character in the data used for our experiments (see Section 5.1). We simply scale the first subscript in the positional encoding for keys by 7 and add the corresponding encodings to keys and queries. Next, we feed each key and query through a fully connected layer. Both keys and queries share the same parameters of the fully connected layer to ensure that the diagonal distribution is not broken by incorrect weight initialization. For further illustrations and details see Ping et al. (2018).

4.3.3 Training

During training, target spectrograms are shifted one position to the left on the input and the model is forced to predict the next spectrogram frame based on input characters and previous frames. Because the network does not keep any hidden states, we can compute the predictions for each time step independently and in parallel.

We minimize the *mean absolute error* (see Definition 6) between the target and predicted spectrograms. Moreover, we also use guided attention loss described in (Tachibana et al., 2018) to aid monotonic alignments. The total loss is the sum of both losses. The guided attention loss is defined below:

Definition 10. *Guided attention loss for the attention matrix* $\mathbf{a} \in \mathbb{R}^{N \times T}$ *is calculated as:*

$$GuidedAtt(\mathbf{a}) = \frac{1}{NT} \sum_{n=1}^{N} \sum_{t=1}^{T} \mathbf{a}[n,t]\mathbf{w}[n,t], \tag{4.3}$$

where $\mathbf{w}[n,t] = 1 - \exp -\frac{(n/N - t/T)^2}{2g^2}$ *is the penalty matrix,* N *is the number of phonemes and* T *is the number of spectrogram frames.*

The variable g can be used to control the loss contribution of matrix elements $\mathbf{a}[n,t]$ as we move further away from the diagonal. The lower the value, the higher the penalty is around the diagonal and the tighter is the low-loss area around the diagonal. The penalty matrix $\mathbf{w}$ is visualized in Figure 4.5.

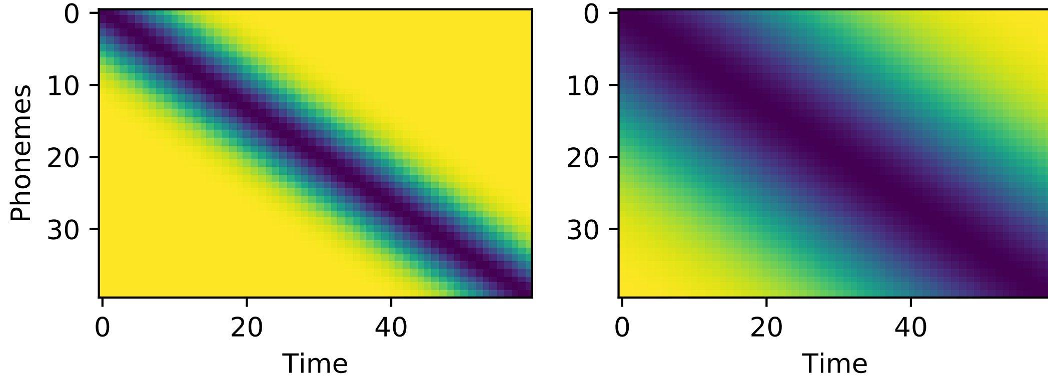

Figure 4.5: The penalty matrix **w** for parameter values $g = 0.1$ (left) and $g = 0.3$ (right). Dark colors signalize low penalty, while bright colors signalize high penalty. The penalty is almost zero near the diagonal.

Adam optimizer (Kingma and Ba, 2015) with default parameters and Noam learning schedule introduced in (Vaswani et al., 2017) is used for gradient descent learning. The learning rate schedule is such that the learning rate increases linearly for the first w "warmup" epochs. Then it starts to behave like an inverse square root decay. The variant of the learning rate used in this thesis is calculated as follows:

Definition 11. *The Noam learning rate with w warmup epochs at epoch i is $lr_i = base_lr \cdot c(i)$ where c is a scaling factor calculated as follows.*

$$c = \begin{cases} \frac{1-s}{w} \cdot i + s & \text{if } i \leq w \\ \sqrt{\frac{w}{i}} & \text{if } i > w \end{cases} \tag{4.4}$$

The constant s is the initial scaling factor.

The Noam learning rate starts at a value of $base_lr \cdot s$ and increases linearly until $w = i$, where the scaling factor c equals 1. Then, the learning rate decreases as an inverse square root. The learning rate attains its highest value in epoch w and the largest value is $base_lr$. This schedule is beneficial for training of the attention layer – the training is more stable and converges faster.

4.3.4 Duration Extraction

Inference needs to be done sequentially in the teacher network – the predicted spectrogram is generated frame by frame. We generate as many frames as there were frames in the corresponding ground-truth spectrogram. Predicted frames are reinserted into the spectrogram encoder input to allow the generation of further frames. To extract the phoneme durations, we first extract phoneme-spectrogram alignments with the following procedure: We run the sequential inference and with each generated spectrogram frame, we get an attention distribution over the input phonemes. For a given frame, we select the phoneme position with the maximum attention value. After generating the full spectrogram, we end up with a non-unique list of phonemes. If a phoneme is repeated in the list, it means that

the model has attended the phoneme in several consecutive timesteps. Assume that the alignment is such that the neighboring phonemes in the phoneme list are always at most one position in the sentence from each other and the model never attends to positions it has already attended except for the last one. Then, we could extract phoneme durations simply by computing number of occurences of each phoneme position in the phoneme list.

Attention masking: Unfortunately, the raw attention output is not completely clean – the model sometimes looks back or skips several phonemes. For this reason, we locally mask the attention scope at each timestep. For timestep t, we take the argmax of the attention distribution at timestep $t-1$ and mask all positions except the argmax position and several following positions with $-\infty$.

Then, we feed the masked scores to the attention softmax following (Ping et al., 2018). The softmax normalizes $-\infty$ to zero and assigns non-negative values to the positions that were not masked. This way, the model is forced to select the next phoneme close to the previous phoneme. If we only allow the model to select either the previous position, or one position ahead, the model only has to decide whether to keep reading the same phoneme or whether to start reading the next phoneme in the sentence. If we force the model to read the first phoneme in the sentence in the first timestep, the model can not skip any phoneme and we are able to extract the duration for each phoneme in the sentence by counting how many times its index appeared in the attended positions. However, it may happen that the model never arrives at the last phoneme. We solve this issue heuristically. If the gets stuck on a non-final phoneme, we check how many phonemes are left before the end of the sentence. If there are n phonemes at the end that were not attended, we simply set the last n extracted phoneme indices to the indices of the unread phonemes.

Using ground-truth spectrograms: We use one more trick to improve the alignment quality. Unfortunately, during sequential inference, the model tends to accumulate small errors in the spectrogram frames, which introduces undesirable duration errors. This is an instance of the *label bias problem* (Wiseman and Rush, 2016). Thus, instead of reinserting the generated spectrogram frames on the input, we use the ground-truth spectrogram frames instead. This technique is called *teacher forcing* (Pascanu et al., 2013). This technique prevents the model from error accumulation and the model is able to generate high-quality alignments even for long sentences. Please see Section 5.2 for more information about error accumulation, which includes a number of further steps we take to improve alignment quality.

Since we are using the ground truth spectrograms on the input during inference, we could theoretically run the network in train mode in parallel and gain a speed boost. However, we would not be able to apply the local attention masking because the argmax position of the attention at the previous timestep would not be known, since all the attention distributions are computed in parallel.

Speeding up inference: The sequential inference is quite slow. For a dataset with 20 hours of audio, it could take hours to extract the durations. To speed up the inference, we use an approach developed by Paine et al. (2016). During

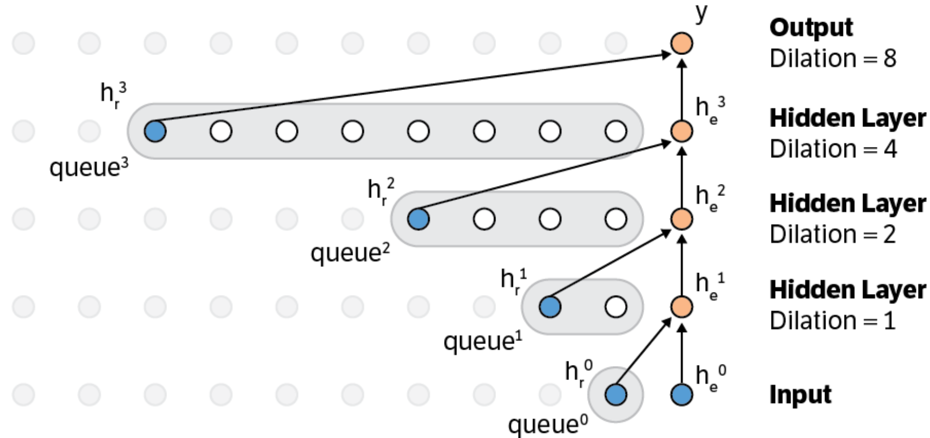

Figure 4.6: Results of intermediate computations are stored in layer-level queues. The items are progressively released at each timestep. Thanks to the cached values, it is not necessary to recalculate the operations in the subtree of the given node. Source: Paine et al. (2016).

inference, models relying on causal dilated convolutions repeat calculations that were already calculated in previous timesteps. To avoid unnecessary calculations, the results of the operations can be cached in queues and progressively released at each timestep. The length of a queue in a given layer depends on the dilation factor of the given layer. The process is visualised in Figure 4.6. The nodes in the computational graph represent mathematical operations. To compute the value of a node, it is necessary to compute the values of some of the nodes in the layer below it. The same holds for the nodes in the lower layer and so on. By using the precomputed value stored in a queue, the network avoids recalculating the nodes in the lower layers, which can save a lot of time, especially for deep networks. Please see (Paine et al., 2016) for more details and visualizations.

4.4 Spectrogram synthesis – Student network

Once durations for training data are extracted, we can train the speech synthesis model. The model does not directly model raw waveforms, but is able to generate mel-scale spectrograms that can be later transformed to raw waveform with a vocoder or iSTFT.

Our model is inspired by the FastSpeech architecture (Ren et al., 2019), but instead of attention blocks we use residual blocks of dilated convolutions. Instead of layer normalization, we employ temporal batch normalization. Unlike FastSpeech, we detach gradient flow from the rest of the network and observe increased performance of spectrogram prediction and reduced overfitting of the duration predictor. We also experiment with different positional encodings on the decoder input.

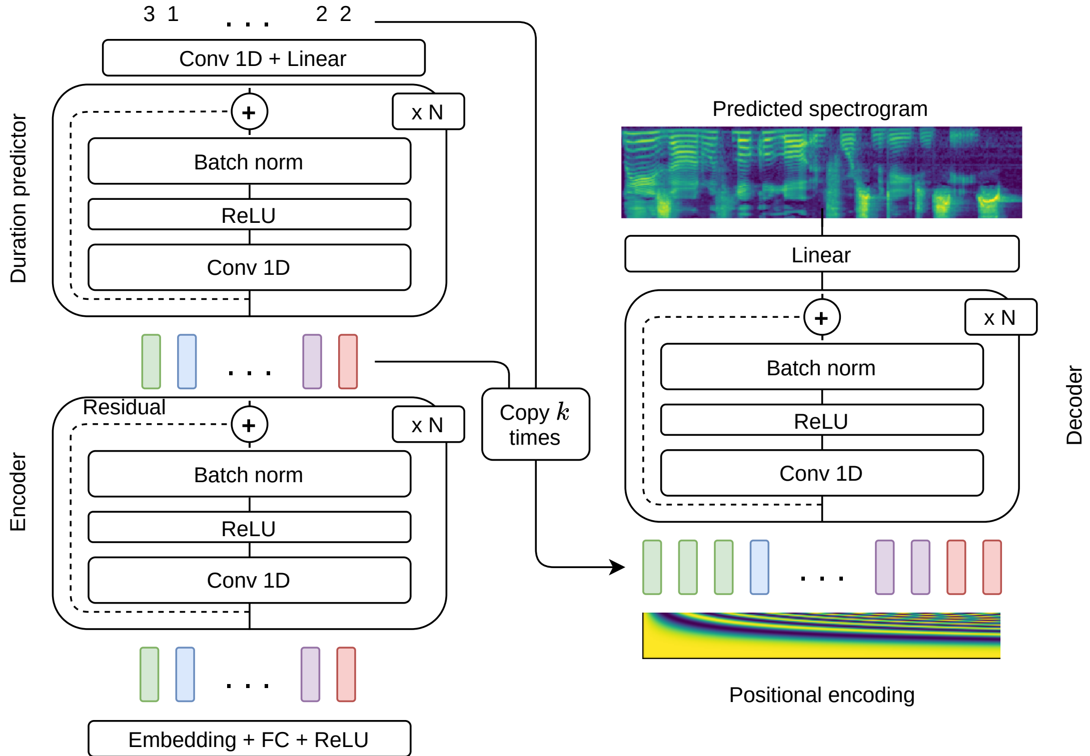

Figure 4.7: Our spectrogram synthesis model. The encoder takes one-hot encoded phonemes on input and outputs contextualized phoneme embeddings. The duration predictor accepts the embeddings and predicts the duration of each phoneme. Each phoneme embedding is copied according to the predicted durations. The decoder accepts the expanded phoneme embeddings summed with the positional encoding matrix and synthesizes a mel-scale spectrogram on output.

4.4.1 Network structure

Our spectrogram synthesis model is depicted in Figure 4.7. Similarly to the duration extraction network, the input phonemes are one-hot encoded and multiplied by an embedding matrix and a fully connected layer with a ReLU activation are applied (see Section 4.3). Then we stack several residual blocks of non-causal convolutions with progressively increasing dilation factors. Each residual block consists of a convolutional layer, an activation and a temporal batch normalization layer. A residual connection is applied for better gradient flow. Finally we add the initial phoneme embeddings to the result to form our final phoneme encodings. The encodings are fed to a duration prediction module. The purpose of this module is to predict how many spectrogram frames correspond to the given phoneme.

We predict the durations in logarithmic domain. Unlike Ren et al. (2019), we detach the gradient flowing from duration predictor to the rest of the network. Thus, the encoder part of the network can be fully utilized for spectrogram prediction. Without detaching the gradient, the duration predictor overfits the training durations even with strong dropout regularization (Srivastava et al., 2014). After detaching, we observe improved generalization in the duration predictor and lower spectrogram prediction loss (see our experiments in Section 5). Since the

detached duration predictor can no longer influence the encoder output, it is necessary to increase the duration predictor capacity for good performance. We again utilise residual blocks of dilated convolutions. Our duration prediction module consists of 3 residual blocks and one convolutional layer that transforms the channel dimension to 1. The final number is the (logarithmic) prediction; applying the exponential function and rounding then yields the number of frames.

Once each phoneme duration is predicted, each phoneme encoding is copied as many times as there are predicted frames for the given phoneme. All the copies are concatenated and form the input for the decoder. Similarly to FastSpeech, we also add positional encodings (Vaswani et al., 2017, see Section 2.5.3) on the decoder input, but instead of generating the encoding for full sequence length, we reset the encoding for each phoneme.

The decoder consists of residual blocks of dilated non-causal convolutions. A fully connected layer is applied at the end to convert the output dimensionality to that of the mel spectrograms.

4.4.2 Positional encoding for student model

We also use positional encoding (see Section 2.5.3) in the spectrogram prediction model. The motivation for this is the following: Because the encoder outputs in the spectrogram prediction model are expanded based on durations, there are homogeneous segments of vector copies. For large enough segments, the first convolutional layer produces identical vectors in the middle of the segment and produces some interpolation of neighboring segments on the segment borders. With more convolutional layers, the segment contents are mixed. To mix contents of neighboring segments faster, it could be beneficial to apply some deterministic interpolation between neighboring segments prior to feeding the vectors to decoder. Likewise, it could be beneficial to apply some positionally unique transformation of each vector to allow the network to distinguish where the originally homogeneous segments start and end.

The standard positional encoding breaks the homogeneous segments and also encodes information about the global position to the encoder outputs. However, we hypothesize that it is more beneficial for the network to distinguish the frame location in context of a single phoneme instead in the global sentence context. Similarly to FastSpeech Ren et al. (2019), we use the positional encoding from (Vaswani et al., 2017) but we reset the encoding for each homogeneous segment. In other words, we create a new positional encoding for each phoneme. After expanding the encoder outputs, positional encoding frames are added element-wise to the outputs. The mapping is visualised in Figure 4.8. Please see Section 5 for comparison of our approach and the approach in FastSpeech (Ren et al., 2019).

4.4.3 Training

During training, we use a combination of losses. We use a sum of *mean absolute error* (see Definition 6) and *structural similarity index* (SSIM) for logarithmic mel spectrogram value prediction (regression) and *Huber loss* for logarithmic duration prediction.

The SSIM loss compares structural similarity of spectrogram patches based

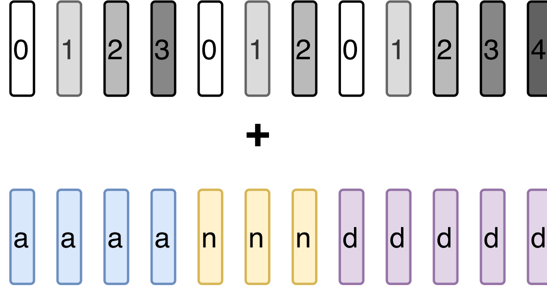

Figure 4.8: Positional encoding frames are mapped to each expanded character/phoneme separately. The grey tiles represent positional encoding frames. The colored tiles represent expanded encoder outputs for the word "and".

on luminance, contrast and structure (Wang et al., 2004). The formula for computation is as follows.

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \tag{4.5}$$

The symbols $\mu_x, \sigma_x, \sigma_{xy}$ represent pixel value average, variance and covariance of a given image (i.e. the spectrogram in our case). c_1 and c_2 are stabilizing constants computed based dynamic range of the pixel values.

Definition 12. *Huber loss for a data pair* $\mathbf{x} \in \mathbb{R}^{n\times k}$, *and* $\mathbf{y} \in \mathbb{R}^{n\times k}$ *can be defined as:*

$$Huber(\mathbf{x}, \mathbf{y}) = \frac{1}{nk}\sum_{i=1}^{n}\sum_{j=1}^{k} z(\mathbf{x}[i,j], \mathbf{y}[i,j]), \tag{4.6}$$

where:

$$z(x,y) = \begin{cases} 0.5(x-y)^2 & |x-y| \leq 1 \\ |x-y| - 0.5 & else. \end{cases} \tag{4.7}$$

Huber loss (Hastie et al., 2009, p.349) behaves like mean square error (MSE) inside the $(-1, 1)$ interval and behaves like mean absolute error outside the $(-1, 1)$ interval. The MAE error can be more robust to outliers, since its gradient is linear with respect to error size. However, this holds even for small error values, where the gradient may be too large to further decrease the error. On the other hand, MSE scales gradient quadratically with error size, which may improve convergence for small error rates. However, MSE is not robust to outliers. Huber loss combines MSE and MAE to get the best of both losses.

We again use Adam optimizer (Kingma and Ba, 2015) for training. Instead of Noam learning rate used in FastSpeech, we use the ReduceOnPlateau learning rate. Since we do not use any attention layers in the speech synthesis model, the warmup at the beginning of training is not necessary. We set the learning rate relatively high at the beginning and decrease the learning rate by a fixed proportion if the validation loss does not improve for a fixed number of epochs. During training, ground truth durations are used for expansion of the phoneme encodings. The durations predicted by the duration prediction module are used only during inference.

4.5 Vocoding

To generate raw audio, the generated spectrograms must be transformed into waveforms. We experiment with two approaches – the Griffin-Lim algorithm already discussed in Section 2.3 and a pretrained MelGAN model (Kumar et al., 2019). Unfortunately, the spectrogram transformation with Griffin-Lim is not fast enough for real-time processing. Most of the time is spent on approximating the linear magnitude spectrogram from mel-scale spectrogram by non-negative least squares.

Unlike Griffin-Lim, which is a rule-based algorithm, MelGAN is a neural-network-based, trained model. MelGAN is a non-autoregressive, fully convolutional model. As the name suggests, MelGAN is trained in an adversarial setting (see Goodfellow et al. (2014) for more information). The input spectrograms have a 256 times reduced temporal resolution compared to the raw waveform. The model uses sequences of deconvolutions followed by residual blocks with dilated temporal non-causal convolutions. The spectrogram frames get upsampled with the transposed convolutions. The model uses weight normalization (Salimans and Kingma, 2016) in all generator layers. Three discriminators are used as a source of loss in the adversarial training setup. The generated waveform is downsampled by different factors with strided average pooling and each discriminator operates on the corresponding scale. According to the authors, this allows the discriminators to learn discriminative features of different frequencies.

The model generates high quality audio and is faster and more lightweight than the standard WaveGlow model (Prenger et al., 2019).

5. Experiments

In this section, we describe our experimental setup. The dataset and data pre-processing is described in Section 5.1. The duration extraction and spectrogram synthesis models are discussed in Sections 5.2 and 5.3 respectively. We focus especially on problems we encountered during training and implementation of our models and provide solutions and comments to some of them.

5.1 Data set and data pre-processing

We trained all our models on the publicly available LJ Speech data set (Ito, 2017). The data set consists of 13,100 recordings and corresponding transcripts of a single professional female speaker reading from several texts in English. Numbers, ordinals, and monetary units in the transcription are expanded into full words. Theb bit depth of the audio files is 16-bits, the sample rate is 22,050 Hz.

We phonemize each transcript with the g2p python package[1], and use phonemic transcription as the input to both our teacher and student network (see Section 2.1). The g2p package searches the *CMU pronunciation dictionary*[2] for the word pronunciation. If the word is out of vocabulary, the pronunciation is estimated with a neural network. We transform amplitude spectrograms to mel-scale and a log transformation is applied on the amplitudes. A log-magnitude mel spectrogram is displayed in Figure 2.3. The final spectrograms used as prediction target for our models can be described as follows:

$$final(STFT) = \left(\log \circ \text{mel} \circ \sqrt{(Re^2 + Im^2)}\right)(STFT) \tag{5.1}$$

We reserve the last 100 utterances for evaluation, the rest is used for training.

5.2 Duration extraction

We used the procedure described in Section 4.3 to train the duration extraction model. In our preliminary experiments, we encountered multiple problems and attempted to address them, which we now describe in detail.

5.2.1 Training stability

The main problem we encountered is the stability of training. The attention layer is relatively hard to train and is sensitive to small changes on the input. If the learning rate is too high, the model diverges. We also experienced numerical instability errors due to gradient explosions.

Gradient clipping solves the large gradient errors, but is not enough to stabilize learning. Simple learning rate schedules such as inverse square root decay (Wu et al., 2019) work, but require a low initial value to compensate for the large weight

[1] `https://github.com/Kyubyong/g2p`
[2] `http://www.speech.cs.cmu.edu/cgi-bin/cmudict`

changes required by the attention layer. Further decreasing the learning rate results in small gradient steps later during training and causes longer convergence times. The inverse square root decay with warmup (Noam scheduler) used in (Vaswani et al., 2017) worked better for us. We experimented with warming up for 10, 20, 30 and 40 epochs and saw fastest convergence with warming up for 30 epochs.

5.2.2 Alignment learning

The attention layer is supposed to learn alignments between phonemes and spectrogram frames. The alignment should be more or less monotonic. We experimented with the guided attention loss introduced by Tachibana et al. (2018) to encourage the network to use monotonic alignments early on. Without the guided attention loss, it took around 100 epochs for the network to start displaying monotonic alignments. With the loss, the model started using alignment near the diagonal in around 50 epochs. However, the attention preconditioning with positional encoding (Ping et al., 2018, see Section 4) helped the most. If combined with the Noam scheduler, the alignment is monotonic from the very beginning of the training and stays monotonic. Nevertheless, if the learning is too large at the beginning, the monotonicity breaks. Even with the attention preconditioning, the model sometimes skips the rest of the sentence at a certain step and attends to the final character such as full stop, or question mark. In an attempt to prevent this behaviour, we use both, the guided attention and attention preconditioning and achieve relatively stable and reliable alignments.

Masking the attention scores in padded regions also helped attention training. Since inputs of uneven length in the same training batch are padded, the attention layer has an option to attend to the padded region, which is undesirable because it allows the model to make easily avoidable mistakes. To prevent such mistakes, we masked the attention scores corresponding to the padding characters with $-\infty$ before feeding the values to softmax.

Because the learned alignments are used to extract phoneme durations, it is desirable that we have sparse alignments, where most of the weight at a given timestep is put on a single phoneme. To encourage sparsity, we tried injecting Gaussian noise to the attention inputs during training, which should make the attention more robust to small input changes and thus improve sparsity. However, we did not observe any significant sparsity improvements.

We also tried to improve the attention robustness by applying dropout to the network weights, but even a very low probability of dropout caused worse alignments and higher guided attention loss, which is illustrated in Figure 5.1.

We also explored some normalization options in the hope that it would further stabilize the attention mechanism. Unfortunately, 1D batch normalization can not be directly used. It could be used during parallel training, because all time steps are known. But it could not be calculated during sequential inference, because it is necessary to average accross timesteps that have not been generated yet. A straightforward option would be to use layer normalization like Vaswani et al. (2017). Unfortunately, layer normalization improved neither stability nor gradient flow. We even observed a performance drop and a training slowdown. Finally, we decided to avoid normalization altogether, similarly to Tachibana

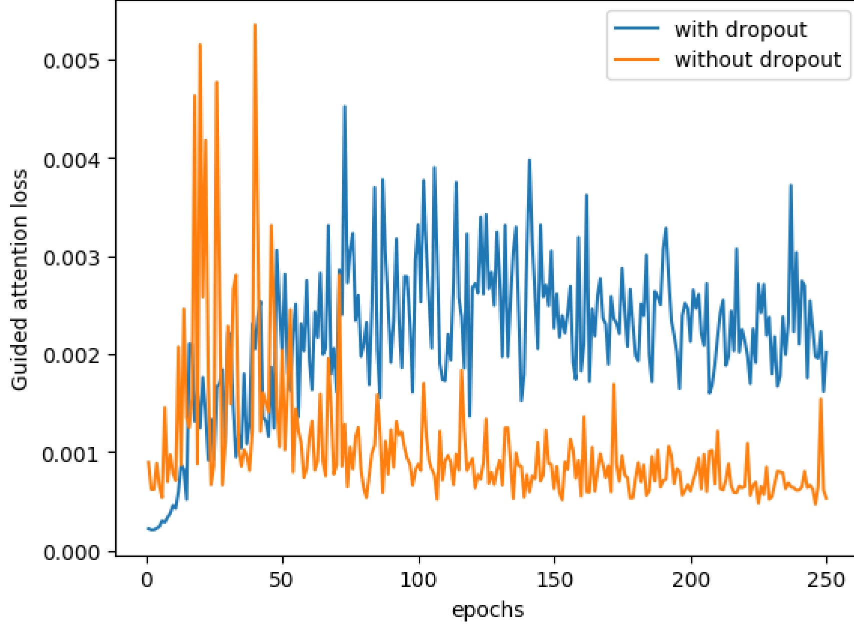

Figure 5.1: Guided attention loss before and after applying dropout with probability 0.1. The dropout did not improve alignment robustness and caused worse performance.

et al. (2018) and Ping et al. (2018).

5.2.3 Error propagation in sequential inference

Another problem that we encountered was the error propagation during sequential inference (see Section 4.3). During training, the model sees only the ground truth spectrograms on the input and tends to overfit. During sequential frame generation, the generated frames are used as the input to the model in the next step. Small errors produced by the model accumulate over time and are able to completely disrupt the attention alignments, which results in the model producing either complete silence too early, or random stuttering.

While not helpful for attention sparsity, injecting random noise to the attention inputs does help with attention robustness to accumulating error to a certain extent. However, further regularization was needed to stabilize the sequential inference.

Surprisingly, scaling the spectrograms to the $(0, 1)$ interval helped a lot. A modified model with linear output activation did not converge at all, and scaling the spectrograms to $(-1, 1)$ interval did not help as much, even though it would allow us to use hyperbolic tangent activation function with more viable gradient values around zero and approximately zero centered inputs. We presume this could be caused by the absolute size of the propagated errors. The network error in the zero-one scaling is mostly smaller in absolute terms, than in the other two cases, which could be beneficial for the alignment stability.

To further improve robustness to error propagation, we employ various data augmentations on the input spectrograms. First, we add a small amount of Gaussian noise to each spectrogram pixel. Next, we attempt to simulate the model

outputs by feeding the input spectrogram through the network without gradient update in parallel mode (not sequentially). The resulting spectrogram is slightly degraded compared to the ground-truth spectrogram. We take this first-order degraded spectrogram and repeat the process. The more times the spectrogram is degraded in this way, the more it should resemble the sequentially generated spectrogram. We could simply generate the degraded spectrogram sequentially, but using the parallel mode several times is still faster than sequential generation. Moreover, in the first 100 epochs the model is virtually unable to sequentially generate more than just a few frames correctly. On the contrary, the parallel approach provides spectrograms that are consistent across time. We observe that this method improves the robustness of sequential generation drastically and the model is able to generate long sentences relatively well with just minor mistakes.

Finally, we further degrade the input spectrograms by randomly replacing several frames with random frames. This is done to encourage the model to use temporally more distant frames. Otherwise, the model tends to overfit to the newest frame on the input and ignores older information, which makes the model less stable.

5.2.4 Inference speed

The sequential inference is quite slow by default. It requires as many forward passes through the network as there are frames to generate. This can be quite problematic for duration extraction, because the model needs to generate alignments for all dataset items. In LJ Speech, there are 13,100 sentences. If we run inference with batch size 64 and each batch takes around one minute to finish, it would take around 3.5 hours to extract all the durations. To speed up the inference, we implemented the dynamic-programming-based inference algorithm introduced by Paine et al. (2016) (see Section 4.3). We apply the algorithm to both the spectrogram encoder and decoder since both of them use temporal causal convolutions. We are able to generate 1,000 frames in a matter of seconds, compared to the naive implementation that can take over a minute.

5.2.5 Location masking in duration extraction

As already mentioned in Section 4.3, we extract the phoneme durations by running sequential inference with ground-truth spectrograms on the input to avoid error propagation. For this reason, it is not necessary to include stop indicator characters on the input such as in (Shen et al., 2018) because we know how long the spectrograms are supposed to be. The alignments generated in this way are quite reliable and monotonic, but are not very sparse and it can happen that the model attends to phonemes it has already seen, which is undesirable. For this reason, we use a location-based masking similar to (Ping et al., 2018).

We experimented with two mask types – argmedian and argmax masks. The argmedian mask finds the index (position) of the median of the attention distribution in the previous time step and masks all positions in the current time step except for a small window around the last argmedian position. Similarly, the argmax mask finds the position of the maximum of the attention distribution in the previous time step and masks all positions except a small window around

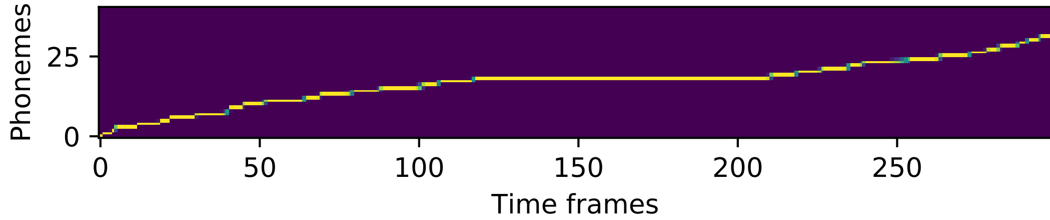

Figure 5.2: Alignment generated by the duration extraction model. The y axis captures phonemes, while the x axis captures spectrogram frames. The window size was set to 2. The generated sentence is *"The quick brown fox jumped over a lazy dog."*

the argmax index. The window can either be centered around the given position or be located before or after the index in a temporal sense. For our use case, it was necessary to locate the window after the index, but include the index as well. Thus, once the model decides to stop attending to a phoneme, it is not possible to return to the same phoneme again. Because all positions outside the window are masked, argmax or argmedian positions are always inside the window. For correct alignment at the first time step, it was necessary to manually set the window to the first one or two phonemes.

The attention distribution inside the window typically had a large weight put on the phoneme attended in the previous time step and some weight on the next one or two phonemes. The weight for the rest of the phonemes usually decreased exponentially with the distance from the phoneme attended in the previous time step. Based on this distribution, The argmax usually selected the previous phoneme or the phoneme right after, while the argmedian took the entire distribution into account and would often select some of the phonemes with lower weight, which resulted in phoneme skipping. We decided to use the argmax masking with a window size of 2. This approach effectively prevented phoneme skipping. An alignment by the duration extraction model is visualized in Figure 5.2.

5.2.6 Model validation

Finally, we would like to discuss model evaluation during training. To have some measure of how the model performs during sequential inference, it is necessary to compare the ground-truth spectrograms from the development set with the generated spectrograms. However, the spectrograms may have different lengths. We tried to measure the performance by running the sequential inference with ground-truth spectrograms on the input. However, this approach did not provide enough information about the quality of the model alignments. The model was able to generate high quality spectrograms regardless of the alignment because the information in the input frame was enough to predict the next time step and there was no error propagation (teacher forcing is in use, see Section 4.3). In fact, we observed that unless the teacher model is able to generate relatively high quality sentences even without the use of ground-truth spectrograms on the input, the student model's performance suffers because the extracted durations are wrong.

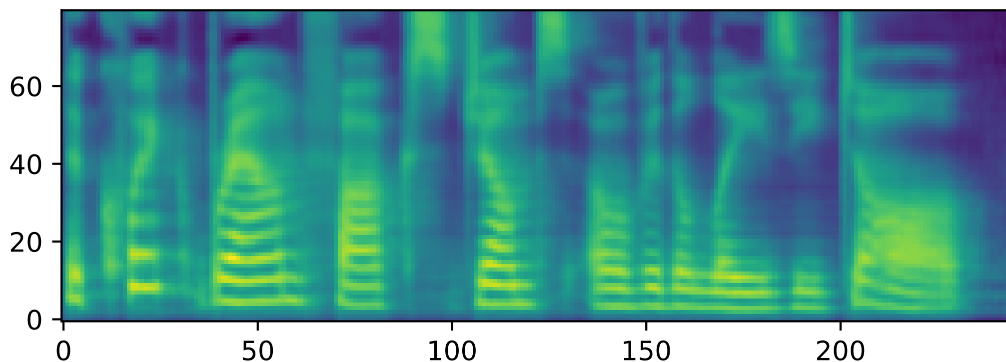

Figure 5.3: A log scale mel spectrogram of the sentence *"The quick brown fox jumped over the lazy dog."* generated by our spectrogram generation model.

Thus, we generated 3 batches fully sequentially every 10 epochs for model validation. Comparison with the ground-truth spectrograms was done using the fast dynamic time warping algorithm (DTW) (Salvador and Chan, 2004). As a metric, we used the L1 loss that was also used as a loss function during training. The results were quite noisy but it was possible to see a decreasing trend in the DTW loss. We also found it helpful to generate audio from a few generated spectrograms and manually check if all the input words were correctly pronounced.

5.3 Spectrogram synthesis

We now move to the model for spectrogram generation (or student network; see Section 4.4 for architecture details). A log-scale mel spectrogram generated by our model is depicted in Figure 5.3. Same as in Section 5.2, we now describe problems encountered in preliminary experiments and our way of addressing them.

5.3.1 Input phoneme durations accuracy

First of all, we would like to say that this model does not work very well without somewhat accurate phoneme durations. We tried to extract the durations from the duration extraction model that was not fully trained, and we were not able to fit the student model properly. Thus, it is crucial to have a reliable extraction model. Once accurate durations are obtained, it is possible to train the model.

5.3.2 Batch normalization and dropout

In initial experiments, we observed relatively slow learning. We ascribed this to small gradient sizes in the initial layers of the network. To overcome the vanishing gradient problem, we tried to employ different normalization techniques. Batch normalization (Ioffe and Szegedy, 2015) applies per-channel normalization across all time steps and across all items in a batch. Unlike in the duration extraction model, it is feasible to use 1D batch normalization for spectrogram synthesis, because all the time steps of the inputs are known. We experimented with layer normalization (Ba et al., 2016) too, but the performance was much better with

batch normalization. The batch normalization decreased the necessary training time.

We also tried applying channel normalization without considering the batch dimension, but this type of normalization provided slightly worse results and did not provide significant stability gains during inference.

We experimented with dropout applied after batch normalization and ReLU in all layers to encourage generalization. However, this approach reduced the model capacity which was already fully used (more on that later), and also made the network very unstable during inference – the duration predictor would predict shorter durations for most of the phonemes which in turn negatively influenced the synthesized spectrograms. We suspect that dropout causes inconsistency in the variance of the intermediate network states. However, more experiments would be needed to confirm this hypothesis. More on the disharmony of batch normalization and dropout can be found in (Li et al., 2018).

5.3.3 Learning phoneme durations versus spectrograms

We also encountered problems due to the need to jointly optimize quality of generated spectrograms and phoneme durations. The durations are always non-negative and can attain relatively large integer values – There can easily be 20–30 spectrogram frames corresponding to one phoneme. We experimented with L1 and Huber loss for durations. Both losses attain larger values than the L1 loss applied to generated spectrograms due to the different scales of both quantities. We observed that the duration predictor submodule has the largest influence on the gradient in the encoder. The gradient from the duration predictor is much larger than the gradient from the spectrogram decoder and has larger impact on the encoder weights. This degrades the quality of the generated spectrograms because the network uses most of the encoder capacity for the prediction of phoneme durations. The duration predictor has more capacity than needed and easily overfits to the training durations, which results in low quality duration predictions on the test data and makes the model largely unstable during inference. To compensate for this issue, we detach the gradient flowing from the duration predictor. In other words, the gradient from the duration predictor is not used to update the encoder weights. This way, the model capacity can be used to predict the spectrograms and the duration predictor needs to adapt to the encoder output. This change made it necessary to slightly increase the capacity of the duration predictor, so we added 3 residual blocks to the predictor to compensate for the capacity loss. The resulting spectrogram quality is better, the duration predictor generalizes better and both training and inference are stable. The effect of detaching the gradient is displayed in Figure 5.4. It is also possible to fit the duration predictor with a different optimizer and learning rate scheduler without any effect on the rest of the network. The predictor could be even trained separately after the rest of the network is trained.

5.3.4 Using logarithmic phoneme durations

The extracted durations are not perfect. In addition, there seem to be some annotation errors in the LJ Speech dataset. As a consequence, a small number

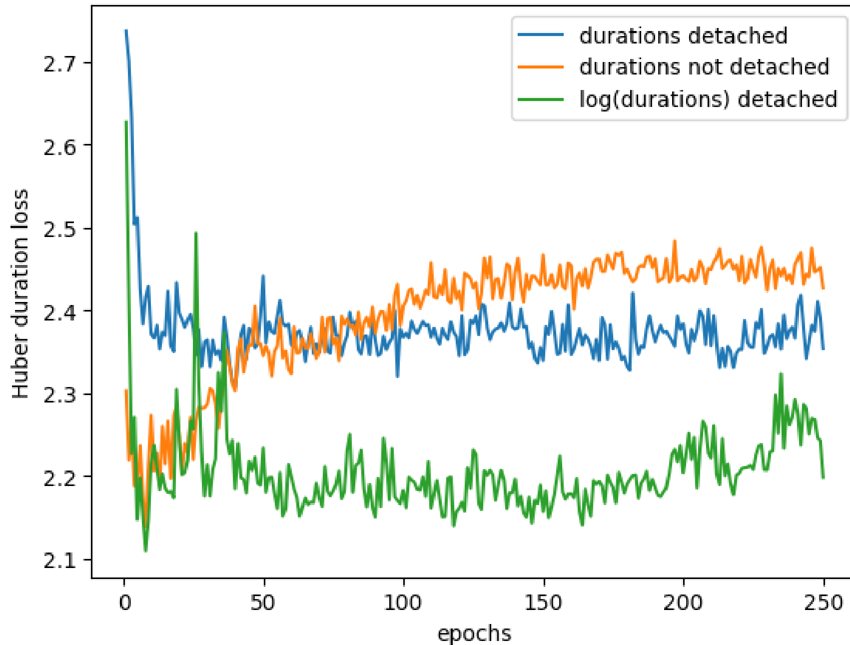

Figure 5.4: Huber loss of predicted phoneme durations. The durations from the model that was trained with detached gradients of the duration predictor and was trained to predict logarithmic-scale durations attains the lowest loss. Before calculating the loss, the exponential of the logarithmic durations was taken so that the losses of different configurations could be compared. The configuration of the duration prediction module was the same in all configurations.

of the phoneme durations are very large compared to the rest. To eliminate the effect of such outliers, we used the Huber loss discussed in Chapter 4. We also set the duration predictor to predict logarithmically scaled durations instead of linear durations, following (Ren et al., 2019). We saw better duration prediction quality mainly due to the logarithmic scaling, as can be seen in Figure 5.4. The logarithmic durations seem to follow a symmetric mean-centered distribution more closely, which plays well with the L1 loss function – L1 loss expects the data to come from the Laplace distribution which is mean-centered and symmetric around the mean (see Definition 6). It could be interesting to apply further transformations on the durations for the distribution to more closely resemble the Laplace distribution. We expected the Huber loss to give slightly more accurate duration predictions compared to L1 loss because the loss decreases quadratically around zero, which allows more precise gradient update. However, we saw no significant improvement.

5.3.5 Learning rates

Detaching the gradient and the use of batch normalization allowed us to use large learning rates early in the training. Otherwise, we experienced exploding gradient problems and a smaller learning rate had to be used. The learning rates produced by reduce-on-plateau and Noam schedulers (see Section 4.3) are displayed in Figure 5.5. The corresponding loss values are visualized in Figure 5.6.

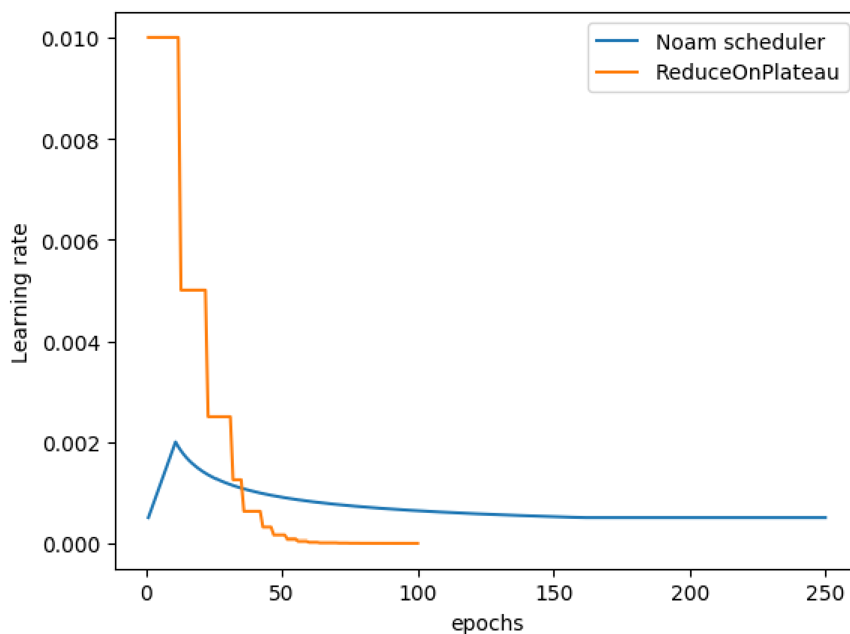

Figure 5.5: Reduce-on-plateau and Noam learning rates (see Figure 5.6 for model training performance).

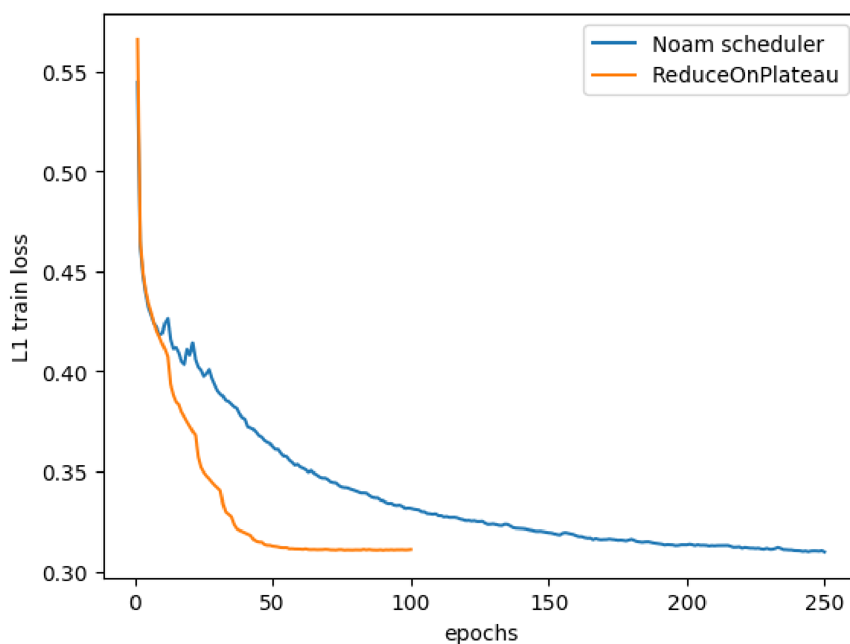

Figure 5.6: L1 spectrogram loss for Noam and reduce-on-plateau schedulers (see Figure 5.5). The model trained with Noam scheduler required around 250 epochs to converge to the final loss value reached in under 100 epochs by the model trained with the reduce-on-plateau scheduler.

5.3.6 Normalizing spectrograms

After we normalized the spectrograms to have zero mean and unit variance, we observed higher-quality spectrograms. This standardization requires computing the train set statistics.

5.3.7 SSIM

The L1 loss for spectrograms can be combined with SSIM loss mentioned in Section 4.4. The model works without SSIM (with L1 loss only) but with SSIM, the learned spectrograms appear less blurry. However, the difference in perceived audio quality is mostly unintelligible due to noise introduced by imperfect vocoding. More experiments with a high quality vocoder are needed to asses the impact of SSIM, but we expect the differences to be rather cosmetic.

5.3.8 Padding and masking

We mask all losses in length-padded regions so that the model does not need to learn additional mapping from padded inputs to zeros on the output. During inference, we observed that the model had pronunciation problems at the end of the longest sentence in each batch. The model sometimes cut the end of the last word or tried to pronounce the word unnecessarily quickly. We suspect that the model is used to the input zero padding seen during training in all examples except the longest one in the batch and uses the zeros around the border to pronounce the last word in the sentence correctly. We solve this by padding the batch with more zeros so that even the longest sentence is padded. This largely solved the pronunciation issue.

5.3.9 Homogeneus spectrogram segments

The phoneme encodings generated by the encoder are expanded (copied multiple times) before being fed into the encoder to compensate for the length mismatch of the input phonemes and the output spectrogram frames (see Section 4.4). The problem is that the expansion creates segments of identical vectors, one segment per phoneme. Applying a convolution on a sequence consisting of homogeneous segments can then result in another sequence of largely homogeneous segments. We observed that using a deeper architecture with dilated convolutions of sufficient dilation factor largely eliminated this issue, and the decoder was able to generate unique frames for each timestep.

5.3.10 Positional encodings

We experimented with adding positional encoding on decoder input, similarly to Ren et al. (2019). We compared three configurations:

1. Standard positional encoding (global position with respect to the whole audio segment)(Ren et al., 2019; Vaswani et al., 2017)

2. Local position – Reset the positional encoding for each phoneme (our new approach, see 4.4.2)

3. No positional encoding

The configurations with both forms of positional encoding were able to reach slightly lower L_1 and SSIM losses. The standard positional encoding attained the lowest loss but overall, the differences among all the configurations were small and there were no audible differences in the synthesized waveform. We conclude that positional encoding is not an essential part of our architecture and could be omitted. The resulting L_1 losses for all configurations are displayed in Table 5.1.

Standard pos. encoding	Our pos. encoding	No pos. encoding
0.2829	0.2854	0.2940

Table 5.1: Comparison of final training L_1 loss for different configurations of positional encodings.

5.4 Persisting problems and challenges

We now discuss some persisting problems in our models. We offer some directions on how to solve them in Section 6.4.

- There are audible small artifacts in the spectrograms around the sentence borders at the beginning and at the end. The zero padding does not help remove the artifacts completely, and perhaps a different approach should be taken.
- The model seems to use its capacity to the fullest, because it is not possible to further decrease the loss even with very small learning rate. Increasing the model capacity in some principled way without significantly increasing the number of parameters could further improve the audio quality without sacrificing the training speed.
- The duration predictor tends to predict shorter durations for spaces between words and the model speaks in a slightly higher tempo than the original. This could be caused by imperfect durations extracted with the duration extractor, but more experiments are needed to properly investigate this issue.

6. Evaluation

Our goal was to provide a high-quality speech sythesis system capable of fast training time and speedy inference while requiring low computational resources. To assess the extent to which we were able to succeed, we evaluate our model in terms of voice quality (Section 6.1), inference speed (Section 6.2) and time required for training (Section 6.3).

6.1 Voice quality

Comparison and evaluation of the quality of the outputs of generative models is notoriously difficult (Taylor, 2009, p. 534). For several generative models, it is possible to compare likelihood of the generated samples. Another option is to calculate and compare similarity metrics such as SSIM for the generated and ground-truth data. However, the metrics are not necessarily correlated with human evaluation. For example, Normalizing flows are capable of achieving higher likelihood scores of generated images than GANs, but GANs usually produce outputs that are preffered by humans (Grover et al., 2018). Speech synthesis systems are typically evaluated in terms of *Mean opinion score* (MOS). The mean opinion score is usually obtained by conducting a survey. The survey participants are required to listen to audios sampled for various systems and rate each system on a scale from 1 to five, where five is the best. Mean and confidence intervals are calculated for each system. Then, the scores of each system considered in the survey are compared. We decided to use a similar quality evaluation method called MUSHRA (ITU, 2015; Schoeffler et al., 2015). The MUSHRA score is measured on a finer scale from 0 to 100, which allows more precise comparison. Recordings in MUSHRA survey usually include reference audio, a hidden reference audio, anchor audios and the synthesized audios of interest. The anchors are artificially degraded recordings, whose purpose is to avoid too low score of the recordings of interest due to small imperfections. The hidden reference recording helps to detect participants that do not pay attention to the survey. If the hidden reference receives significantly different score from the revealed reference, the entries of the given participants can be discarded (Latorre et al., 2019).

To measure and compare quality of the synthesized audio, we conducted an online survey with the following setting. We used the last 100 held-out sentences from the LJ Speech dataset (Ito, 2017, see Section 5.1) as test data and synthesized the held-out sentences with our model as well as with two baseline models: Tacotron 2 (Shen et al., 2018, see Section 3.3) and Deep Voice 3 (Ping et al., 2018, see Section 3.3). The corresponding ground-truth recordings were used as a reference.

6.1.1 Survey interface

At the beginning of the survey, the participants were presented with short information about the survey and were asked to jointly evaluate perceived speech fluency and voice quality. Each participant was presented with 10 randomly selected sentence recordings, one sentence per page. On each page, there were 5

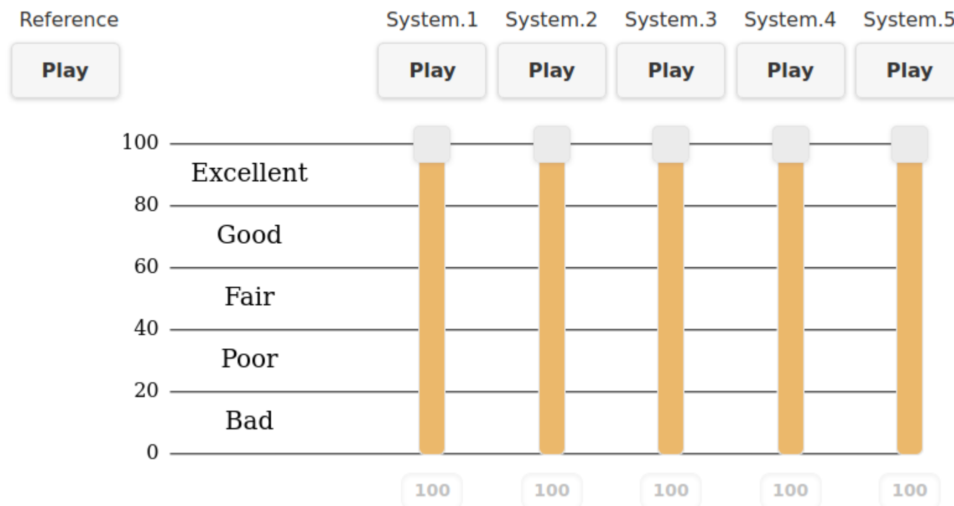

Figure 6.1: The user interface of our survey, based on webMUSHRA (Schoeffler et al., 2018). The survey participant can click the play buttons to play the corresponding samples and slide the sliders to adjust the sample evaluations.

audio recordings of the same sentence to be rated and compared amongst each other (see the listing of systems below). We used a setting similar to MUSHRA and based the survey interface on the webMUSHRA[1] (Schoeffler et al., 2018) project. The recordings were evaluated on a fine-grained scale between 0 and 100 to allow the participants to properly rate recordings with very small differences. The scale was visually divided into 5 categories: "Excellent", "Fair", "Good", "Poor" and "Bad". Participants used sliders to express their preferences. The interface can be seen in Figure 6.1

The displayed order of the recordings was randomised and the model names were hidden from the participant. Unlike MUSHRA, we did not use any anchor recordings since are only interested in a relative comparison of the models. The reference human recording was playable separately, but was also included among the systems as an annotation sanity check. We discarded any participants who rated the reference audio under 90 in 8 or more recordings out of 10.

6.1.2 Compared setups

We selected the following TTS system setups for comparison:

- Tacotron 2 + MelGAN
- Deep Voice 3 + MelGAN
- Deep Voice 3 + lws
- Ours + Griffin-Lim
- Ours + MelGAN
- Reference

[1] https://github.com/audiolabs/webMUSHRA

We used the Nvidia implementation[2] of Tacotron 2. For Deep Voice 3, the implementation from Ryuichi Yamamoto[3] was utilised. For the MelGAN vocoder, we used the implementation and model checkpoint from Seungwon Park.[4] For our own model, we used the same signal processing as in the NVIDIA implementation of Tacotron 2 and MelGAN, in order to be able to use the same vocoder for both models and get more comparable results. Unfortunately, the Deep Voice 3 implementation uses a different signal processing pipeline and a different library for STFT calculation (see Section 2.3) and we were not able to get good results when combining Deep Voice 3 with MelGAN. The Deep Voice 3 implementation uses lws (Le Roux et al., 2010) for vocoding by default. To make our results easier to compare, we also present our model with non-neural vocoding using the Griffin-Lim algorithm (Griffin and Lim, 1984). All models were trained on LJ Speech.

6.1.3 Results

The results of our survey are displayed in Table 6.1. We recruited participants by advertising the interface on Facebook and through the school department mailing list. We used bootstrap resampling (Efron, 1979) to obtain the mean and 95% confidence intervals[5].

Model (vocoding)	**MOS**	**95 % CI**
Tacotron2 (MelGAN)	62.82	$(-2.01, +2.20)$
Deep Voice (lws)	43.61	$(-2.25, +2.20)$
Reference	97.85	$(-0.76, +0.66)$
Ours (Griffin-Lim)	47.03	$(-2.00, +2.16)$
Ours (MelGAN)	**75.24**	$(-1.91, +1.73)$

Table 6.1: Resulting MOS scores from our survey of 40 participants with 95% confidence intervals calculated with bootstrap resampling.

Our model attained the average score of 75 when using MelGAN as a vocoder and scored higher than Tacotron 2 with the same vocoder. (Neekhara et al., 2019) obtained significantly better results when using lws than with Griffin-Lim. Nevertheless, our model was able to achieve higher score with Griffin-Lim than Deep Voice 3 with lws. This shows that our model is clearly preferred to both baselines when used with a similar vocoder.

We performed a small-scale manual analysis of the outputs to find the most likely causes of the higher score for our model. Our model makes virtually no pronunciation mistakes, has consistent speech tempo, does not stutter and has a good voice quality overall. We observed better intonation consistency and word pronunciation in our model compared to the baseline models. We account this to the fact that the baseline models are both sequential. While the sequential models can use the information from already generated frames, they cannot utilize any information from the spectrogram frames that have not been generated yet. This

[2] `https://github.com/NVIDIA/tacotron2`

[3] `https://github.com/r9y9/deepvoice3_pytorch`

[4] `https://github.com/seungwonpark/melgan`

[5] The resampling was done 1000 times

can make the spectrograms more locally accurate, but the global consistency may be lower. In contrast, our model does not condition spectrogram frames on other generated frames, because the frames are all generated in parallel, but is able to aggregate global information across the entire sentence.

6.2 Inference speed

We evaluate the inference speed on a gaming laptop witha 4-core Intel Core i5-6300HQ 2.30 GHz CPU and a 4GB GeForce GTX 960M GPU. Our model, the MelGAN checkpoint and phonemes to synthesize were loaded to the main system or GPU RAM. Then, the wall-clock time from start of the inference to the end of the inference was measured. We used the python build-in time module for our measurements. The inference speed was measured for different batch sizes of fixed length. We created the batches by repeating the same sentence. We used a quote by Antoine de Saint-Exupéry: *"If you want to build a ship, don't drum up people to collect wood and don't assign them tasks and work, but rather teach them to long for the endless immensity of the sea."* There are 34 words composed of 171 characters including punctuation. The sentence can be represented with 143 phonemes to form the input to our network. The generated spectrograms consist of 832 frames and the final audio has 9.72 seconds. We measured the inference 10 times for each configuration and averaged the results. The average inference times are shown in Table 6.2 and Table 6.3.

Batch size	Spectrogram	Audio	Total
1	0.032	0.165	0.197
2	0.035	0.325	0.359
4	0.050	0.647	0.697
8	0.097	1.291	1.388
16	0.203	4.065	4.268

Table 6.2: Average inference time for batches of different size on a 4GB GeForce GTX 960M GPU. The times to produce the spectrogram and the waveform (audio), as well as the total processing time, are reported in seconds. Each produced audio sample in the batch is 9.72 seconds long.

Batch size	Spectrogram	Audio	Total
1	0.105	1.702	1.808
2	0.137	3.211	3.348
4	0.263	6.788	7.051
8	0.591	14.061	14.652
16	1.219	27.685	28.904

Table 6.3: Average inference time for batches of different size on Intel(R) Core(TM) i5-6300HQ CPU @ 2.30GHz. The time is reported in seconds. Each produced audio sample in the batch is 9.72 seconds long.

We are able to synthesize 9.72 seconds of audio in 197 milliseconds on a GPU, which is 49 times faster than real time. On a CPU, we are able to synthesize

Model	Total parameters
Spectrogram generation	4 306 001
MelGAN	4 524 321

Table 6.4: A count of trainable parameters for spectrogram and audio synthesis models.

approximately 5 times faster than real time as the inference takes around 1.8 seconds. Synthesizing batches, we are able to synthesize $16 \times 9.72 = 155.52$ seconds of audio in 4.268 seconds on a GPU, which is over 36 times faster than real time. Our model for spectrogram generation scales well even on a CPU. We did not use any advanced optimizations such as weight pruning or weight quantization. The main speed bottleneck is the MelGAN vocoder (Park, 2020) for audio generation, especially on the CPU where the inference for a batch of size 16 takes over 27 seconds. The comparison of the number of trainable parameters in both models is displayed in Table 6.4. Both models use around 4 million parameters. However, the vocoder generates output in the raw audio domain, which means it has to generate 22,050 samples for one second of audio. Thus, the last convolutional layer in the vocoder has to do relatively many operations compared to the spectrogram generation model. This is acceptable on a GPU where the operations can be parallelized, but is rather expensive on a CPU.

As indicated by Ren et al. (2019), the FastSpeech model should be able to synthesize one spectrogram with 800 frames in around 0.027 seconds on a single NVIDIA V100 GPU, which should be equally fast or slightly faster than our implementation. Unfortunately, Ren et al. (2019) do not provide measured times for different batch sizes or an official implementation that would allow speed comparison on the same machine. Therefore our results are not directly comparable. The Deep Voice 3 implementation does not support the CUDA kernels, as described by Ping et al. (2018). Therefore, the inference speed measured on our machine is not directly comparable either. The inference times for the NVIDIA Tacotron 2 implementation are displayed in Table 6.5. On a CPU, Tacotron 2 was able to generate one sample in 5 seconds on average. Based on the available evidence, our model is comparable to FastSpeech, faster than Tacotron 2 and probably faster than Deep Voice 3 in its current state.

Batch size	Spectrogram	Audio	Total
1	1.552	0.196	1.748
2	1.612	0.388	2.000
4	1.669	0.769	2.438
8	1.877	1.562	3.439
16	2.140	6.796	8.936

Table 6.5: Average inference time for batches of different size on a 4GB GeForce GTX 960M GPU for Tacotron 2. The times to produce the spectrogram and the waveform (audio), as well as the total processing time, are reported in seconds. Tacotron 2 produced spectrograms of approximately 980 frames for the reference sentence, which is over 100 frames more than our model. This also explains why the MelGan inference is slower for Tacotron 2.

6.3 Training time

Both the duration extraction model (see Section 4.3) and the spectrogram synthesis model (see Section 4.4) were trained on a single GeForce GTX 1080 GPU with 8GB RAM. We were able to use a batch of 64 sentences in both cases. The training time on the LJ Speech dataset (Ito, 2017, see Section 5.1) also includes logging of training statistics with Tensorboard. Three audio samples were synthesized every 10 epochs with non-negative least squares linear spectrogram reconstruction and Griffin-Lim (see Section 2.3; MelGAN could not fit on the same GPU anymore). The duration extraction model is smaller in terms of parameters, and finishes one epoch faster than the spectrogram prediction model. However, more than twice as many epochs are required to train the model to convergence because a carefully tuned and comparatively small learning rate must be used to achieve stable learning and convergence to reasonable results (see Section 5.2). The spectrogram prediction model is relatively large and one epoch takes longer to finish. On the other hand, the model architecture is simple and does not contain any hard-to-train layers, such as an attention layer. It is possible to use a large learning rate without any stability issues and the model converges in under 100 epochs (see Section 5.3). The training times along with the total number of parameters in each model are depicted in Table 6.6.

	Duration extraction	Spectrogram prediction
Train time (hours)	19	13
Epochs till convergence	250	100
Time per epoch (minutes)	4.56	7.8
Steps	50 750	20 300
Time per step (seconds)	1.34	2.31
Total params.	708 920	4 306 001

Table 6.6: The wall-clock training duration for the duration extraction model and the spectrogram synthesis model. Total number of parameters in both models as well as the number of epochs and gradient steps are shown. Both models together can be trained on the LJ Speech dataset in 32 hours. The training time was measured on one GeForce GTX 1080 GPU with 8GB RAM.

For comparison, the FastSpeech (Ren et al., 2019) student network alone trains in 80K steps on 4 V100 GPUs.[6] The batch size is also 64, but must be split among the 4 GPUs (batch size 16 per GPU). The FastSpeech student network has 30.1M parameters, which is approximately 7 times more than our proposed student network. The teacher network in FastSpeech is based on a Transformer network (Li et al., 2018; Vaswani et al., 2017) and contains over 30M parameters. According to Li et al. (2018), the teacher model can be trained in around 3 days on LJ Speech (Ito, 2017, see Section 5.1). The Tacotron 2 model (Shen et al., 2018) can be trained in 4.5 days on LJ Speech according to (Li et al., 2018). According to Ping et al. (2018), the Deep Voice 3 model can be trained in 500K steps on a single GPU with batch size 4, where each step takes around

[6] Ren et al. (2019) do not report on the wall clock time taken for training, but given the lower number of parameters in our network and the lower number of steps needed for convergence, we can assume that our model's training needs much less time.

0.06 seconds. This would imply a training time of 8.3 hours. The results in Deep Voice 3 were measured on an internal dataset containing approximately 20 hours of audio, which is comparable to LJ Speech.

In conclusion, our models can be trained orders of magnitude faster than FastSpeech and Tacotron 2. Deep Voice 3 is trained faster than our systems, but provides lower speech quallity according to our survey (see Section 6.1).

6.4 Directions for future work

While our model is able to produce high-quality speech much faster than real time on commodity hardware, there are a few areas that could be explored more and could potentially lead to further improvements. Based on that, we now provide some directions for future work.

Normalization and padding: We observed cutoffs of the last word in a synthesized sentence if the input phonemes were not padded with several frames filled with zeros at the end of the sentence. During batch synthesis, the cutoff always happened only for the longest sentence in the batch. We padded the longest sentence in the batch with several zeros, which resolved this issue. However, a more careful treatment of the phonemes around the edges could be used to eliminate the network dependence on the padding. A promising approach could be the partial-convolution-based padding introduced by Liu et al. (2018), where the convolution results are re-weighted based on the padding area and convolution window overlap.

The zero padding used during training also influences batch normalization. Padded regions decrease the variance estimate and bias the mean estimate towards the pad value. This fact increases the output dependence on the padding used, which may be problematic for synthesis of a single sentence. The batch normalization statistics during inference could be very different from training statistics because there are no padded utterances on the input.

Using masking for the calculation of the statistics could decrease the model dependence on the padded regions. Another option would be to use inputs of similar length in each training batch (bucket sampling) and minimize the necessary padding length. Finally, a different padding scheme could be used. Reflection padding reuses the values along the input edges as padding values, which may help to keep the batch normalization statistics unbiased.

Normalization for duration extraction: It would be interesting to introduce some form of normalization to the duration extraction model (see Section 4.3). This could speed up training and partially alleviate the error propagation problem (see Section 5.2). Normalization of channels across time cannot be directly applied and layer normalization (Ba et al., 2016) did not improve the results in our experiments (see Section 5.2). Recent work on self-normalizing networks offers novel activation functions such as SeLU (Klambauer et al., 2017). Applying similar activation functions would probably be nontrivial and would require careful tuning of the model. However, the model would probably benefit from at least some sort of normalization.

Alternative alignment models: Correct phoneme durations are crucial for training of the spectrogram synthesis model. It does not matter how the durations are obtained as long as they are correct. Therefore, It would be interesting to experiment with alternative models that learn the alignment between phonemes and spectrograms. For example, the AlignTTS model (Zeng et al., 2020) could be adapted for our setup. Similarly, one could experiment with durations extracted from pretrained ASR systems.

We used location-based attention masking as a heuristic to extract monotonic alignment from the attention matrix (see Section 4.3). It would be interesting to adapt Viterbi decoding (Bishop, 2006, p. 629) for extraction of the most likely alignment from the attention matrix. Masking would not be necessary and the alignment would always end on the last phoneme.

Network shape adjustments: Because we were not able to improve the likelihood score of the student network even with very low learning rate (see Section 5.3), we assume that the model already uses all its capacity (expressivenes). The duration predictor seems to be relatively reliable even though it has no influence on the encoder weights (see Section 5.3). We hypothesize that the number of layers in the encoder could be decreased in favor of more layers in the decoder. The decoder would have more transformations available and could generate higher quality spectrograms, while the duration predictor performance would not significantly decrease.

Multi-speaker extensions: We trained our models on a single-speaker dataset. Applications in multi-speaker domains would require extensions of the model to the multi-speaker case. We are curious if the model is able to capture the voices of multiple speakers. Training the model on a different language or on a dataset with a male speaker could also give interesting insights and directions for further improvement.

Better integration with the vocoder: Another quality improvement could come from training the MelGAN vocoder (Park, 2020, see Section 3.2) directly on spectograms synthesized by our model. Recently, the SqueezeWave vocoder (Zhai et al., 2020) was released along with its source code. SqueezeWave is based on WaveGlow (Prenger et al., 2019) but uses various optimizations to improve inference speed. It would be interesting to see a short study that would compare MelGAN and SqueezeWave in terms of training time, inference speed and quality. Some optimizations presented by Zhai et al. (2020) such as depthwise convolutions (Gao et al., 2018) and reshaping of the input in order to avoid the need for dilated convolutions could also be applied to our spectrogram generation model.

Model compression: There are further optimizations such as mixed precision training (Micikevicius et al., 2017) and weight pruning (Molchanov et al., 2016) that could reduce the model size and allow it to run on low-resource devices.

Linear-scale spectrograms: Neural vocoders are able to synthesize high quality audio from mel spectrograms (see Section 3.2). Unfortunately, it is not always possible to use a neural vocoder trained on one speaker for vocoding of

another speaker. Vocoder training for each dataset could be avoided if we used lws (Le Roux et al., 2010) or Griffin-Lim (Griffin and Lim, 1984) for the spectrogram inversion (see Section 2.3). However, to achieve high-quality output audio, Griffin Lim and lws require linear spectrograms on the input. Therefore, it is necessary to reconstruct the linear scale spectrograms from the mel-scale spectrograms. Using non-negative least squares to do this is slow, inaccurate, and has a large impact on the final audio quality. It could thus be interesting to generate linear scale spectrograms from the mel spectrograms with a super-resolution GAN and apply a phase estimation technique on the final spectrogram similarly to (Neekhara et al., 2019). If the linear spectrograms were high-quality, the resulting audio quality could be sufficient and there would be no need for a neural vocoder.

STFT implementations mismatch: Finally, we would like to address an issue that is not only relevant to our work, but also to the speech synthesis research in general. There are many available implementations of STFT and accompanying tools:

- librosa[7]
- torch.stft[8]
- nnAudio[9]
- torchaudio[10]
- lws[11]

Furthermore, there are project-specific implementations such as the Pytorch implementation used in Nvidia Tacotron 2[12]. Different speech synthesis projects often utilize different implementations of STFT. Unfortunately, some of the STFT implementations are incompatible even if the STFT parameters are identical. This makes comparison and combining of different models very hard. For example, we tried to combine spectrograms generated with the STFT implementation from Nvidia Tacotron 2 with Griffin-Lim implementation from librosa, which yielded lower-quality audio than using spectrograms generated with the librosa. A comprehensive survey comparing different STFT frameworks and providing compatibility information would be really useful, especially for researchers in the field of speech synthesis. Such a survey could speed up the research process and might make future spectrogram synthesis models and vocoders mutually widely compatible without retraining.

[7]https://github.com/librosa/librosa
[8]https://pytorch.org/docs/stable/torch.html#torch.stft
[9]https://github.com/KinWaiCheuk/nnAudio
[10]https://pytorch.org/audio/
[11]https://github.com/Jonathan-LeRoux/lws
[12]https://github.com/NVIDIA/tacotron2/blob/master/stft.py

7. Conclusion

We presented a convolutional model for spectrogram synthesis from phonemes that supports both speedy training and inference, while maintaining significantly better output voice quality than strong baselines (see Section 6). We believe all our goals set in Section 1.2 were accomplised.

We trained our model on a standard dataset and conducted comprehensive experiments with different model configurations in Section 5. The entire system can be trained in 32 hours on one GeForce GTX 1080 GPU with 8GB RAM. A quality evaluation and comparison to other recent speech synthesis systems was done (Section 6.1). Further, we measured the model speed during inference on both GPU and CPU (Section 6.2). We also compared our model to baseline models in terms of required training time in Section 6.3. The results suggest that our system achieves state-of-the-art in terms of the trade-off between synthesized voice quality and training and inference time. The system is able to generate significantly better audio samples than similar synthesis models while being faster to train and run. We also provided insight into challenges of training speech synthesis models and suggested further improvements and directions for future research.

Bibliography

Jont B. Allen and Lawrence R. Rabiner. A Unified Approach to Short-Time Fourier Analysis and Synthesis. *Proceedings of the IEEE*, 65(11):1558–1564, 1977.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. In *Proceedings of the Neural Information Processing Systems Deep Learning Symposium*, Barcelona, Spain, December 2016.

Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*. International Conference on Learning Representations, ICLR, 9 2015.

Mikołaj Bińkowski, Jeff Donahue, Sander Dieleman, Aidan Clark, Erich Elsen, Norman Casagrande, Luis C. Cobo, and Karen Simonyan. High Fidelity Speech Synthesis with Adversarial Networks. *arXiv preprint arXiv:1909.11646*, 9 2019.

Christopher M. Bishop. *Pattern Recoginiton and Machine Learning*. Springer-Verlag New York, 2006.

Donghui Chen and Robert J. Plemmons. Nonnegativity constraints in numerical analysis. In *Symposium on the Birth of Numerical Analysis, World Scientific*. Press, 2009.

Chris Donahue, Julian McAuley, and Miller Puckette. Adversarial audio synthesis. In *ICLR*. International Conference on Learning Representations, ICLR, 2 2019.

H W Dudley. System for the artificial production of vocal or other sounds. Technical report, 4 1938.

Homer Dudley and T. H. Tarnoczy. The Speaking Machine of Wolfgang von Kempelen. *Journal of the Acoustical Society of America*, 22(2):151–166, 3 1950.

Bradley Efron. Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics*, 7(1):1–26, January 1979.

Hongyang Gao, Zhengyang Wang, and Shuiwang Ji. ChannelNets: Compact and efficient convolutional neural networks via channel-wise convolutions. In *Advances in Neural Information Processing Systems*, volume 2018-Decem, pages 5197–5205. Neural information processing systems foundation, 2018.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, volume 3, pages 2672–2680. Neural information processing systems foundation, 6 2014.

Daniel W. Griffin and Jae S. Lim. Signal Estimation from Modified Short-Time Fourier Transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 32(2):236–243, April 1984.

Aditya Grover, Manik Dhar, and Stefano Ermon. Flow-gan: Bridging implicit and prescribed learning in generative models. In *AAAI*, 2018.

Fredric J. Harris. On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform. In *Proceedings of the IEEE*, volume 66, pages 51–83, 1978.

T. Hastie, R. Tibshirani, J. Friedman, and J. Franklin. *The elements of statistical learning: data mining, inference and prediction.* Springer, 2nd edition, 2009.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2016-Decem, pages 770–778. IEEE Computer Society, 12 2016.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1 1989.

Satoshi Imai. Cepstral Analysis Synthesis on the Mel Frequency Scale. In *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, volume 1, pages 93–96. IEEE, 1983.

Sergey Ioffe and Christian Szegedy. Batch normalization: accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, pages 448–456, Lille, France, July 2015.

Keith Ito. The LJ Speech Dataset, 2017. URL `https://keithito.com/LJ-Speech-Dataset/`.

ITU. Method for the subjective assessment of intermediate quality level of audio systems. Recommendation BS.1534, International Telecommunication Union, Geneva, 2015.

Ye Jia, Yu Zhang, Ron J. Weiss, Quan Wang, Jonathan Shen, Fei Ren, Zhifeng Chen, Patrick Nguyen, Ruoming Pang, Ignacio Lopez Moreno, and Yonghui Wu. Transfer learning from speaker verification to multispeaker text-to-speech synthesis. In *Advances in Neural Information Processing Systems*, volume 2018-Decem, pages 4480–4490, 2018.

Nal Kalchbrenner, Erich Elsen, Karen Simonyan, Seb Noury, Norman Casagrande, Edward Lockhart, Florian Stimberg, Aaron van den Oord, Sander Dieleman, and Koray Kavukcuoglu. Efficient Neural Audio Synthesis. pages 2410–2419, July 2018.

Nasser Kehtarnavaz. *Digital signal processing system design.* Elsevier Inc., 2008.

Diederik Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, San Diego, CA, USA, May 2015.

Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-Normalizing Neural Networks. In *Advances in Neural Information Processing Systems*, volume 2017-Decem, pages 972–981. Neural information processing systems foundation, 6 2017.

Dennis H Klatt. Software for a cascade/parallel formant synthesizer. *The Journal of the Acoustical Society of America*, 67(3):971–995, 3 1980.

Dennis H Klatt. Review of text-to-speech conversion for English. *The Journal of the Acoustical Society of America*, 82(3):737–793, 9 1987.

Ivan Kobyzev, Simon Prince, and Marcus A. Brubaker. Normalizing Flows: An Introduction and Review of Current Methods. 8 2019.

Kundan Kumar, Rithesh Kumar, Thibault de Boissiere, Lucas Gestin, Wei Zhen Teoh, Jose Sotelo, Alexandre de Brebisson, Yoshua Bengio, and Aaron Courville. MelGAN: Generative Adversarial Networks for Conditional Waveform Synthesis. In *Advances in Neural Information Processing Systems 32 (NeurIPS)*, pages 14910–14921, Vancouver, BC, Canada, December 2019.

Javier Latorre, Jakub Lachowicz, Jaime Lorenzo-Trueba, Thomas Merritt, Thomas Drugman, Srikanth Ronanki, and Klimkov Viacheslav. Effect of data reduction on sequence-to-sequence neural tts. pages 7075–7079, 2019.

Jonathan Le Roux, Hirokazu Kameoka, Nobutaka Ono, and Shigeki Sagayama. Fast signal reconstruction from magnitude STFT spectrogram based on spectrogram consistency. In *Proceeedings of the 13th International Conference on Digital Audio Effects (DAFx)*, Graz, Austria, September 2010.

Xiang Li, Shuo Chen, Xiaolin Hu, and Jian Yang. Understanding the Disharmony between Dropout and Batch Normalization by Variance Shift. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2677–2685, 1 2018.

Guilin Liu, Kevin J. Shih, Ting-Chun Wang, Fitsum A. Reda, Karan Sapra, Zhiding Yu, Andrew Tao, and Bryan Catanzaro. Partial Convolution based Padding. *arXiv preprint arXiv:1811.11718*, 11 2018.

Michael McAuliffe, Michaela Socolof, Sarah Mihuc, Michael Wagner, and Morgan Sonderegger. Montreal Forced Aligner: Trainable Text-Speech Alignment Using Kaldi. In *Interspeech 2017*, volume 2017-Augus, pages 498–502. ISCA, 8 2017.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed Precision Training. In *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*. International Conference on Learning Representations, ICLR, 10 2017.

Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning Convolutional Neural Networks for Resource Efficient Inference. In *5th*

International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings. International Conference on Learning Representations, ICLR, 11 2016.

Paarth Neekhara, Chris Donahue, Miller Puckette, Shlomo Dubnov, and Julian McAuley. Expediting TTS synthesis with adversarial vocoding. In *INTERSPEECH*, volume 2019-Septe, pages 186–190, 2019.

Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. WaveNet: A Generative Model for Raw Audio. *arXiv preprint arXiv:1609.03499*, September 2016.

Tom Le Paine, Pooya Khorrami, Shiyu Chang, Yang Zhang, Prajit Ramachandran, Mark A. Hasegawa-Johnson, and Thomas S. Huang. Fast Wavenet Generation Algorithm. *arXiv preprint arXiv:1611.09482*, 11 2016.

Seungwon Park. MelGan vocoder, 2020. URL `https://github.com/seungwonpark/melgan`.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318, 2013.

Wei Ping, Kainan Peng, Andrew Gibiansky, Sercan Arık, Ajay Kannan, Sharan Narang, Jonathan Raiman, and John Miller. Deep Voice 3: Scaling text-to-speech with convolutional sequence learning. In *Proceedings of the 6th International Conference on Learning Representations (ICLR)*, Vancouver, BC, Canada, May 2018.

Wei Ping, Kainan Peng, and Jitong Chen. Clarinet: Parallel wave generation in end-to-end text-to-speech. In *Proceedings of the Seventh International Conference on Learning Representations (ICLR)*, New Orleans, LA, USA, May 2019.

Martin Popel and Ondřej Bojar. Training Tips for the Transformer Model. *The Prague Bulletin of Mathematical Linguistics*, 110(1):43–70, 3 2018.

Ryan Prenger, Rafael Valle, and Bryan Catanzaro. Waveglow: A Flow-based Generative Network for Speech Synthesis. In *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, volume 2019-May, pages 3617–3621. Institute of Electrical and Electronics Engineers Inc., 5 2019.

Yi Ren, Yangjun Ruan, Xu Tan, Tao Qin, Sheng Zhao, Zhou Zhao, and Tie-Yan Liu. FastSpeech: Fast, Robust and Controllable Text to Speech. In *Advances in Neural Information Processing Systems 32 (NeurIPS)*, pages 3171–3180, Vancouver, BC, Canada, December 2019.

Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

Y Sagisaka. Speech synthesis by rule using an optimal selection of non-uniform synthesis units. In *ICASSP-88., International Conference on Acoustics, Speech, and Signal Processing*. IEEE, 1988.

Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. pages 901–909, 2016.

Stan Salvador and Philip Chan. FastDTW: Toward Accurate Dynamic Time Warping in Linear Time and Space. *Intelligent Data Analysis*, 01 2004.

Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *ICANN*, pages 92–101. Springer, Berlin, Heidelberg, 2010.

Michael Schoeffler, Fabian-Robert Stöter, Bernd Edler, and Jürgen Herre. Towards the next generation of web-based experiments: A case study assessing basic audio quality following the ITU-R recommendation BS. 1534 (MUSHRA). In *1st Web Audio Conference*, pages 1–6, 2015.

Michael Schoeffler, Sarah Bartoschek, Fabian-Robert Stöter, Marlene Roess, Susanne Westphal, Bernd Edler, and Jürgen Herre. webMUSHRA — A Comprehensive Framework for Web-based Listening Tests. *Journal of Open Research Software*, 6(1):8, February 2018.

Diemo Schwarz. Current research in concatenative sound synthesis. 2005.

Jonathan Shen, Ruoming Pang, Ron J. Weiss, Mike Schuster, Navdeep Jaitly, Zongheng Yang, Zhifeng Chen, Yu Zhang, Yuxuan Wang, Rj Skerrv-Ryan, Rif A. Saurous, Yannis Agiomvrgiannakis, and Yonghui Wu. Natural TTS Synthesis by Conditioning Wavenet on MEL Spectrogram Predictions. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4779–4783, Calgary, AB, Canada, April 2018.

Leyuan Sheng and Evgeniy N. Pavlovskiy. Reducing over-smoothness in speech synthesis using Generative Adversarial Networks. In *SIBIRCON 2019 - International Multi-Conference on Engineering, Computer and Information Sciences, Proceedings*, pages 972–974, 10 2019.

Jose Sotelo, Soroush Mehri, Kundan Kumar, João Felipe Santos, Kyle Kastner, Aaron Courville, and Yoshua Bengio. Char2Wav: End-to-End Speech Synthesis. In *ICLR*, 2 2017.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, June 2014.

Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway Networks. *arXiv preprint arXiv:1505.00387*, 5 2015.

S. S. Stevens, J. Volkmann, and E. B. Newman. A Scale for the Measurement of the Psychological Magnitude Pitch. *Journal of the Acoustical Society of America*, 8(3):185–190, 1937.

Hideyuki Tachibana, Katsuya Uenoyama, and Shunsuke Aihara. Efficiently Trainable Text-to-Speech System Based on Deep Convolutional Networks with Guided Attention. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings (ICASSP)*, pages 4784–4788, Calgary, AB, Canada, April 2018.

Paul Taylor. *Text-to-speech synthesis*, volume 9780521899. Cambridge University Press, 1 2009.

Keiichi Tokuda, Yoshihiko Nankaku, Tomoki Toda, Heiga Zen, Junichi Yamagishi, and Keiichiro Oura. Speech synthesis based on hidden Markov models. In *Proceedings of the IEEE*, volume 101, pages 1234–1252. Institute of Electrical and Electronics Engineers Inc., 2013.

Aaron Van Den Oord, Yazhe Li, Igor Babuschkin, Karen Simonyan, Oriol Vinyals, Koray Kavukcuoglu, George Van Den Driessche, Edward Lockhart, Luis C. Cobo, Florian Stimberg, Norman Casagrande, Dominik Grewe, Seb Noury, Sander Dieleman, Erich Elsen, Nal Kalchbrenner, Heiga Zen, Alex Graves, Helen King, Tom Walters, Dan Belov, and Demis Hassabis. Parallel WaveNet: Fast high-fidelity speech synthesis. In *Proceedings of the 35th International Conference on Machine Learning (ICML)*, pages 6270–6278, Stockholm, Sweden, July 2018.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30 (NeurIPS)*, pages 5999–6009, Long Beach, CA, USA, December 2017.

Zhou Wang, Alan Conrad Bovik, Hamid Rahim Sheikh, and Eero P. Simoncelli. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, April 2004.

Sam Wiseman and Alexander M Rush. Sequence-to-Sequence Learning as Beam-Search Optimization. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1296–1306, Austin, Texas, 11 2016. Association for Computational Linguistics.

Yanzhao Wu, Ling Liu, Juhyun Bae, Ka-Ho Chow, Arun Iyengar, Calton Pu, Wenqi Wei, Lei Yu, and Qi Zhang. Demystifying Learning Rate Policies for High Accuracy Training of Deep Neural Networks. In *Proceedings - 2019 IEEE International Conference on Big Data, Big Data 2019*, pages 1971–1980. Institute of Electrical and Electronics Engineers Inc., 8 2019.

Heiga Zen, Keiichi Tokuda, Takashi Masuko, Takao Kobayashi, and Tadashi Kitamura. Hidden semi-Markov model based speech synthesis. In *Eighth International Conference on Spoken Language Processing*, 2004.

Heiga Zen, Keiichi Tokuda, and Alan W. Black. Statistical parametric speech synthesis. In *2007 IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '07*, volume 51, pages 1039–1064, 11 2009.

Zhen Zeng, Jianzong Wang, Ning Cheng, Tian Xia, and Jing Xiao. AlignTTS: Efficient Feed-Forward Text-to-Speech System without Explicit Alignment. *arXiv preprint arXiv:2003.01950*, 3 2020.

Bohan Zhai, Tianren Gao, Flora Xue, Daniel Rothchild, Bichen Wu, Joseph E. Gonzalez, and Kurt Keutzer. SqueezeWave: Extremely Lightweight Vocoders for On-device Speech Synthesis. *arXiv preprint arXiv:2001.05685*, 1 2020.

Maggie Zhang and Grzegorz Karch. Generate Natural Sounding Speech from Text in Real-Time | NVIDIA Developer Blog, 9 2019. URL `https://devblogs.nvidia.com/generate-natural-sounding-speech-from-text-in-real-time/`.

A. Appendix

STFT parameters

STFT	
sample rate	22050
n_mels	80
n_fft	1024
hop size	256
window size	1024
mel_fmin	0.0
mel_fmax	8000.0
clip value	1e-5

Table A.1: The list of parameters used for discrete short-time Fourier transform (see Section 2.3). The *sample rate* refers to the audio sample rate; one second of audio contains 22050 samples. The parameters *n_fft* and *n_mels* refer to the number of frequency and mel frequency bins used. The *window size* is the width of the STFT window and *hop size* is the overlap size of neighboring windows. Parameters *mel_fmin* and *mel_fmax* refer to minimum and maximum frequencies considered for the mel spectrograms. The *clip value* is the minimum value allowed in a spectrogram. Lower values get clamped to this value.

Parameters of the duration extraction model

Parameter	Text encoder	Audio encoder	Decoder
Layers	10	10	14
Dilation factors	2 * [1, 3, 9, 27] + [1,1]	2 * [1, 3, 9, 27] + [1,1]	2 * [1, 3, 9, 27] + [1,1]
Kernel size	3	3	3
Final activation	-	-	Sigmoid
Convolution channels	40	40	40
Hidden channels	80	80	80

Training	
Optimizer	Adam
Learning rate	0.002
Scheduler	Noam, warmup 30 epochs
Gradient clipping	1
Batch size	64
Guided att. weight	0.3
Pos. encoding slope	6.42
Attention noise std	0.1
Replace random frame proba.	0.1
Spectrogram noise std	0.01
Self feed repeats	2

Table A.2: The list of parameters of the duration extraction architecture and paramaters for training (see Sections 4.3 and 5.2)

Parameters of the speech synthesis model

Parameter	Encoder	Decoder	Duration predictor
Residual blocks	13	17	3
Length of residual block	2	2	1
Dilation factors	4 * [1,2,4] + [1]	4 * [1,2,4,8] + [1]	[4,3,1,1]
Kernel size	4	4	4
Batch norm.	affine	affine	affine
Activation	ReLU	ReLU	ReLU
Final activation	-	-	Linear
Convolution channels	128	128	128

Training	
Optimizer	Adam
Learning rate	0.002
Scheduler	ReduceOnPlateau, patience=3
Gradient clipping	1
Batch size	64

Table A.3: The list of parameters of the final spectrogram generator architecture and paramaters for training (see Sections 4.4 and 5.3)

.

List of Figures

2.1 The spectra and approximate spectral envelopes for vowels “a” on the top and “e” on the bottom. The peaks correspond to resonating harmonics, while the valleys correspond to suppressed harmonics. 8
2.2 Log-magnitude spectogram with 512 frequency bins of a sentence *"Printing, in the only sense with which we are at present concerned, differs from most if not from all the arts and crafts represented in the Exhibition"*. 10
2.3 Log-magnitude mel spectogram with 80 mel frequency bins of a sentence *"Printing, in the only sense with which we are at present concerned, differs from most if not from all the arts and crafts represented in the Exhibition"*. 10
2.4 One-dimensional non-causal convolution with four input channels, three output channels and three kernels. The green filters transform the input sequence (bottom) to output sequence (top). The length of the output sequence is decreased due to kernel size larger than 1. 15
2.5 Dilated temporal non-causal convolutions with dilations 1 (bottom layer) and 3 (top layer), kernel size 3 and stride 1. Dashed circles represent zero padding necessary to keep the same output size in each layer. For visualisations of dilated temporal causal convolutions, see (Oord et al., 2016). 16
2.6 A computational graph for a temporal convolutional network with causal convolutions. The function f maps a window of input elements to an output element. For example, the function f can be composed of a convolution operation, normalization and non-linear activation such as ReLU. We omit the dependence on $\mathbf{X}$ for simplicity. 17
2.7 A computational graph for a recurrent network. The variable h_i represents the hidden state carried between timesteps. The function f maps the hidden state from the previous timestep $\vec{h}_{i-1}$ and the input $\vec{y}_{i-1}$ at current timestep to output $\vec{y}_i$. We omit the dependence on $\mathbf{X}$ for simplicity. 18
2.8 A computational graph for a conditioned recurrent network. . . . 18
2.9 An attention distribution for each timestep. The horizontal axis represents the query sequence. The vertical axis represents the key/value sequence. Each column forms the attention distribution over keys for the given query. Dark color is close to zero, light color is close to 1. We can extract the alignment of keys and queries by taking the argmax of each column. 19
2.10 The positional encoding matrix. The horizontal axis represents time. The vertical axis represents dimensionality of the encoding. 20

3.1 The full architecture of Tacotron 2. Source: Shen et al. (2018) . . 28

3.2 The components of the FastSpeech architecture. The full model (left) consists of an encoder, a length regulator and a decoder. The encoder and the decoder consist of stacks of FFT blocks (second from the left). The encoder produces contextualized phoneme embeddings. The length regulator (second from the right) expands the embeddings based on predicted phoneme durations. The decoder synthesizes a spectrogram from the expanded embeddings. The durations are predicted by a duration prediction module (right). Source: Ren et al. (2019) . . 30

4.1 A gated residual block. The convolutions are progressively dilated to increase the receptive field. The blocks' skip connection sums outputs from all layers to form the network output. The "dot" symbol represents element-wise multiplication and the "plus" symbol represents element-wise addition. The "FC" block stands for a fully connected layer. 33
4.2 The network used for duration extraction from data. The positional encoding is element-wise added to the queries and a fully connected layer is applied on each column. The fully connected layers for keys and queries share weights. The gates inside the residual blocks are not visualized for better readability and dilated convolutions are displayed instead. 34
4.3 The value of $\mathbf{p_k} \cdot \mathbf{p_l}$ as k and l become more distant. The horizontal axis represents the distance $|k - l|$. The vertical axis represents the value of the dot product. The more channels are added to the positional encoding, the longer the period is. 35
4.4 Dot product of positional encoding vectors for combinations of positions between 0 to 1000. A point (m, n) in each of the sub-figures represents the value of the dot product of $\mathbf{p}_m$ and $\mathbf{p}_n$. Positional encoding in the sub-figures has 4, 8 and 80 channels. As the number of channels increases, the periodicity also increases and for 80 channels, values outside the diagonal are relatively small compared to the diagonal. 36
4.5 The penalty matrix $\mathbf{w}$ for parameter values $g = 0.1$ (left) and $g = 0.3$ (right). Dark colors signalize low penalty, while bright colors signalize high penalty. The penalty is almost zero near the diagonal. 37
4.6 Results of intermediate computations are stored in layer-level queues. The items are progressively released at each timestep. Thanks to the cached values, it is not necessary to recalculate the operations in the subtree of the given node. Source: Paine et al. (2016). 39

4.7 Our spectrogram synthesis model. The encoder takes one-hot encoded phonemes on input and outputs contextualized phoneme embeddings. The duration predictor accepts the embeddings and predicts the duration of each phoneme. Each phoneme embedding is copied according to the predicted durations. The decoder accepts the expanded phoneme embeddings summed with the positional encoding matrix and synthesizes a mel-scale spectrogram on output. 40
4.8 Positional encoding frames are mapped to each expanded character/phoneme separately. The grey tiles represent positional encoding frames. The colored tiles represent expanded encoder outputs for the word "and". 42

5.1 Guided attention loss before and after applying dropout with probability 0.1. The dropout did not improve alignment robustness and caused worse performance. 46
5.2 Alignment generated by the duration extraction model. The y axis captures phonemes, while the x axis captures spectrogram frames. The window size was set to 2. The generated sentence is *"The quick brown fox jumped over a lazy dog."* 48
5.3 A log scale mel spectrogram of the sentence *"The quick brown fox jumped over the lazy dog."* generated by our spectrogram generation model. 49
5.4 Huber loss of predicted phoneme durations. The durations from the model that was trained with detached gradients of the duration predictor and was trained to predict logarithmic-scale durations attains the lowest loss. Before calculating the loss, the exponential of the logarithmic durations was taken so that the losses of different configurations could be compared. The configuration of the duration prediction module was the same in all configurations. . . 51
5.5 Reduce-on-plateau and Noam learning rates (see Figure 5.6 for model training performance). 52
5.6 L1 spectrogram loss for Noam and reduce-on-plateau schedulers (see Figure 5.5). The model trained with Noam scheduler required around 250 epochs to converge to the final loss value reached in under 100 epochs by the model trained with the reduce-on-plateau scheduler. 52

6.1 The user interface of our survey, based on webMUSHRA (Schoeffler et al., 2018). The survey participant can click the play buttons to play the corresponding samples and slide the sliders to adjust the sample evaluations. 56

List of Tables

3.1 A list of selected neural TTS models divided into categories based on their synthesis scope. The systems marked with "*" were originally designed as end-to-end but with slight modifications can be used to synthesize audio from spectrograms. 25

5.1 Comparison of final training L_1 loss for different configurations of positional encodings. 54

6.1 Resulting MOS scores from our survey of 40 participants with 95% confidence intervals calculated with bootstrap resampling. 57
6.2 Average inference time for batches of different size on a 4GB GeForce GTX 960M GPU. The times to produce the spectrogram and the waveform (audio), as well as the total processing time, are reported in seconds. Each produced audio sample in the batch is 9.72 seconds long. 58
6.3 Average inference time for batches of different size on Intel(R) Core(TM) i5-6300HQ CPU @ 2.30GHz. The time is reported in seconds. Each produced audio sample in the batch is 9.72 seconds long. 58
6.4 A count of trainable parameters for spectrogram and audio synthesis models. 59
6.5 Average inference time for batches of different size on a 4GB GeForce GTX 960M GPU for Tacotron 2. The times to produce the spectrogram and the waveform (audio), as well as the total processing time, are reported in seconds. Tacotron 2 produced spectrograms of approximately 980 frames for the reference sentence, which is over 100 frames more than our model. This also explains why the MelGan inference is slower for Tacotron 2. . . . 59
6.6 The wall-clock training duration for the duration extraction model and the spectrogram synthesis model. Total number of parameters in both models as well as the number of epochs and gradient steps are shown. Both models together can be trained on the LJ Speech dataset in 32 hours. The training time was measured on one GeForce GTX 1080 GPU with 8GB RAM. 60

A.1 The list of parameters used for discrete short-time Fourier transform (see Section 2.3). The *sample rate* refers to the audio sample rate; one second of audio contains 22050 samples. The parameters *n_fft* and *n_mels* refer to the number of frequency and mel frequency bins used. The *window size* is the width of the STFT window and *hop size* is the overlap size of neighboring windows. Parameters *mel_fmin* and *mel_fmax* refer to minimum and maximum frequencies considered for the mel spectrograms. The *clip value* is the minimum value allowed in a spectrogram. Lower values get clamped to this value. 72

A.2 The list of parameters of the duration extraction architecture and paramaters for training (see Sections 4.3 and 5.2) 73
A.3 The list of parameters of the final spectrogram generator architecture and paramaters for training (see Sections 4.4 and 5.3) 74