UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER THESIS

Deep learning applied to Speech Synthesis

Author:
Santiago PASCUAL DE LA PUENTE

Supervisor:
Dr. Antonio BONAFONTE CAVEZ

A thesis submitted in fulfillment of the requirements for the degree of Master in Telecommunications Engineering

in the

TALP Research Center
Signal Theory and Communications Department
Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

June 30, 2016

"We are drowning in information but starved for knowledge."

John Naisbitt

Abstract

Deep Learning has been applied successfully to speech processing problems. In this work we explore its capabilities, focusing concretely in recurrent neural architectures to build a state of the art Text-To-Speech system from scratch. The different steps to make the full TTS system are shown. Also, a post-filtering method to improve the generated speech naturalness is applied and evaluated. The objective results show which architecture fits better our problem, achieving low error rates in term of cepstral distortion, pitch estimation error and voiced/unvoiced classification error. Also, subjective results suggest that the model achieves a state of the art quality in the synthesis, where the post-filtering factor seems to be a key component to get a good level of naturalness.

A novel architecture called Multi-Output TTS is also proposed to hold multiple speakers inside the same structure. Some hidden layers are shared by all the speakers, while there is a specific output layer for each speaker. Objective and perceptual experiments prove that this scheme produces much better results in comparison with single speaker models. Moreover, we also tackle the problem of speaker adaptation by adding a new output branch to the model and successfully training it without the need of modifying the base optimized model. This fine tuning method achieves better results than training the new speaker from scratch with its own model.

Moreover, we also tackle the problem of speaker interpolation by adding a new output layer (α-layer) on top of the Multi-Output branches. An identifying code is injected into the layer together with acoustic features of many speakers. Experiments show that the α-layer can effectively learn to interpolate the acoustic features between speakers.

Acknowledgements

First, I would like to thank my advisor and mentor, Antonio Bonafonte, who taught me what I know of the speech synthesis field and never hesitates to help, always supporting the ideas I come up with.

Also, I would like to thank Enric Monte for the great background he gave me in the machine learning field, and José Adrián Rodríguez Fonollosa for solving any question I had regarding software issues.

Finally, thanks to all those people who helped in the development of the experiments by taking our subjective evaluations so kindly.

Contents

Abstract v

Acknowledgements vii

1 Introduction 1

2 State of the Art 3

2.1 Unit Selection Speech Synthesis 3
2.2 Statistical Parametric Speech Synthesis 4
2.3 Deep Learning in Speech Synthesis 9
2.4 Summary 12

3 Introduction to Deep Learning 13

3.1 Artificial Neural Network 13
3.2 Training the network: Back-Propagation algorithm 18
3.2.1 Dropout Method 23
3.3 Deep Neural Network 26
3.4 Recurrent Neural Network 26
3.5 Back-Propagation Through Time 28
3.6 Long Short Term Memory 30
3.7 Summary 32

4 Two stage Text-to-Speech with RNN-LSTM 33

4.1 Introduction 33
4.2 Data Preparation 35
4.2.1 Text to Label process 35
4.2.2 Acoustic parameters 38
4.2.3 Encoding and normalizing the features 40
4.2.4 Parallelizing the pipeline: Speeding up the data generation 42
4.3 Two-stage RNN-LSTM model 46
4.4 A post-processing technique for better naturalness 52
4.5 Experimental Design and Results 55
4.5.1 Database 57

4.5.2 Objective Evaluationt 58
4.5.3 Subjective Evaluation 60
4.5.4 Gate activations analysis 63
4.6 Discussion 68

5 Multiple Output Acoustic Mapping 69
5.1 Introduction 69
5.2 Data Preparation 69
5.3 Multi-Output architecture for acoustic mapping 70
5.4 Multi-Output architecture for speaker adaptation 71
5.5 Multi-Output architecture for speaker interpolation: α-model 72
5.6 Results 74
5.6.1 Results: Multi-output 76
5.6.2 Results: Adaptation 78
5.6.3 Results: α-interpolation 78
5.7 Discussion 82

6 Conclusions 83
6.1 Thesis Review 83
6.2 Future work 85
6.3 Research Contribution 86

Bibliography 87

List of Figures

2.1 Voice stream where blue dashed lines show hypothetic phoneme divisions and green lines show hypothetic diphoneme divisions. . . . 3
2.2 Unit selection scheme. . . . 4
2.3 Unit selection synthesis process. Dashed lines represent the concatenation cost. Solid lines represent the target cost. Grey lines depict the computation performed against all units. Blue lines are the computation of the actual used units. Based on Zen, Tokuda, and Black (2009) Figure 1. . . . 5
2.4 Hidden Markov Model example with 3 states. Based on Ling et al. (2015), FIG1. . . . 6
2.5 Block diagram of typical Statistical Parametric Speech Synthesis HMM-based systems. Based on Ling et al. (2015), FIG2. 8
2.6 Hidden Markov Model clustering process. Based on Ling et al. (2015), FIG3. . . . 8

3.1 Artificial Neuron 14
3.2 Sigmoid function with different scalar weightings and bias: red $w = 5$, $b = 0$; blue $w = 10$, $b = 0$; green $w = 15$, $b = 9$. . . 14
3.3 Logistic Regression of N outputs. . . . 16
3.4 XOR resulting approximate surface. . . . 17
3.5 XOR Neural Network with Sigmoid perceptrons. . . . 17
3.6 Calculation of δ_j for the hidden neuron j back-propagating the δ's from the connected n units (which can be output units or next-hidden units). Based on (Bishop, 1995) Figure 4.16. . 21
3.7 Qualitative examples of training loss evolution for various learning rate magnitudes. . . . 23
3.8 Examples of a good fit and an overfitting for a binary classification task with a $2 - D$ space (i.e. 2 input features). . . . 24
3.9 (a) Standard network with 2 hidden layers. (b) Example of thinned network by applying dropout to the network on (a). Based on (Srivastava et al., 2014) Figure 1. . . . 25
3.10 Example RNN neuron unfolded in time. . . . 27
3.11 Gradient flow at the fourth time-step 3. Violet arrows: inference flow of the input data. Red arrows: back-propagated gradients/errors. . . . 29

3.12 Architecture of an LSTM cell. i(t): input gate at time-step t. o(t): output gate at time-step t. f(t): forget gate at time-step t. c(t): cell state at time-step t. 30
3.13 LSTM cell unfolded in time. The red arrows depict the inference flow of data between time-steps. 32

4.1 Schematic of the developed two stage Text-To-Speech system. 34
4.2 Label file example resulting from processing the Spanish text *"buenos días"*. 37
4.3 Schematic representation of the Vocoder for encoding the input voice with windowed frames into acoustic parameters. . 39
4.4 Histogram of WAV files' durations for speaker M1. It is shown in a scale of seconds. 43
4.5 Data generation architecture with parallelized pipeline. Every pipeline processes a tuple (lab,wav) and accumulates the result to be given to the data generator. 44
4.6 WAV files' durations histograms for M1 and F1 subsets of 100 files each. 45
4.7 Results of data generation acceleration for speakers M1 and F1. M1-1x: 63 min. M1-32x: 3 min. F1-1x: 22 min. F1-32x: 1 min. 45
4.8 Vocoder sliding window example for an arbitrary phoneme. The stride is the increment in time taken by the next window $\Delta t = t_{i+1} - t_i$. The red zone shows a hypothetical empty zone out of the phoneme, so the frame duration is including information from the next phoneme or a silence. The grey arrow is the direction of the windowing analysis. 47
4.9 Duration histograms for speaker M1 phonemes. Top plot shows the real durations in milliseconds, bottom plot shows the log-compressed durations. 48
4.10 Plot of $|\hat{y} - y|^2$ against $|\hat{y} - y|$ and the derivative with respect to the error:$2|\hat{y} - y|$. 49
4.11 Example of a regression estimation where we see how the outlier produces a high deviation over the correct estimation. (a) Gaussian distributed data with $\mu = 0$ and $\sigma = 4$ over the line $y = 1.2x$. (b) Outlier artificially inserted to the data in (a). 50
4.12 Top: F0 contour example for male speaker M1. Bottom: F0 contour (blue line) example for male speaker M1 with interpolation contour (green line). 51
4.13 Histogram of voiced frequency values in the training data. . 52
4.14 Final setup of the architecture, where duration prediction and acoustic prediction work together in a pipeline fashion. 53
4.15 Ratio $\frac{\sigma_g^i}{\sigma_p^i}$ per phoneme and coefficient, where σ_g^i stands for ground-truth standard deviation at i-th coefficient, and σ_p^i stands for prediction standard deviation at i-th coefficient. There are 32 phonemes analyzed. 54

4.16 Geometric mean of $\frac{\sigma_g}{\sigma_p}$ compared to the post-filtering curves for values $pf = 1.04$, $pf = 1.05$, $pf = 1.06$. 55
4.17 Geometric mean of $\frac{\sigma_g}{\sigma_p}$ before and after post-filtering with $p = 1.04$. Green: post-filtered. Blue: raw prediction. 56
4.18 Evolution in time of the 5th cepstral coefficient for two test files. Blue: natural speech. Green: post-filtered prediction. Red: raw prediction. 56
4.19 Evolution in time of the 10th cepstral coefficient for two test files. Blue: natural speech. Green: post-filtered prediction. Red: raw prediction. 56
4.20 Evolution in time of the 15th cepstral coefficient for two test files. Blue: natural speech. Green: post-filtered prediction. Red: raw prediction. 57
4.21 Box-plot of the subjective naturalness test results relative to real human voice. SPSS: Statistical Parametric Speech Synthesis. US: Unit Selection. LSTM-raw: Two-stage TTS without post-filtering. LSTM-pf: Two-stage TTS post-filtered with $pf = 1.04$. Ahocoded: Natural speech parameterized with the Ahocoder and reconstructed. Natural: real human voice. Red line: median. Red dot: mean. 62
4.22 Activations map for the forget gates of the first 20 hidden cells in the first LSTM hidden layer. Red regions are high activations (thus preserve the past) and yellow means forget the past. 63
4.23 Activations map for the input gates of the first 20 hidden cells in the first LSTM hidden layer. Red regions mean updating the cell state a lot with the new candidate. On the other hand, yellow means not letting the information in. 64
4.24 Activations map for the output gates of the first 20 hidden cells in the first LSTM hidden layer. Red regions mean letting the cell state flow out of the cell. 64
4.25 Average activations for the first LSTM hidden layer with 512 cells. Input gate, forget gate and output gate are shown. Green dashed lines are the phoneme boundaries. 65
4.26 Averaged activations of the input gates for the first hidden LSTM layer in blue line and $1 - \mathbf{f}_t$ average activation in red. 65
4.27 Activations map for the forget gates of the 43 output LSTM cells. Red regions mean letting the cell state flow out of the cell. 66
4.28 Activations map for the input gates of the 43 output LSTM cells. Red regions mean letting the cell state flow out of the cell. 66
4.29 Activations map for the output gates of the 43 output LSTM cells. Red regions mean letting the cell state flow out of the cell. 67

4.30 Average activations for the output LSTM layer with 43 cells. Input gate, forget gate and output gate are shown. Green dashed lines are the phoneme boundaries. 67

5.1 Proposed architecture using regular feed forward (dense) layers and recurrent LSTM layers. There are N outputs belonging to N different speakers. 71

5.2 Exemplified training round for the N mini-batches. Dashed lines represent the corresponding output error back-propagation. The numbers in brackets express the order of that mini-batch inside the round. 72

5.3 Speaker adaptation system by fine-tuning a pre-trained Multi-Output model. The new layer can be trained in two ways: solid-line: 1) fine-tune only new branch with frozen model in the lower layers. 2) Fine-tune the whole model, thus propagating the error until the first hidden layer. 73

5.4 Speaker interpolation system by training a new mixing layer, the α-layer. The new layer uses the input α codes to learn to interpolate between the extreme examples given. 73

5.5 α-interpolation training method for an example with $M = 3$ for 3 batches of examples. The one-hot code expresses the identity of the currently shown speaker. Each s_m is the output prediction of the corresponding multi-output branch for the m-th speaker. 75

5.6 Training loss evolution comparison. Speakers F1 and M1 decrease the learning cost when trained with other speakers altogether. 76

5.7 Box plot of preference test scores. Scores range from -2 (multiple output model is preferred) to 2 (single output trained model is preferred). Both is the summary of all the answers, joining both speaker results. Red lines: medians. Blue dots: means. 77

5.8 Validation loss evolution comparison of different batch sizes, with frozen shared layers and fine-tuned shared layers. . . . 79

5.9 MCD when varying α values. The variation is made for speaker F1 and it is $(1-\alpha)$ for M1. All others speakers remain 0. $M = 6$. 79

5.10 F0 RMSE when varying α values. The variation is made for speaker F1 and it is $(1-\alpha)$ for M1. All others speakers remain 0. $M = 6$. 80

5.11 F0 Histograms: original M1 and F1 speakers in blue. $\alpha_{F1} = (0.25, 0.5, 0.75)$ and $\alpha_{M1} = (0.75, 0.5, 0.25)$ interpolations in green. $M = 2$. 81

5.12 F0 Histograms: original M1 and F1 speakers in blue. $\alpha_{F1} = (0.25, 0.5, 0.75)$ and $\alpha_{M1} = (0.75, 0.5, 0.25)$ interpolations in green. $M = 6$. 81

List of Tables

4.1 Context-dependent label format. 37

4.2 Number of questions per Entity. LL:Left-Left, L:Left, C:Central, R:Right, RR:Right-Right 38

4.3 Label symbol types and amount of classes per symbol. See Table 4.1 for a description of each symbol. 42

4.4 Comparison of different architectures for the duration model with their objective result. FC: Fully-Connected layer. The best performing model is in bold text. 58

4.5 Comparison of different architectures for the acoustic model with their objective results. LSTM: Hidden LSTM layer and units. Params: Number of parameters of the network. Embeddings: Number of input Fully Connected layers for first projections of the data. F0 RMSE: Root Mean Square Error of the F0. MCD: Mel Cepstral Distortion. UV Acc: Accuracy of Voiced/Unvoiced flag prediction. The best performing model is in bold text. 60

4.6 Statistics of the subjective results for the 6 systems. 62

5.1 Objective evaluation for M1 and F1 trained alone with a single output model and together with other speakers (mixed) in the multiple output architecture. 77

5.2 Objective evaluation for F3 as an adaptation subject. Full: all layers are fine-tuned. Frozen: only new output branch is fine-tuned. 78

List of Abbreviations

ANN	**A**rtificial **N**eural **N**etwork
BP	**B**ack **P**ropagation
BPTT	**B**ack **P**ropagation **T**hrough **T**ime
DBN	**D**eep **B**elief **N**etwork
DNN	**D**eep **N**eural **N**etwork
FC	**F**ully **C**onnected
FIFO	**F**irst **I**n **F**irst **O**ut
GMM	**G**aussian **M**ixture **M**odel
GPOS	**G**uess **P**art **O**f **S**peech
GPU	**G**raphics **P**rocessing **U**nit
GRU	**G**ated **R**ecurrent **U**nit
HMM	**H**idden **M**arkov **M**odel
LSTM	**L**ong **S**hort **T**erm **M**emory
MCD	**M**el **C**epstral **D**istortion
MDN	**M**ixture **D**ensity **N**etwork
MLP	**M**ulti **L**ayered **P**erceptron
MO	**M**ulti-**O**utput
MTL	**M**ulti **T**ask **L**earning
MSE	**M**ean **S**quared **E**rror
NMT	**N**eural **M**achine **T**ranslation
NN	**N**eural **N**etwork
RBM	**R**estricted **B**oltzman **M**achine
RMSE	**R**oot **M**ean **S**quared **E**rror
RNN	**R**ecurrent **N**eural **N**etwork
SGD	**S**tochastic **G**radient **D**escent
SPSS	**S**tatistical **P**arametric **S**peech **S**ynthesis
US	**U**nit **S**election
UV	**U**nvoiced **V**oiced

Chapter 1

Introduction

Speech synthesis or Text-To-Speech is the process of converting text into a voice signal. Previous to Deep Learning, existing text to speech technologies included the unit selection speech synthesis (Hunt and Black, 1996) and the statistical parametric speech synthesis (SPSS) (Zen, Tokuda, and Black, 2009). Unit selection analyzed the set of phonemes contained in a sentence and their context, and those features were mapped into pieces of recorded natural speech, all being concatenated to produce a continuous stream of voice signal. SPSS introduced the concept of learning a speaker model from data with parametric representations and then throw away the data once speaker characteristics were learned. Some remarkable differences between both was that, although SPSS could not reproduce the same level of naturalness (Zen, Tokuda, and Black, 2009) as unit selection did, it had much less footprint in memory, and it also let the user transform any speaker model to adapt the voice to different requirements in speed, intonation, etc. Two important techniques developed within the framework of SPSS were the speaker adaptation and the speaker interpolation. In the former case we could add the voice of someone that was not previously in the system for whom we have a low amount of data, and extracting characteristics from the other speaker models it could reach a good level of similarity to that of the original speaker. In the case of interpolation, a new voice model could be build from scratch without any data for a new speaker by combining the existing speaker models available in the system.

Deep Learning has been applied successfully to different kinds of tasks such as computer vision, natural language processing or speech processing (Deng and Yu, 2014), outperforming the existing systems in many cases. In the case of speech synthesis, many works included Deep Neural Networks and Deep Belief Networks to perform acoustic mappings in the SPSS framework and prosody prediction. Also, Recurrent Neural Networks and their variants, like the Long Short Term Memory architecture, have leveraged completely the sequences processing and prediction problem, which makes them lead to interesting results in the speech synthesis field, where an acoustic signal of variable length has to be generated out of a set of textual entities.

Despite Deep Learning TTS has improved the speech quality generated by HMM-based SPSS, it lost some of the flexibility offered by these models. First, the research in speaker adaptation for the DNN/RNN approach is very recent and not many techniques are proposed in comparison to the SPSS. Moreover and to the best of our knowledge, there are no works exploring the speaker interpolation capabilities of these systems, whilst this

was a remarkable feature in the SPSS.

The first goal of this work is to build a state of the art TTS in UPC with Deep Learning techniques and analyze the insights of the developed architecture. After this, we want to achieve new flexible models capable of doing speaker adaptation and interpolation without sacrificing the generated speech quality, in addition to represent many speaker models inside the same structure.

The structure of this work is the following: Chapter 2 begins exploring the state of the art techniques for speech synthesis, from Unit Selection systems to the latest Deep Learning TTS architectures and methods. Then in Chapter 3 there is an introduction to the Deep Learning topic, its elements, techniques and terminology. It is a guide to follow what comes in Chapter 4, which is the research and development of the TTS system made from scratch with RNN-LSTM. Afterwards, in Chapter 5 a proposed multiple speaker TTS is shown and analyzed in detail, as well as the proposed adaptation and interpolation methods built on top of it. Finally, the conclusions, future work and research contributions can be found in Chapter 6.

Chapter 2

State of the Art

Speech synthesis, also known as Text To Speech, is the technique with which computers can speak. These systems have gone through a great evolution during the past two decades, and in this chapter some currently existing systems are introduced with their most used techniques. First, we review the most used in comercial environments which are the so called Unit Selection systems. Secondly, the Statistical Parametric Speech Synthesis is seen, which has leveraged the speech synthesis research during the last decade. The chapter then concludes presenting the state of the art techniques of Deep Learning applied to speech synthesis.

2.1 Unit Selection Speech Synthesis

This type of synthesis has been operative during many years because it offers the best naturalness level, as it is based on real recorded speech (Hunt and Black, 1996). The way in which this system works is by concatenating segments of speech, which are usually the so called diphone. A diphone is a voice unit of the same size as a phoneme, defined in between of two phonemes (i.e. from the middle of a phoneme to the middle of the next one). Figure 2.1 exemplifies some hypothetic diphone boundaries compared to those of phonemes. The reason to do the division at the middle point of the phoneme is because it is the more stable point and the one least influenced by the co-articulation, which is the influence of neighboring phonemes to the current one.

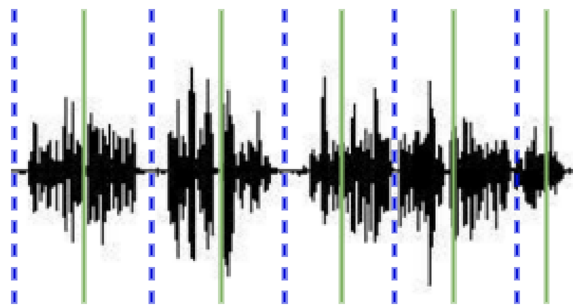

FIGURE 2.1: Voice stream where blue dashed lines show hypothetic phoneme divisions and green lines show hypothetic diphoneme divisions.

During concatenation to make the speech reconstruction the following issues need to be considered: discontinuities between the speech segments in

phase, pitch, etc. and differences in prosody, which means that the recorded segments may vary in duration or pitch (thus expressiveness) with respect to the targeted prosody that should be achieved. To cope with these there are two approximations:

- Process the signal to smooth the discontinuities and force the prosody to match.
- Use a large database with many repetitions of the contained diphonemes, such that there is more variability to adapt to more possible contexts and in reconstruction the chosen one is that matching better the prosodic requirements.

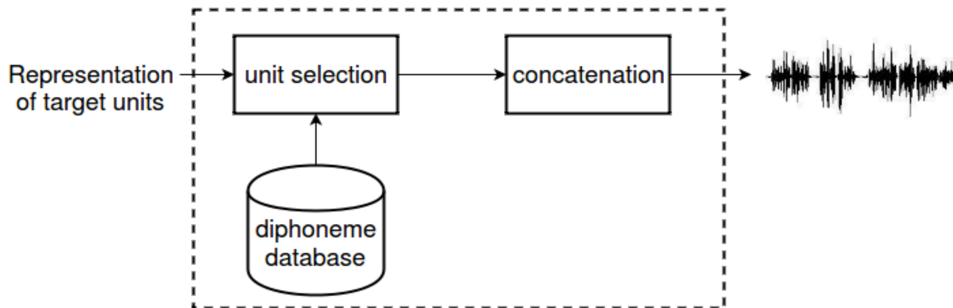

FIGURE 2.2: Unit selection scheme.

The general unit selection concatenative system is shown in Figure 2.2. Depending on the system and the dataset available there might be post-processing in the concatenation block shown or not, as mentioned previously. The way in which the units are selected is by maximizing the prosodic match (or minimizing the mismatch) and, at the same time, minimizing the discontinuities that can appear during the concatenation of the speech units. To do so, a cost function is defined to achieve these objectives, so that the units with the best scores are the ones selected, as shown in equation 2.1.

$$\hat{\mathbf{u}}_1^n = \underset{\mathbf{u}_1^n}{\operatorname{argmin}}(\rho \sum_{i=1}^{n} c_{target}(u_i, t_i) + (1-\rho) \sum_{i=1}^{n-1} c_{concat}(u_i, u_{i+1})) \tag{2.1}$$

Where c_{concat} stands for the concatenation cost between consecutive units and c_{target} is the target cost, which considers the worthiness of getting a certain unit to achieve the desired prosodic target. Also, we can set some weight ρ which lets us give more importance to one criterion or the other, depending on our requirements (e.g. we may worry only about the prosody, not considering the concatenation cost).

2.2 Statistical Parametric Speech Synthesis

The aim of this section is to make a brief review to what is the Statistical Parametric Speech Synthesis (SPSS). There are good works reviewing and

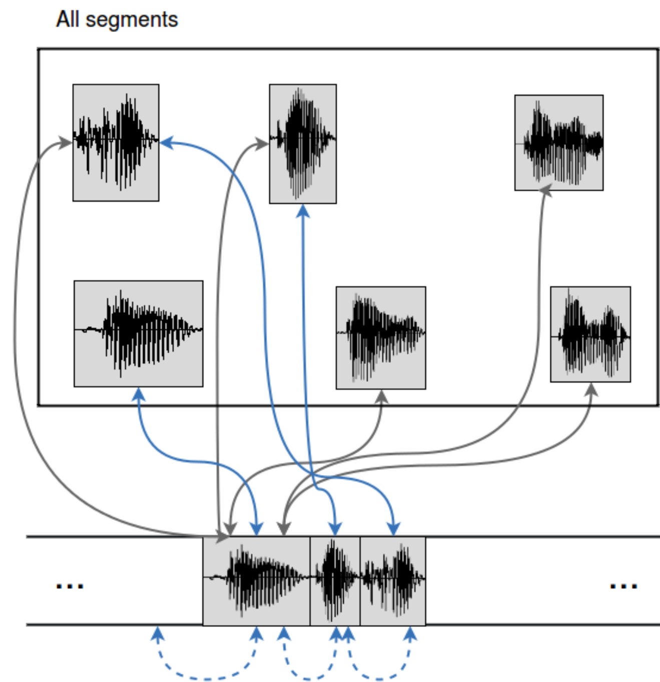

FIGURE 2.3: Unit selection synthesis process. Dashed lines represent the concatenation cost. Solid lines represent the target cost. Grey lines depict the computation performed against all units. Blue lines are the computation of the actual used units. Based on Zen, Tokuda, and Black (2009) Figure 1.

gathering information about the SPSS systems in Zen, Tokuda, and Black (2009) and Ling et al. (2015). The following description is based on Ling et al. (2015) work, as it also goes over the Deep Learning state of the art in speech synthesis, something that follows in the next section.
In this TTS approach, a set of stochastic generative acoustic models relate text derived features to acoustic frames. In order to do so the speech signal is represented in a parameterized way, meaning that it gets encoded in a vocoder stage that creates an acoustic feature vector in a windowed fashion shifted every 5ms. The acoustic parameters of each phoneme in a given phonetic and prosodic context are represented by a stochastic generative model. Concretely context-dependent phoneme Hidden Markov Models (HMM) with single Gaussian state-output Probability Density Functions (PDF) are used. The phoneme contexts are defined using a decision-tree that clusters similar HMM output PDFs attending to the phonetic and prosodic features.

An HMM is a statistical Markov model with unobserved (hidden) states, which means that the state of every Markov state is not directly seen by the observer, but the outputs generated by this state (generated by means of the state-output PDF aforementioned). The state transitions work the same way as in a Markov chain, with state-transition probabilities, which makes them suitable to model sequences. Figure 2.4 depicts and example of a 3-state HMM.

Figure 2.5 is a schematic of the SPSS framework, where we can separate two stages: training and synthesis. During training, acoustic features of speech are extracted from the speech waveforms contained in a training

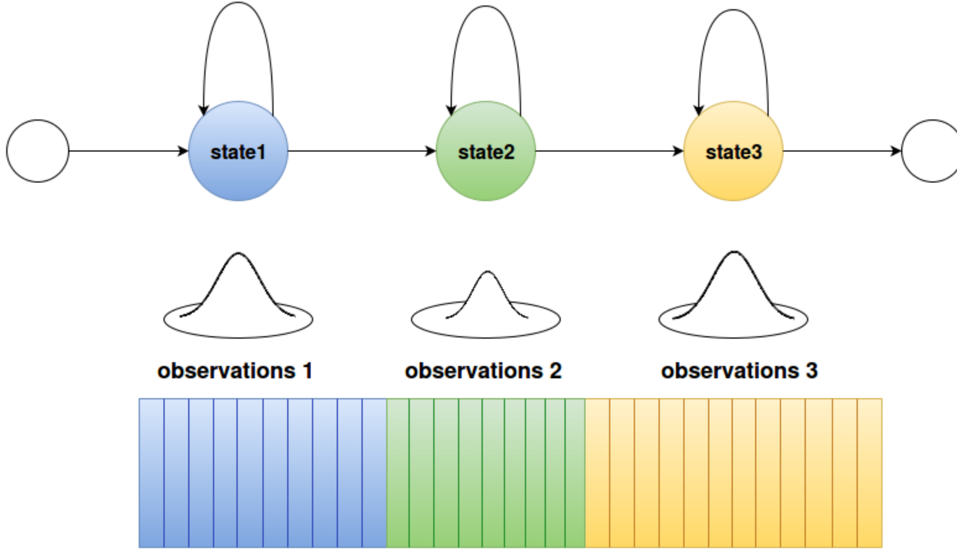

FIGURE 2.4: Hidden Markov Model example with 3 states.
Based on Ling et al. (2015), FIG1.

data set (i.e. vocal tract and vocal source parameters). Context features are also extracted from the text transcriptions to build what are called the *labels*. Once we have the features, the context-dependent HMMs (λ^*) are estimated based on the Maximum Likelihood criterion:

$$\lambda^* = \underset{\lambda}{\operatorname{argmax}}\, p(\mathbf{y}|x, \lambda) \tag{2.2}$$

where $p(.)$ is a continuous PDF, $\mathbf{y} = \{y_1, y_2, \cdots, y_T\}$ is a sequence of acoustic features with T frames, being y_t the acoustic frame at time t. Finally, $\mathbf{x} = \{x_1, \cdots, x_N\}$ is a sequence of linguistic context features, with N the number of phonemes. The acoustic feature vector is normally composed of static components and their first and second derivatives, such that:

$$\mathbf{y} = \{\mathbf{y}_{st}, \Delta\mathbf{y}_{st}, \Delta^2\mathbf{y}_{st}\} \tag{2.3}$$

Where Δ and Δ^2 stand for first and second derivatives. The complete acoustic feature set at time t can then be considered as a linear transform over the static feature sequence $\mathbf{y}_s = \{\mathbf{y}_{s1}, \mathbf{y}_{s2}, \cdots, \mathbf{y}_{sT}\}$:

$$\mathbf{y} = \mathbf{M}_y \mathbf{y}_s \tag{2.4}$$

$\mathbf{M}_y$ is determined by the expression to compute first and second derivatives used in equation 2.3.
Usually there are too many context-dependent HMMs to take into account given the large set of possible fine-grained linguistic contexts to consider in comparison to Automatic Speech Recognition (ASR) systems. This large amount of models can lead to over.fitting the context-dependent models that have few examples to be trained on, and also some valid combinations of linguistic contexts will be missed in the training set. To deal with this, there is a decision-tree based clustering method applied after the initial

training in order to cluster state-output PDFs of the HMMs. The process in shown in Figure 2.6, where the PDFs of similar context descriptions are all represented by a shared distribution. This clustering is achieved by means of the a set of questions specifically designed considering the characteristics of the processed language. After this, the state alignment results using the HMMs are used to train context-dependent state-duration Gaussian distributions, one per model. Also, a clustering process is used for these duration models (54). The acoustic model $p(\mathbf{y}|x, \lambda)$ can then be rewritten like so:

$$
\begin{aligned}
p(\mathbf{y}|x, \lambda) &= \sum_{\forall q} p(\mathbf{y}, \mathbf{q}|x, \lambda) \\
&= \sum_{\forall q} P(\mathbf{q}|x, \lambda) p(\mathbf{y}|\mathbf{q}, \lambda) \\
&= \sum_{\forall q} P(\mathbf{q}|x, \lambda) \prod_{t=1}^{T} p(\mathbf{y}_t|q_t, \lambda)
\end{aligned} \tag{2.5}
$$

Where $P(.)$ denotes a probability mass function and $p(\mathbf{y}_t|q_t, \lambda)$ is a state-output PDF associated to the q_t state (typically a single Gaussian distribution with diagonal covariance matrix). Finally, $\mathbf{q} = \{q_1 \cdots q_T\}$ is an HMM state sequence. As we have mentioned previously the state duration probability $P(\mathbf{q}|x, \lambda)$ is modeled using a context-dependent Gaussian distribution for each state. Note that equations 2.5 are based on the assumption that the frame observations are independent from each other in the state sequence.

During synthesis we first get the same context features out of the linguistic front-end analysis, then we get $\hat{x}$. The features are injected into the acoustic parameter generation process, where we want to maximize the output probabilities of a certain acoustic feature given the sentence HMM.

$$
\mathbf{y}_s^* = \operatorname*{argmax}_{\mathbf{y}_s} p(\mathbf{y}|\hat{x}, \lambda^*)\Big|_{\mathbf{y}=\mathbf{M}_y \mathbf{y}_s} \tag{2.6}
$$

To conclude this brief introduction to SPSS it important to mention these systems have a great advantage in flexibility in comparison to unit selection systems. This means that with mathematical transformations we can change the speaker characteristics (duration of phonemes, expressiveness, identity, etc.). A key method taking advantage of SPSS flexibility is speaker adaptation, where a new speaker can be introduced into the system even if he/she has a low amount of data to train: we can take information from surrounding speaker models to fill the "missing spots" of the new speaker. Furthermore, new speakers can be created out of the existing models without any new data, by means of the so called speaker interpolation technique. Also, the footprint in memory is lower for SPSS compared to unit selection due to the fact that once the models are trained, all the voice data can be left behind and only the models (reduced representations of the real speakers' data) are kept. Even though, the main drawback of these systems is the quality degradation, which comes mainly from three components:

- the vocoder: parameterizing the speech provokes some loss of quality.

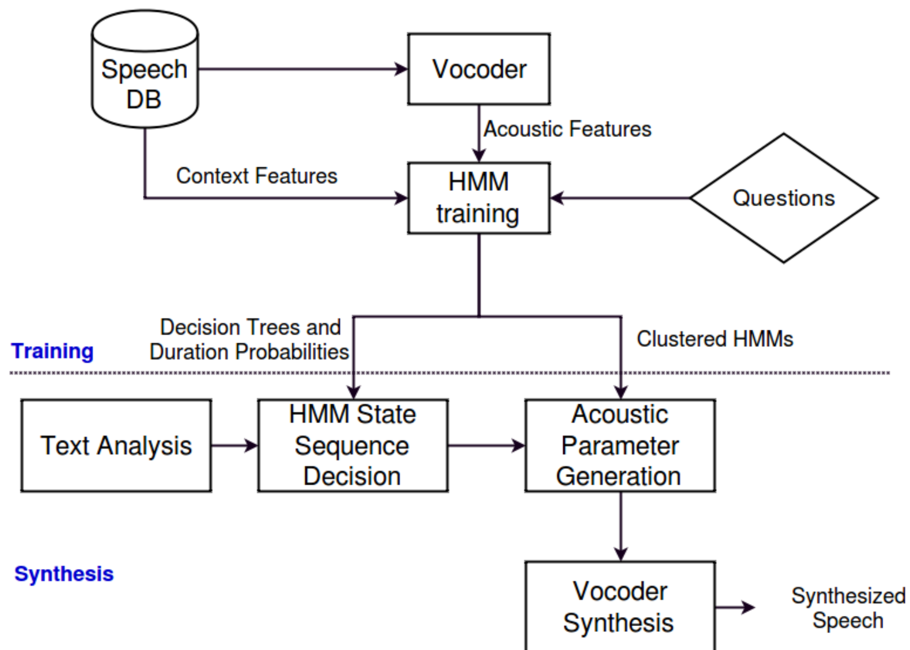

FIGURE 2.5: Block diagram of typical Statistical Parametric Speech Synthesis HMM-based systems. Based on Ling et al. (2015), FIG2.

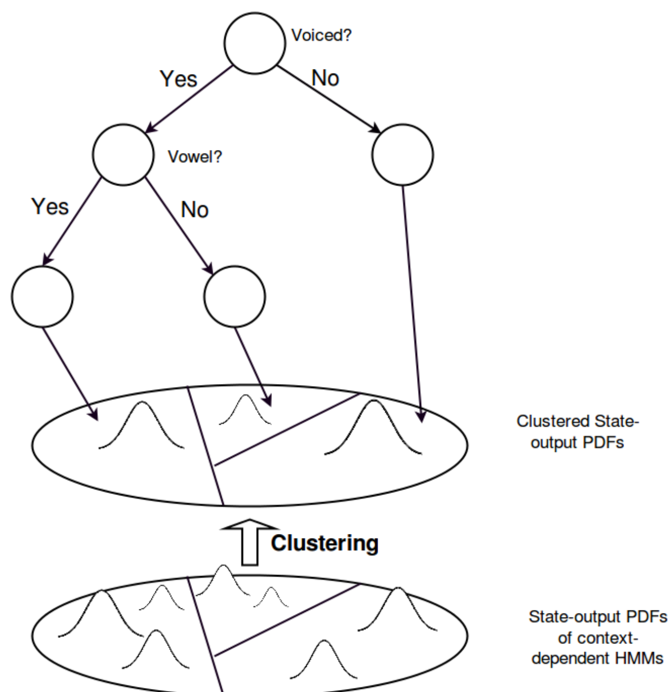

FIGURE 2.6: Hidden Markov Model clustering process. Based on Ling et al. (2015), FIG3.

- the acoustic model: the errors inherent to the prediction.
- the over-smoothing effect: coming from a lack of representational power of the model.

2.3 Deep Learning in Speech Synthesis

Now that the most recently developed systems have been briefly reviewed we can get a better look at the state of the art in deep learning methodologies for speech synthesis. First it is worth mentioning that Deep Learning can be interpreted as another type of Statistical Parametric Speech Synthesis, because the Neural Networks can be interpreted as statistical models. However, from now on we may refer to HMM-based synthesis by SPSS, and to the applied Deep Learning synthesis as DNN synthesis or LSTM synthesis (this is meaningful for when we reach the system developed in this work, where LSTM keyword will be understood). In Chapter 3 there is a review of deep learning techniques and the type of architectures existing at the moment, which includes explanations of what is a Deep Neural Network (DNN) and a Recurrent Neural Network (RNN), and it also contains references to DBNs (which turn out to be a pre-initialized version of the DNNs). Unfamiliar readers may prefer reading this section after Chapter 3 to have a clearer picture of the following references.

The use of Deep Learning in TTS is very recent, and in the early works the feed forward Neural Networks were used as acoustic models for SPSS, like in Ze, Senior, and Schuster (2013) where a DNN is used in the acoustic mapping process. In the case of SPSS the acoustic mapping is the generation of acoustic parameters out of the Gaussian mean from the proper cluster (see section 2.2). Here, the decision tree with the Gaussian distribution is substituted by a DNN emitting the predictions. It is with the DNNs that they obtained a better modeling of the complex context dependencies, outperforming the more classic approach of decision trees.

Similarly, in Lu, King, and Watts (2013) the authors combined a vector-space representation of linguistic context with a DNN that directly adapted such continuous representations to make the acoustic mapping. In Qian et al. (2014) the authors also worked with a DNN to perform the acoustic mapping from linguistic inputs, making a study of the different parameters affecting better convergence schemes for the TTS task and using a moderate sized corpus. They proved how their scheme outperformed the conventional HMM, and they found out that the main improvement came from the prosody prediction (concretely with F0 contour). Hu et al. (2015) used a dynamic sinusoidal model (DSM) (Hu et al., 2014) in a DNN-based acoustic mapping model with multi-task learning. They first model cepstra for spectral parametrization of the DSM and then the log-amplitudes as a direct parametrization of the DSM. During synthesis they did not discard the second task but fused it with the first one, getting an improved performance with respect to simply using the prediction from the cepstral parameters. Furthermore, Valentini-Botinhao, Wu, and King (2015) made an acoustic mapping with DNNs, training the models with a perceptually-oriented cost function. This cost function was defined in a domain different than

the one for the predictions of the DNN. Given this approach, their objective results indicated that the perceptual domain system achieved the best quality. Kang, Qian, and Meng (2013) followed a similar approach with a Deep Belief Network (DBN) generative model to represent the dependencies between linguistic and acoustic features. They also showed how with these RBM-DBN systems outperformed the classical HMM approach used in SPSS with objective and subjective results.

In Zen and Senior (2014) the authors claim that the previous approaches where a DNN was used to model the acoustic mapping had some limitations, and they addressed the following ones with a Deep Mixture Density Network (MDN): First, the regular DNNs do not have the power to model distributions of outputs that are more complex that a unimodal Gaussian distribution. Secondly, The outputs of an ANN only provide the mean values, whilst variance has been proven to be an important property to achieve good naturalness in speech synthesis. Their MDN then provides a set of outputs that model Gaussian Mixture Models of every output acoustic parameter, with means and variances. Their results, both objective and subjective, reflect that their model can relax the limitations in the DNN-based acoustic modeling. Similarly, the work in Uria et al. (2015) use a Real-valued Neural Autoregressive Density Estimator (RNADE) (Uria, Murray, and Larochelle, 2013), which is a similar approach to that of MDN. RNADE also uses a neural network to predict a distribution of acoustic features conditioned on a set of phonetic labels by outputting the parameters of a real-valued distribution. The main difference with MDN however is the fact that RNADE predicts each dimension within an acoustic frame sequentially, thus the values of features already predicted are also input to the network, which allows RNADE to capture dependencies between the different acoustic features in a frame. In Wu et al. (2015b) the authors address two problems for the way in which DNNs are applied to speech synthesis: Perceptual sub-optimality and frame-by-frame independence. They claim that the first problem comes from the training criterion, which typically aims to maximize the likelihood of acoustic features which are a rather poor representation of human speech perception. Besides, the error in the speech feature space is not an accurate reflection of the expected perceptual error, and they propose a Multi-Task Learning (MTL) procedure to get around it, where the DNN learns to predict a perceptual representation of the target speech as a secondary task besides predicting the typical invertible vocoder parameters as the main task. These secondary task predictions are discarded during synthesis, such that they only serve as "hints" during the main task training. Regarding the second problem they refer to Recurrent Neural Networks as effective models to treat with sequential data, but they are difficult to optimize as well as computationally expensive. Their solution is simpler than any recurrent topology by means of a technique called bottleneck feature stacking, where they train a first DNN with a bottleneck hidden layer (a hidden layer with a smaller set of units than the ones preceding it). Afterwards they take the activations of the bottleneck (which yield a compact representation of both acoustic and linguistic information for each frame) of many contiguous frames (e.g. h_{t-1}, h_t, h_{t+1}). These activations are then stacked together and joint with the linguistic features to

get into another DNN stage that makes the acoustic mapping, thus considering the dependencies between frames in this case. In Wu and King (2015) they take advantage of the stacked bottleneck activations to have a wide linguistic context with a training criterion that minimizes the speech parameter trajectory errors taking dynamic constraints from a wide acoustic context. This way they minimize the utterance-level trajectory error instead of the frame-by-frame error, and they achieve better naturalness than previous approaches with the proposed training criterion. In Fan et al. (2015) they propose a model to hold many speakers out of the same shared DNN structure with a specific training mechanism where they back-propagate all the speakers information in the same mini-batch. They achieve better results with the multitask approach than learning a single speaker parameters isolated. They further transfer the learning of the base shared structure for a new speaker to achieve speaker adaptation with limited training data, achieving good results in naturalness and similarity to the original speaker. In Wu et al. (2015a) they perform DNN speaker adaptation with three types of techniques: adding identity information to the input features, Learning Hidden Unit Contribution (LHUC) (Swietojanski and Renals, 2014), and making output feature space transformations.

On the other hand, RNNs and their variants, like the LSTM architecture introduced in Hochreiter and Schmidhuber (1997) (see Chapter 3, sections 3.4 and 3.6), have leveraged completely the sequences processing and prediction problem, which makes them lead to interesting results in the speech synthesis field, where an acoustic signal of variable length has to be generated out of a set of textual entities. Regarding RNNs in Chen, Hwang, and Wang (1998) they explored the usage of standard RNN architectures with many hidden layers synchronized by different timings (at syllable level and at word level) for prosodic parameter prediction, such as syllable pitch contours, syllable energy levels, syllable initial and final durations, as well as intersyllable pause durations. In Achanta, Godambe, and Gangashetty (2015) they investigate two variants of RNNs applied to acoustic parameter generation: Elman-RNN and Clockwork-RNN. They show that Clockwork is equivalent to an Elman-RNN with a particular form of Leaky Integration (LI) (Bengio, Boulanger-Lewandowski, and Pascanu, 2013).
Even though, the most widely used RNN in speech processing applications is the LSTM aforementioned. In Fernandez et al. (2014) they use a bidirectional LSTM architecture to predict F0 contours. In Zen and Sak (2015) they employed a unidirectional LSTM architecture to make a low-latency speech generation model. An interesting result is that using a recurrent output layer they obtained better results than making dynamic parameters prediction and deriving the trajectory using Tokuda's algorithm (Tokuda et al., 2000).
In Wu and King (2016) the authors explore the effectiveness of LSTM architectures for speech generation purposes. Concretely, they explore the necessity of the different gating mechanisms (see section 3.6) and come up with a more efficient solution that only requires the forget gate and the input gate (which in turn is the inverse of the forget gate value and does not need parameters to be learned). Finally, in Coto-Jiménez and Goddard-Close (2016) they proposed a post-filtering methodology to be carried out in SPSS where an LSTM learns its parameters to perform enhancement of

the predicted speech in order to be closer to a natural voice than what is obtained with HMM synthesis.

2.4 Summary

In this chapter the different currently-in-use speech synthesis systems have been reviewed, where Statistical Parametric Speech Synthesis and Deep Learning Speech Synthesis are the ones under more research nowadays, whilst Unit Selection systems are the ones raising the highest-quality voice, achieving more naturalness owing to the fact that they treat with natural voice directly rather than generating acoustic trajectories with statistical models. Regarding SPSS there are explanations about the training procedure as well as the synthesis one, briefly explaining the concepts of states clustering with decision trees and linguistic contexts, which will be useful to understand the contents in Chapter 4. Finally, the set of state of the art Deep Learning techniques applied to the speech generation problem have been shown, which contain many acoustic mapping mechanisms by means of DNN and DBN architectures. Also, speaker adaptation and multitask methods are shown which raise better results than those with direct acoustic mappings. RNN architectures are also reviewed, and more concretely advanced variants like LSTM extensions, which achieve state of the art results thanks to their implicit temporal-dependency modeling of sequences.

Chapter 3

Introduction to Deep Learning

Deep Learning is an area of Machine Learning that has been recently very explored, composed of a set of tools and techniques that let powerful models learn complex patterns automatically from data. This learning process includes the multiple transformations of the data within the model itself to get from low level processing functions to more abstract ones. This means that, when we inject information for a specific task, the model makes the data flow through many stages, extracting different sub-types of information that interact and produce the final prediction. We will see some examples of these levels decomposition, picturing some internal structures of the learned models.

The models we use in this framework are the Artificial Neural Networks (ANN), and the number of transformation stages, called layers, are what conform the depth of these networks. To be more specific, lets review what an Artificial Neural Network is and what types of layer do compose it, and later some other types of ANN that get us closer to the specific tasks of this thesis will be seen. From now on we may refer to an ANN as a Neural Network (NN).

3.1 Artificial Neural Network

An Artificial Neural Network is a composition of elementary units called neurons. First, we can see what a neuron is and what computation does it perform and later go to the stacking process to obtain the full network. A neuron, then, is a basic computation unit and also an analogy of what a biological neuron is. A schematic of it is depicted in Figure 3.1. There we can see some operations performed to get an output scalar y out of an input vector $\mathbf{x} = \{x_1, x_2, x_3\}$.

Equation 3.1 is the operation made in this unit. There we can see that, when the input vector $\mathbf{x}$ is injected into the neuron inputs, they are multiplied by a set of weights (arranged as a vector $\mathbf{w}$), such that each input link has a weight, and then we sum up these products together with a bias term.

Finally, the result of this addition of products is passed through a function f shown in Equation 3.2, which can be of many types. To emulate the biological neurons, which fire an electrical impulse or not depending on a

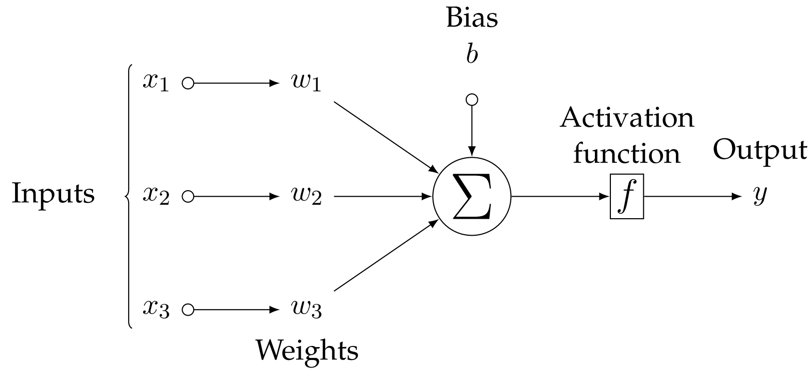

FIGURE 3.1: Artificial Neuron

threshold on the input sum, it is usually exemplified with the *Sigmoid* function $\sigma(x)$, shown in Figure 3.2 in the scalar case. This unit is also called the perceptron.

$$a = (\mathbf{w}^T \cdot \mathbf{x} + b) \tag{3.1}$$

$$y = f(a) \tag{3.2}$$

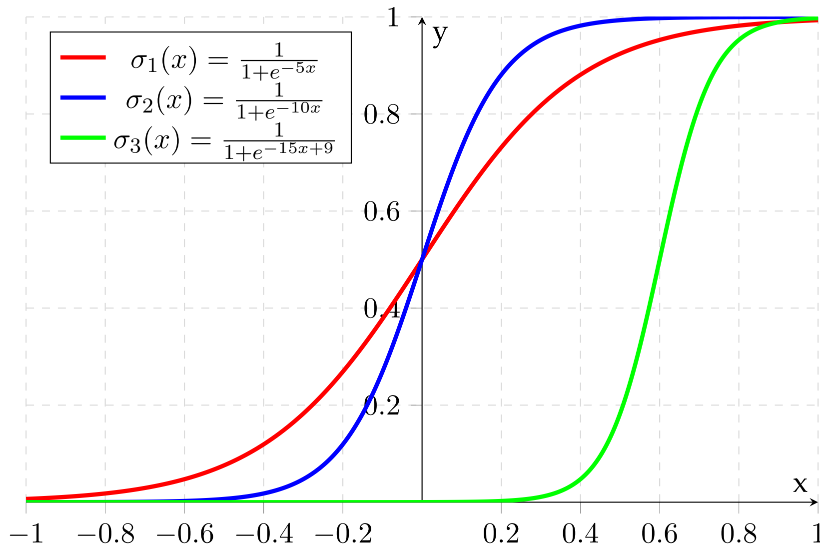

FIGURE 3.2: Sigmoid function with different scalar weightings and bias: red $w = 5$, $b = 0$; blue $w = 10$, $b = 0$; green $w = 15$, $b = 9$.

Equation 3.3 shows the $\sigma(\mathbf{x})$ function's form. This is a thresholding function, such that when the sum in 3.1 goes beyond the saturation point, the output is constantly 1, and the same happens in the negative region where the output goes to 0. We could obtain the same behavior with a step function, but an important feature of $\sigma(\mathbf{x})$ is that it is differentiable, something required for the learning process as we will see later in this chapter.

$$\sigma(\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^T \cdot \mathbf{x} + b}} \tag{3.3}$$

Now that the artificial neuron has been introduced along with the operation it performs, an explanation of what this means geometrically will further help to interpret what happens in the stacking process of these elements to form a Neural Network. The equation 3.1 is the expression of a hyper-plane, where the set of weights $\mathbf{w} = \{w_1, w_2, \cdots, w_N\}$ control the rotation and skew of it, and the bias term b controls its translation from the origin. This is then a linear operation dividing the hyper-space S into two-splits, and the application of the *Sigmoid* function lets us interpret the regions in those splits as probabilities of pertaining to a class conditioned on the input.

The neuron's operation after the $\sigma(x)$ is called the Logistic Regression, a technique to perform binary classification tasks modeling the posterior probabilities with the hyper-plane equation we have seen earlier (Hastie et al., 2005). So if we interpret the $\sigma(x)$ as being the probability, as mentioned earlier:

$$P(Y=1|X) = \frac{1}{1+e^{-\mathbf{w}^T\cdot\mathbf{x}+b}} = \frac{e^{\mathbf{w}^T\cdot\mathbf{x}+b}}{1+e^{\mathbf{w}^T\cdot\mathbf{x}+b}} \tag{3.4}$$

This brings the idea of the *logit*, which is the inverse of the logistic function and expresses that this probability is derived from a linear regression of the boundary between the classes, which is in turn the neuron operation.

$$\text{logit}(P(Y=1|X)) = \log(\frac{P(Y=1|X)}{1-P(Y=1|X)}) = \mathbf{w}^T\cdot\mathbf{x}+b \tag{3.5}$$

Equation 3.5 is a Linear Regression (Hastie et al., 2005), and it lets us approximate functions depending on the $\mathbf{w} = \{w_1, w_2, \cdots, w_N\}$ predictors (e.g. Price of a house based on its location, year of construction and area). So this shows how the neuron first estimates an approximating linear function to build a hyper-plane, and then the element-wise *Sigmoid* turns the estimation into a posterior probability.

The following explanation will be focused on classification tasks, but it is a natural extension to an N-dimensional regression as well. The neuron is the beginning of what a neural network will be, and now we will see the limitations of the single perceptron in order to get to a more complex model.

First, the perceptron only lets us classify in a binary fashion, because there is only one neuron to fire an output. Second, it can only work with a single hyper-plane to discriminate highly complex patterns.

The natural extension to overcome the binary limitation will be to put many neurons in parallel, each processing its binary output y_n from a set of inputs: $\mathbf{x} = \{x_1, x_2, \cdots, x_M\}$. When dealing with classification all the binary contributions are normalized to sum up to one, such that we obtain a probability distribution out of the N output neurons. The output activation becomes the one in equation 3.6, and its name is the *Softmax* function.

$$P(y=k|\mathbf{x}) = \frac{\exp \mathbf{x}^T\mathbf{w}_k}{\sum_{n=1}^{N}\exp \mathbf{x}^T\mathbf{w}_n} \tag{3.6}$$

Where k stands for the k-class and N is the total output neurons as previously stated. This topology is depicted in Figure 3.5. For convention, the *Input Layer* is drawn like a set of M neurons, but they are not real neurons in the sense of computation units, just each input x_m.

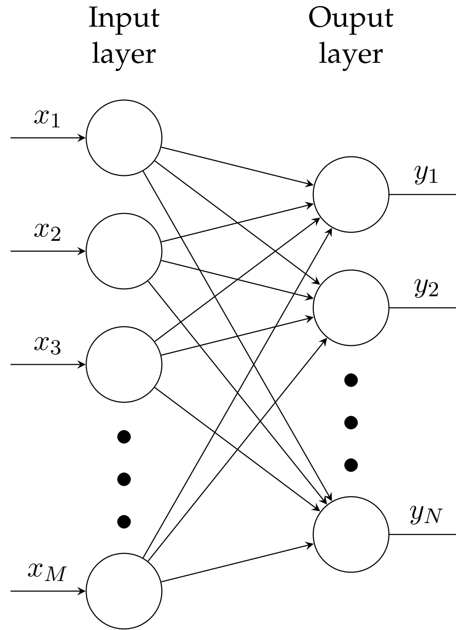

FIGURE 3.3: Logistic Regression of N outputs.

This is letting us make a many-to-many mapping operation:

$$\mathbb{R}^N \rightarrow \mathbb{R}^M$$

Now the other limitation was the computational power of this system. This can be seen with the XOR example, which is frequently used to illustrate the shortcomings of the Logistic Regression with respect to the Neural Network. This is again a binary classification problem (i.e. single perceptron). The neuron has to learn the XOR operation for two inputs, such that we have:

$$\mathbf{X} = \{(0,0),(0,1),(1,0),(1,1)\}$$

Note the **X** notation meaning we can arrange the values as a matrix of values, where we have 4 rows and 2 columns.

$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Each row is called an *input sample* to our model, and we want the following predictions:

$$\mathbf{Y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Where we have 1 prediction per input sample. So in this example $M = 2$ and $N = 1$, following our previously established notation. If we try to plot this in a $3 - D$ space, we would get what is depicted in Figure 3.4.

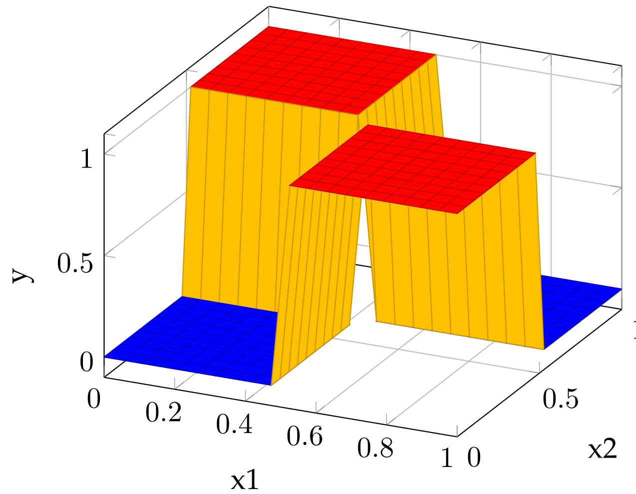

FIGURE 3.4: XOR resulting approximate surface.

From Figure 3.4 we can intuitively get to the conclusion that a single neuron could separate one of the two four existing regions (two regions with 0 and two regions with 1). Thus we need to add at least the capacity to define two discriminating units, which brings the idea of the *Hidden Layer*, an intermediate set of neurons that will first map the input space to a linearly separable representation where the final decision will be taken (either classification or regression). In this case, the following Neural Network has the weights trained to make the XOR function:

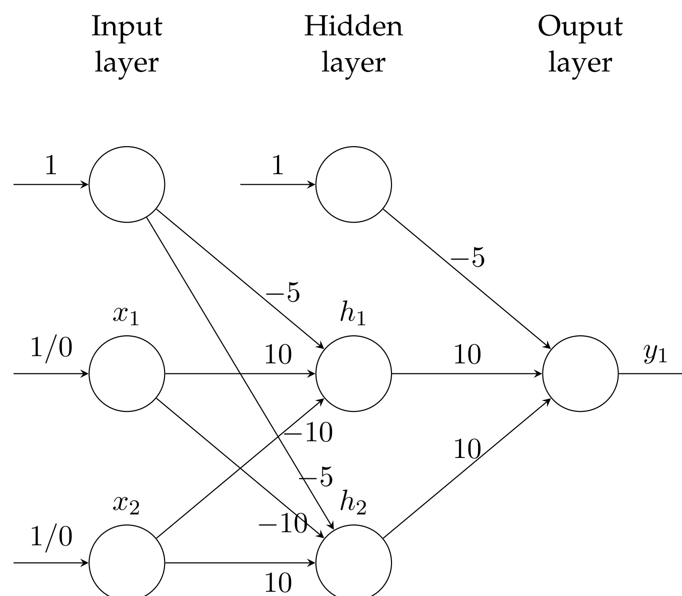

FIGURE 3.5: XOR Neural Network with Sigmoid perceptrons.

This is the so called Artificial Neural Network, a set of artificial neurons linked together in order to perform an arbitrary function, as in this example where the XOR is implemented with the depicted trained weights. An important and hard task for these systems is the learning process, that will be reviewed later in this chapter. The learning process is what makes the

network set its weights to perform the desired function. Another important thing to point out is that connections between layers are depicted with arrows, which indicates that this model is directed, and this is why this is also called a feed-forward Neural Network, thus making the inputs flow up to the output layer in what we call the inference operation. To inspect a little bit the final components and equations of this system, we see that in this XOR toy example the inputs are x_1 and x_2, which can have binary values 0 or 1, and there is one output y_1, binary as well. The unlabeled neurons are the biases, drawn here as inputs in every layer such that the bias values can be appreciated together with the weights. This is incorporated in the equation 3.1 by appending the scalar value 1 as an input in $\mathbf{x}$, and letting the value $w_{i0} = b_i$ in the set of weights on every neuron.

So now, as we have many neurons in every layer, the set of all links from layer i to layer j can be written as a matrix $\mathbf{W}_{ji}$, where each w_{ji} is the link weight between neuron/input in layer i and neuron in layer j, which turns to be the layer $i+1$. Also, I is the number of neurons/inputs in layer i, and J is the number of neurons in layer j, leading to a matrix $\mathbf{W}_{J\times I}$. Now, the computation of a layer are called activations (e.g. for the hidden layer in Figure 3.4) and the operation can be written as:

$$z = \sigma(\mathbf{W}_{ji} \cdot \mathbf{x}) \quad \text{where} \quad \mathbf{x} = \{1, x_1, \cdots, x_N\} \tag{3.7}$$

Later, when going in depth with many architectures, we may refer to a feed-forward layer as a hidden or output layer composed of this directed connections and activation functions (such as the Sigmoid). Yet another plausible name will be *Fully Connected*(FC) layer. Depending on the type of input/output values the sigmoidal units may implement the $\tanh$ as a non-linearity instead of the Sigmoid, due to its extended range between $\{-1, 1\}$. Nonetheless, in regression tasks the output might be directly the operation in equation 3.1 for all neurons in the output layer, such that the layer operation is purely linear and returns a real value. Regression also works whilst predictions are made within the linear region of the function with normalized outputs. Also, in terms of notation there is a variant name for the whole feed-forward architecture which might be used in this work as well, the Multiple Layer Perceptron (MLP).

3.2 Training the network: Back-Propagation algorithm

In this section it is discussed the way in which the network is trained, thus the way in which it learns the weight value of every link and the bias value of every neuron. The book of Bishop (1995) is a very good reference for understanding this methodology, and many parts of this section will be based on it. Also the work from LeCun et al. (2012) is a thorough analysis of good practices to make this algorithm work well, but it is out of the scope of this work to discuss every aspect with the tricks and tweaks of the learning procedure.

The learning process is based on a cost function (also called error function) that we define depending on the problem we are facing, whether it is classification or regression. Given this cost the network's task is to minimize it

with respect to its weights and biases, so the cost function must be differentiable. The algorithm to evaluate the derivatives for the cost function of our choice is called the Back-Propagation (BP) algorithm, and it is based on the propagation of errors backwards through the network structure, from the output layer to the input layer, in order to correct the weights that provoke these mistakes with small modifications proportional to the committed errors. It is important to recall that there can be many non-linearities happening layer by layer at this point (see equation 3.7), and this provokes the lack of a closed solution for estimating the parameters. This algorithm then behaves in an iterative manner, such that it learns step-by-step for certain inputs shown many times to the model. As posed in Bishop (1995), each step can be divided in two stages:

1. First compute the derivatives from the error and back-propagate them.
2. Secondly update the weights of each layer.

It is worth mentioning that the second stage can be implemented in a variety of ways, and some reference to modern techniques will be given later in this section.

Now it is good to have a more in-depth analysis of the BP algorithm for a general network, with an arbitrary amount of feed forward layers, with an arbitrary non-linearity and with an arbitrary error function. In a general feed-forward neural network, each unit computes its activation like shown in equation 3.7 previous to applying σ, which can be rewritten as:

$$a_j = \sum_i w_{ji} z_i \tag{3.8}$$

where z_i is the activation of a unit or an input (equation 3.7 is written with x_i, which has been changed by z_i to better conceive it as an arbitrary input, either from the input neurons or from an intermediate layer). z_i is then injected through the connection w_{ji}, meaning it goes from all neurons i to the receiving neuron j. After applying the $f(.)$ non-linearity to the activation we have the final activation of the j neuron as:

$$z_j = f(a_j) \tag{3.9}$$

In equation 3.2 the output unit was denoted with a y, but again z_j generalizes to show it could be the activation of an intermediate hidden layer. However, when referring to the network outputs the notation will still be y_n. It has been mentioned that what we want are a set of weights optimized by means of an error function. We can take into account an error defined for each (input,output) training tuple of examples with the following expression:

$$E = \sum_t E^t \tag{3.10}$$

Where t enumerates the input/output pair. The error E^t is a differentiable function of the outputs, so that:

$$E^t = E^t(y_1 \cdots y_N) \tag{3.11}$$

The goal of this algorithm is to evaluate the derivatives of the error function with respect to the weights and biases of the network, as aforementioned. Because of the fact that in equation 3.10 the derivatives can be expressed as sums over the set of training examples, each output pattern can be treated separately, so the analysis from now on is focused on a single example, the t-th.

To back-propagate the error for the t-th output example the input features are injected into the network and forward-propagated by means of equation 3.7 layer by layer, thus making inferences.

The error E^t depends on the weight w_{ji} through the summed input a_j to unit j. Hence we can apply the chain rule for partial derivatives:

$$\frac{\partial E^t}{\partial w_{ji}} = \frac{\partial E^t}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}} \tag{3.12}$$

Now, the following notation can be introduced for the sake of simplicity:

$$\delta_j = \frac{\partial E^t}{\partial a_j} \tag{3.13}$$

The term δ is defined as an error term. We can use equations 3.8 and 3.2 to write the derivative as:

$$\frac{\partial a_j}{\partial w_{ji}} = z_i \tag{3.14}$$

And subsituting both equations 3.13 and 3.14 into 3.12 we get:

$$\frac{\partial E^t}{\partial w_{ji}} = \delta_j z_i \tag{3.15}$$

This result in 3.15 is expressing that the derivative is obtained by multiplying the value δ for the unit at the output end of the link (or the weight) by the value of z of the unit at the input end of the link ($z = 1$ for the biases). To evaluate the derivatives, then, the value of δ_j has to be computed for each neuron in the network to apply 3.15 afterwards. There are two different cases to compute δ:

- In the case of output units the evaluation of δ_n is simple, taking the definition in equation 3.13:

  $$\delta_n = \frac{\partial E^t}{\partial a_n} = g'(a_n)\frac{\partial E^t}{\partial y_n} \tag{3.16}$$

  To evaluate the equation 3.16 it is only required to substitute the expressions of $g'(a)$ and $\frac{\partial E^t}{\partial y_n}$ appropriately.

- In the case of hidden units we need to make use of the chain rule for partial derivatives again, such that:

$$\delta_j = \frac{\partial E^t}{\partial a_j} = \sum_n \frac{\partial E^t}{\partial a_n} \frac{\partial a_n}{\partial a_j} \tag{3.17}$$

Where the sum is running over all n units to which the unit j sends connections. Note that the units labeled n could include other hidden units or output units. Figure 3.6 illustrates the mentioned procedure with the arrangement of units and weights. Now substituting the definition of δ at equation 3.13 into 3.17, following 3.8 and 3.9:

$$\delta_j = g'(a_j) \sum_n w_{nj} \delta_n \tag{3.18}$$

This is the back-propagation formula, and it tells us that the δ value for a particular hidden neuron can be obtained with the δ from neurons in later hidden layers (or output layer).

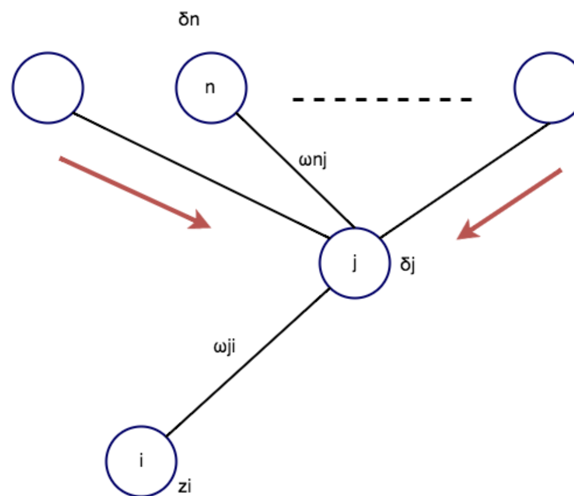

FIGURE 3.6: Calculation of δ_j for the hidden neuron j back-propagating the δ's from the connected n units (which can be output units or next-hidden units). Based on (Bishop, 1995) Figure 4.16.

Since the values of the output δ's are known from equation 3.16, the algorithm is based on applying 3.17 recursively, evaluating all the δ's for every hidden neuron in a feed-forward neural network. This is then the first part of the BP algorithm, which is the evaluation of the derivatives of the error E^t with respect to the weighs and biases. It can be summarized in four main steps:

1. Make the inference injecting the inputs **x** and propagating the activations forward.
2. Evaluate the δ_n for all output units.
3. Back-propagate the δ's to the δ_j for every hidden neuron.
4. Evaluate the derivatives with respect to the weights as in equation 3.15.

This methodology has been developed for the case of a training pattern (a t-th pattern), but the above can be repeated for every pattern in the training set so that:

$$\frac{\partial E}{\partial w_{ji}} = \sum_t \frac{\partial E^t}{\partial w_{ji}} \tag{3.19}$$

Now to have the complete learning algorithm (or the first sight of what the learning algorithm is) the weight update method has to be introduced. There are many types of strategies nowadays, but a very simple method can be a fixed-step gradient descent learning. This is based on subtracting a fixed amount of the error to the weights to correct little by little like so:

$$w_{ji}^{(k+1)} = w_{ji}^{(k)} - \Delta w_{ji}^{(k)} = w_{ji}^{(k)} - \eta_0 \sum_t \delta_j^t z_i^t \tag{3.20}$$

Where (k) remarks the current iteration, η_0 is the so called *learning rate*, and the sum operations involves a batch learning procedure (thus updating the weights after seeing a batch of data from the training set instead of a single sample). The parameters t and η_0 are chosen before the training, as well as the topology of the network and the number of iterations to update the weights. All these parameters are called *hyperparameters*, and they are part of the tuning behind the intuition of these systems. Regarding the batches, they can contain the full dataset (so a single batch would be all training samples) or slices of it, which would then be called mini-batches. Nowadays the most used methods work with mini-batches because these take advantage of the GPUs parallelization capabilities, in addition to the fact that training sets are usually quite large and they exceed the memory capacities easily. An important term to be introduced at this point is the *Epoch*, which is the amount of mini-bathes that have to be processed to see all the training set. This terminology will appear in the experimental setup of the models developed in this work. The basic mini-batch learning is called Stochastic Gradient Descent (SGD), and it comes with many flavors, because we can also set some additional parameters to it, like momentum (Sutskever et al., 2013) or some learning decay factor, with which the learning rate decays over time to converge more easily.

Regarding the learning rate we have to set it to a proper value that lets the model learn in a paced manner to converge as well as possible, but without doing it too slowly. Typical values for this are between $\{0.1, 0.001\}$. A value that is too large will lead to divergence of the learning, thus increasing the training loss rather than minimizing it. On the other hand, a learning rate that is too low will probably make the model get stuck in some minima rather than going further to a better point during the optimization. Figure 3.7 shows some feasible training cases depending on the learning rate applied.

In this work the SGD has been used and also the Adam (Kingma and Ba, 2014) optimizer, which is a computationally efficient algorithm, has little memory requirements and is invariant to diagonal rescaling of the gradients. It is well suited for problems that are large in terms of data and/or

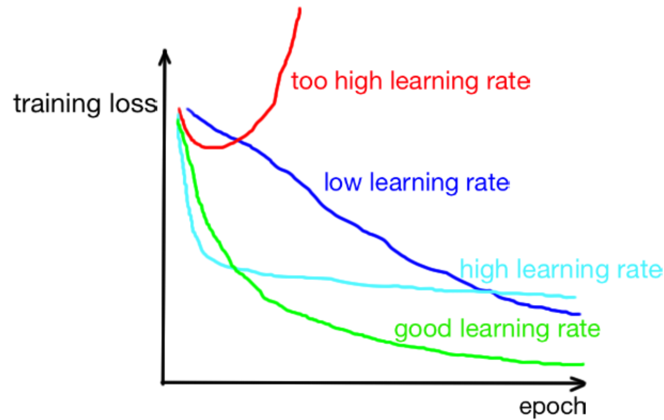

FIGURE 3.7: Qualitative examples of training loss evolution for various learning rate magnitudes.

parameters as claimed by the authors, and it performed well during the development of this work. However, it is out of the scope of this thesis to explain it in more details, as we have already seen what is needed to learn and the algorithm is an improvement over the basic details explained. One last important thing to mention is the way in which the weights and biases are initialized. It is fine if biases are initialized to be zero at the beginning of the training stage, but the weights need to have some random values, and they must be very small for propagating the data well through the non-linear activations. The reason to initialize the weights randomly is breaking the symmetry between hidden units of the same hidden layer (Bengio, 2012). The works in Orr and Müller (2003) and Glorot and Bengio (2010) propose some methodologies for a proper initialization of the weights depending on the number of units in the topology and the activation functions used.

3.2.1 Dropout Method

One of the known possible issues of Deep Neural Networks is the overfitting produced by the large amount of parameters that these models can easily have. We can make a quick calculation of how many parameters are held inside a neural network with L layers, H hidden neurons per layer (we consider all layers have the same amount of hidden units), N inputs and M outputs:

$$params = (H \times M + M) + (H \times N + H) + \sum_{l=1}^{L-2} H \times H + H \tag{3.21}$$

To exemplify the magnitude of a network, we could set 4 hidden layers with 500 neurons each, an output layer of 20 units and 300 input features, giving as a result:

$$params = (500{\times}20{+}20){+}(500{\times}300{+}500){+}\sum_{l=1}^{3} 500{\times}500{+}500 = 912020 \approx 912K$$

This is a huge amount of parameters, yet by far not one of the biggest considering the state of the art architectures. It is known from every machine learning system that when the number of parameters increases to a huge amount the chances of over-fitting the training data get quite higher. Over-fitting is the effect of learning too much about the training set used to update the weights of our model. We can see this effect for an hypothetic binary classification task in Figure 3.8. This is bad because it means that our model does not generalize well, so that when test data (new samples not seen during training) is evaluated it is likely that the model does not perform really well, because it learned noisy characteristics of the training set and not the truth underlying pattern that generates the real data. To combat this effect there are two main ways:

- Add more training data to fill in the blanks.
- Reduce the complexity of the model by means of reducing the amount of parameters to learn.

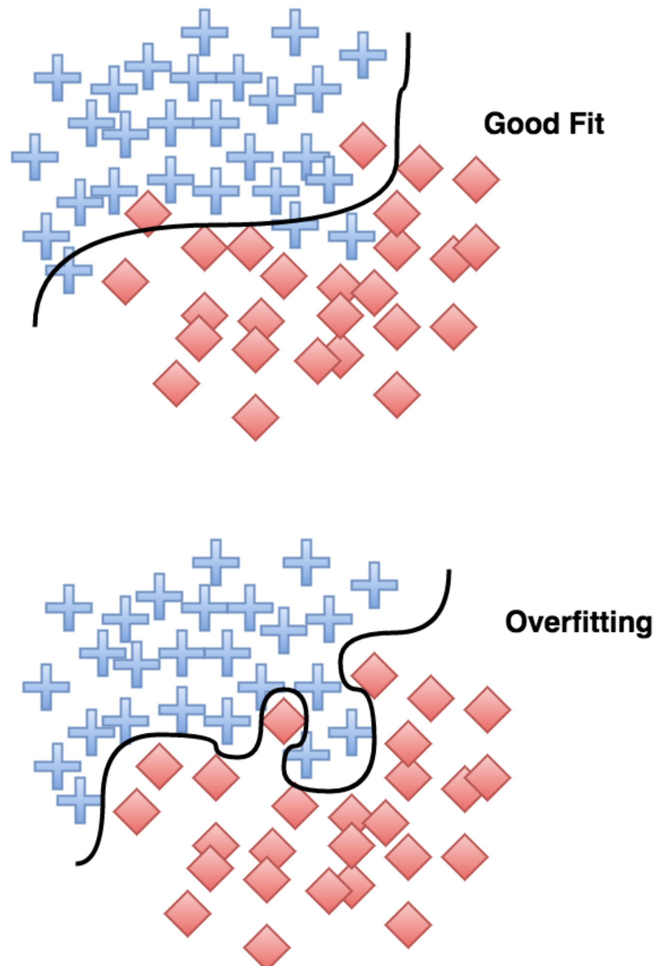

FIGURE 3.8: Examples of a good fit and an overfitting for a binary classification task with a $2-D$ space (i.e. 2 input features).

In the case of a Neural Network there are some techniques to implement the first idea by means of generating artificial examples out of the distortion of

the original data. The possibility to do this depends on the type of data we are facing (e.g. images, speech, etc.), and it is called *data augmentation*. The second approach is more feasible to adapt to every type of data we have, but reducing the number of parameters could result in diminishing too much the power of our model. A softer solution to this is using *regularization*, which means imposing restrictions to the learning process such that the weights are learned in a controlled manner.

An interesting and effective regularization method is the Dropout method presented here and proposed in Srivastava et al. (2014), which is based on making a poll to a set of thinned neural networks (networks with randomly dropped out neurons in some layers) so that the final decision of our multi-layered model will be voted by many thinned networks. The key idea of this algorithm is to randomly drop units during training, which prevents those neurons to co-adapt too much to the training data.

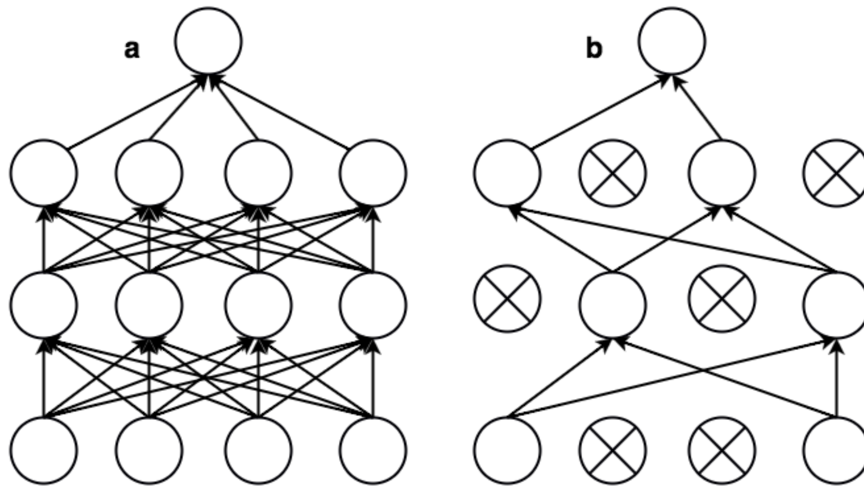

FIGURE 3.9: (a) Standard network with 2 hidden layers. (b) Example of thinned network by applying dropout to the network on (a). Based on (Srivastava et al., 2014) Figure 1.

Dropping a unit means temporarily removing it from the network along with its connections (inputs and outputs). The choice of which units are dropped is performed randomly with a probability p, which is independent for every unit. The authors claimed that setting it to $p = 0.5$ seems to be close to optimal for a wide range of networks and tasks. However if some dropout is going to be applied to the input units, it is better to give it a value closer to 1. For each presentation of a training case a new thinned network is built and trained. During test time, all neurons are present but they have smaller weights because of the dropped links during training.

Also, another technique that can be used as a regularization mechanism and can also be used in conjunction with Dropout is taking a separate data split to that from training to validate our model after each epoch. This means that after updating the network for an epoch we make an inference with the so called validation examples (from the validation set), and by means of observing the evolution of the error committed by the network predicting the validation examples we decide whether to stop or not the training before all the epochs have been completed. This is called early-stopping, and it is used during model search in this work together with the

Dropout.

3.3 Deep Neural Network

The concept of Deep Neural Network (DNN) is based on stacking many hidden layers to make the model deeper. Neural Networks appeared decades ago, but building deep models was not a simple task by then, because the BP algorithm did not work well when dealing with deep architectures as the gradients vanished easily, and it was also usual falling at a local minima, because increasing the model complexity makes it highly non-linear so that local minima might be more frequent. Training deep architectures was also quite slow because of the lack of computational resources. And another very important drawback was the lack of data. Nowadays it is different though, because the Internet provoked a huge increase in the amount of data available (images, videos, speech, audio, text, etc.), and the Graphical Processing Units (GPUs) make the matrix operations really fast and efficiently (for instance, the equation 3.7). Also, new learning algorithms like Adam (Kingma and Ba, 2014), make the learning faster with a better propagation of the gradients through the model. Furthermore, many initialization schemes for the weights have been proposed so that it gets harder to fall into a local minima, and this has been a trendy area of research over the last years in Deep Learning. These methods are mostly based on pretraining the models (i.e. give some meaningful values to the weights in order to make them not random and close to what we want to achieve). It is out of the scope to explain how these mechanisms work, however the Restricted Boltzman Machines (*Tutorial on Restricted Boltzman Machines (RBM)* 2010), the Denoising Autoencoders (*Tutorial on Denoising Autoencoders (DA)* 2010) and the Deep Belief Networks (*Tutorial on Deep Belief Networks* 2010) are referenced if the readers want to get further details. Special attention is given to recurrent architectures as they are the main type of model considered in this work.

3.4 Recurrent Neural Network

Recurrent Neural Networks (RNNs) are a special type of Neural Network topology well suited for processing sequences. The *recurrent* keyword stands for the fact that these networks perform the same task for every element of a sequence (*Recurrent Neural Networks Tutorial* 2015), having an output that depends on previous computations. This characteristic can be seen as a memory feature that these models have, as they "remember" the past flows of information. When we talk about a unidirectional recurrent layer we can say that it has memory about the past, so at time t they have an input vector $\mathbf{x}_t \in \mathbb{R}^n$ and the memory state at time $t-1$ $\mathbf{h}_{t-1} \in \mathbb{R}^m$, producing the new memory state $\mathbf{h}_t$, also called hidden state, with the following set of operations:

$$\mathbf{h}_t = g(\mathbf{W} \cdot \mathbf{x}_t + \mathbf{U} \cdot \mathbf{h}_{t-1} + \mathbf{b}_h) \tag{3.22}$$

where $\mathbf{W}$ is the input-to-hidden weights matrix (feed forward behavior), $\mathbf{U}$ is the hidden-to-hidden weights matrix where the feedback is made, $\mathbf{b}_h$ is the bias vector and g is a specified element-wise non-linear transformation, such as the hyperbolic tangent or the sigmoid seen previously. As we can see, for every input sequence $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T\}$ we obtain an output sequence $\mathbf{H} = \{\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_T\}$, where each output from the layer keeps track of dynamic changes in time, and this is what makes the recurrent model a really powerful option for sequences.

These models then keep track of the context for the input features, and a very interesting property they have is that they share the same parameters for every time-step of the process, thus reducing the amount of weights needed to take into account a large context. It is usual to visualize an RNN unfolded in time, like in Figure 3.10. There the output is also shown as the result of another matrix computation, though it is not part of the recurrent layer:

$$\mathbf{y}_t = f(\mathbf{V} \cdot \mathbf{h}_t + \mathbf{b}_v) \tag{3.23}$$

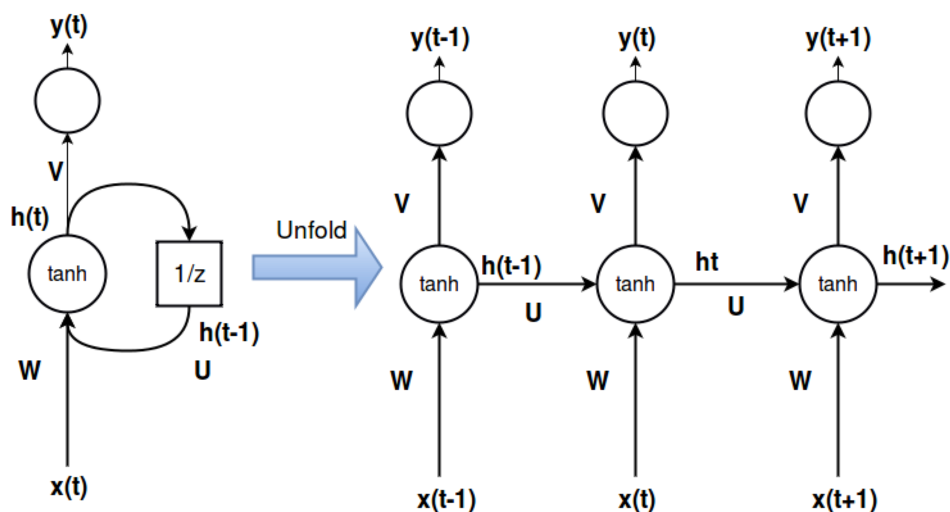

FIGURE 3.10: Example RNN neuron unfolded in time.

The training for RNNs is similar to that of feed forward ones by means of using a BP algorithm, but here the time dimension also needs to be taken into account. This is important because as mentioned previously, the parameters of the recurrent matrix are shared between time-steps, thus the gradient depends not only on the current time step, but also on the previous ones. For instance as it is exemplified in (*Recurrent Neural Networks Tutorial* 2015) if we wanted to compute the gradient at $t = 4$ we would need to back-propagate 3 steps and sum up the gradients. This technique is called Back-Propagation Through Time (BPTT), and a quick review to take a glance at the different with the standard one is given in the next section.

3.5 Back-Propagation Through Time

First and as previously settled for standard BP (see section 3.2), a cost function is defined to train our RNN, and in this case the total error at the output of the network is the sum of the errors at each time-step:

$$E(\mathbf{y}, \hat{\mathbf{y}}) = \sum_{t=1}^{T} E_t(\mathbf{y}_t, \hat{\mathbf{y}}_t) \tag{3.24}$$

Where $\mathbf{y}_t$ is the ground-truth example we show to the network at time-step t and $\hat{\mathbf{y}}_t$ is its prediction. For the case of the network shown in Figure 3.10 the goal of BPTT is to compute the gradients with respect to our parameters **W**,**U** and **V** to learn the appropriate weight values, so it is the same idea as in the feed forward BP. Just as it is done in equation 3.24 to sum up the errors, the gradients also get summed through time as:

$$\frac{\partial E}{\partial \mathbf{W}} = \sum_{t=0}^{T-1} \frac{\partial E_t}{\partial \mathbf{W}} \tag{3.25}$$

The following procedure can be found in more detail in (*Recurrent Neural Networks Tutorial* 2015). The chain rule is applied again here, and the first gradients we refer to are the ones for **V** which are quite straightforward:

$$\frac{\partial E_3}{\partial \mathbf{V}} = \frac{\partial E_3}{\partial \hat{\mathbf{y}}_3}\frac{\partial \hat{\mathbf{y}}_3}{\partial \mathbf{V}} = \frac{\partial E_3}{\partial \hat{\mathbf{y}}_3}\frac{\partial \hat{\mathbf{y}}_3}{\partial \mathbf{z}_3}\frac{\partial \mathbf{z}_3}{\partial \mathbf{V}} \tag{3.26}$$

Where $z_3 = \mathbf{V} \cdot h_3$. Beware that the time-step 3 has been considered to continue with the first example that was mentioned with the RNN gradient computations. An important thing to note here is how $\frac{\partial E_3}{\partial \mathbf{V}}$ does not depend on other values than the current time-step, as the dependency is built upon ($\hat{\mathbf{y}}_3$,$\mathbf{y}_3$,$\mathbf{h}_3$). It is not like so with the matrices within the recurrent layer however, because for **U** matrix for example:

$$\frac{\partial E_3}{\partial \mathbf{U}} = \frac{\partial E_3}{\partial \hat{\mathbf{y}}_3}\frac{\hat{\mathbf{y}}_3}{\partial \mathbf{h}_3}\frac{\partial \mathbf{h}_3}{\partial \mathbf{U}} \tag{3.27}$$

Now from equation 3.22 we see that $\mathbf{h}_3$ depends on $\mathbf{h}_2$, which in turn depends on $\mathbf{h}_1$ and so forth. This means that taking the derivative is applying the chain rule for as many time-steps as we have:

$$\frac{\partial E_3}{\partial \mathbf{U}} = \sum_{t=0}^{3} \frac{\partial E_3}{\partial \hat{\mathbf{y}}_3}\frac{\hat{\mathbf{y}}_3}{\partial \mathbf{h}_3}\frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{U}} \tag{3.28}$$

This is expressing that the gradient is accumulated through time and not only in the feed-forward way. Figure 3.11 illustrates the gradient flowing backwards from the current time-step. As previously seen with the feed-forward BP-algorithm, a delta can also be defined here, being it:

$$\delta_2^{(t)} = \frac{\partial E_t}{\partial \mathbf{z}_{t-1}} = \frac{\partial E_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}\frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{z}_{t-1}} \tag{3.29}$$

Where $\mathbf{z}_{t-1}$ is defined to be the activation (previous to applying any non-linearity) of the recurrent layer:

$$\mathbf{z}_{t-1} = \mathbf{W} \cdot \mathbf{x}_{t-1} + \mathbf{U} \cdot \mathbf{h}_1 \tag{3.30}$$

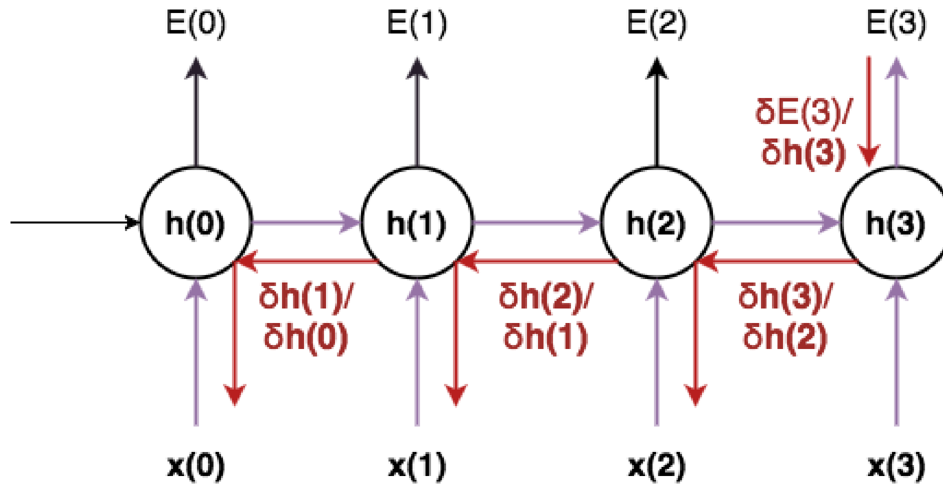

FIGURE 3.11: Gradient flow at the fourth time-step 3. Violet arrows: inference flow of the input data. Red arrows: back-propagated gradients/errors.

In practice the distance-in-time can get too large and that makes the gradient propagation difficult, because the back-propagation in time can be seen as back-propagating through many layers (as many as time-steps), so the number of time-steps is truncated to have a maximum length of the training sequences, and then the algorithm is sometimes called Truncated Back-Propagation Through Time.

This type of RNN are usually called the vanilla ones, which stands for the basic model with $\tanh$/sigmoidal units. In theory they should work well to model any time dependencies so that they recall any context within a range of T time-steps, but in practice they suffer from two problems: *The vanishing gradients* and *The exploding gradients*. Both of these problems are based on the inherent behavior of the equation 3.22 applied time-step by time-step:

$$\mathbf{h}_t = g(\mathbf{W} \cdot \mathbf{x}_t + \mathbf{U} \cdot g(\cdots g(\mathbf{W} \cdot \mathbf{x}_{t-T} + \mathbf{U} \cdot \mathbf{h}_{t-T} + \mathbf{b}_h) \cdots) + \mathbf{b}_h) \tag{3.31}$$

The process of multiplying the matrix $\mathbf{U}$ at every time-step is what makes the gradients so unstable, rapidly diminishing to zero or exploding depending on the value of the highest eigenvalue of the matrix. This makes it really complicated to train these models in an effective manner and also provokes the lost of long-term dependencies in memory when doing inference, because the multiplicative behavior of these neurons do not preserve well the information through long sequences. To overcome these effects there are extensions of the vanilla RNN, leading to what are called memory cells. Currently there are two main types of these cells: Long Short Term Memory (LSTM) cells and Gated Recurrent Units (GRU) (Cho et al., 2014b). The

former ones have been extensively used in this work and this is why the next section concludes the background by presenting them. Nevertheless, there are some really interesting works exploring the capabilities of some of these architectures empirically, showing that they perform quite similar most of the time (Chung et al., 2014)(Wu and King, 2016).

3.6 Long Short Term Memory

In this work we used LSTM layers, as they cope better with the vanishing gradient problems (Hochreiter, 1998) that appeared when training regular RNNs as mentioned previously. They also model the long term dependencies in a better way than the simple RNNs do because of their gating mechanisms (Hochreiter and Schmidhuber, 1997). The structure of an LSTM cell is shown in Figure 3.12. There are three gates in this structure:

- Input gate: control the flow of information coming in.
- Forget gate: control which components of the cell state are forgotten (i.e. multiplying by zero to delete from memory).
- Output gate: control the flow of information going out.

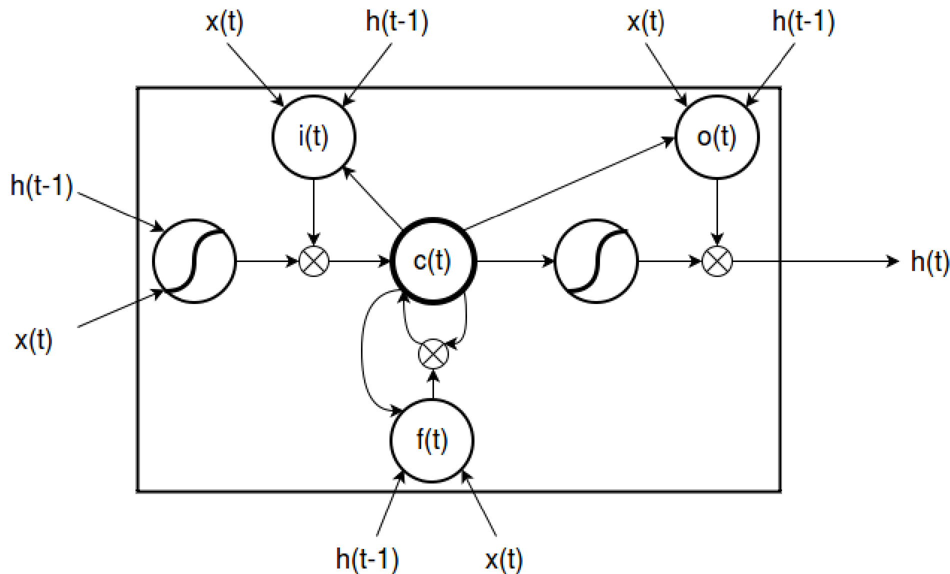

FIGURE 3.12: Architecture of an LSTM cell. i(t): input gate at time-step t. o(t): output gate at time-step t. f(t): forget gate at time-step t. c(t): cell state at time-step t.

Figure 3.13 shows the unfolded version of the LSTM cell, where we can see two main flows of information through time: $\mathbf{h}_t$ and $\mathbf{c}_t$. The equations that describe this architecture are the following ones:

$$\mathbf{i}_t = \sigma(\mathbf{W}_i\mathbf{x}_t + \mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{b}_i) \tag{3.32}$$

$$\hat{\mathbf{C}}_t = \tanh(\mathbf{W}_c\mathbf{x}_t + \mathbf{U}_c\mathbf{h}_{t-1} + \mathbf{b}_c) \tag{3.33}$$

$$\mathbf{f}_t = \sigma(\mathbf{W}_f \mathbf{x}_t + \mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{b}_f) \tag{3.34}$$

$$\mathbf{C}_t = \mathbf{i}_t \odot \hat{\mathbf{C}}_t + \mathbf{f}_t \odot \mathbf{C}_{t-1} \tag{3.35}$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_o \mathbf{x}_t + \mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{b}_o) \tag{3.36}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{C}_t) \tag{3.37}$$

The symbol $\odot$ means element-wise product here. We can see how there are many matrices now, as well as bias vectors: $\mathbf{W}_i$, $\mathbf{U}_i$, $\mathbf{W}_f$, $\mathbf{U}_f$, $\mathbf{W}_o$, $\mathbf{U}_o$, $\mathbf{W}_c$, $\mathbf{U}_c$, $\mathbf{b}_i$, $\mathbf{b}_f$, $\mathbf{b}_o$ and $\mathbf{b}_c$. All these weights are now parameters to be learned for the LSTM layer, so the architecture has become way more complex that the vanilla one, but the interesting thing is that the gating mechanisms control what to keep and what to forget about the inputs and the past. There is a good analysis in *Understanding LSTM Networks* (2015) about it, summarized here.

The gates are seen as soft-switches (because of the sigmoid that is bounded $\sigma \in \{0, 1\}$, but with a transition of real values between the boundaries):

- The forget gate then decides whether to forget contents or not for each cell in the LSTM layer (i.e. we have a vector of forget activations per time-step) by multiplying the past cell states by its activation value: 0 means forget completely about what was seen, 1 means keep it all. The forget activation is obtained by looking at the past $\mathbf{h}_{t-1}$ and at the current input $\mathbf{x}_t$.
- The input is on behalf of deciding whether the new information available at the input gets into the memory state or not. This process is divided in two steps:
  - Get the activation of the soft-switch by looking at the past $\mathbf{h}_{t-1}$ and the input $\mathbf{x}_t$.
  - A candidate vector $\hat{\mathbf{C}}_t$ is computed (equation 3.33) that could be added to the memory cell state.

  Then the results of these two steps are combined to update the state with the right amount and type of information.
- Next, the new cell states $\mathbf{C}_t$ are computed. It is important to note that the computation (equation 3.35) includes applying the forget element-wise product to every past cell state. Also, the combination of "What information should get into the memory" and the candidate memory are multiplied, and this way of updating the information stored in the cell is the main difference with the vanilla RNN. Here the updates are computed through a summation instead of a multiplication, and also the regulation of input and forgot flows makes these systems able to store long-term dependencies.

- To finish, the output gate activation is computed (from $\mathbf{h}_{t-1}$ and $\mathbf{x}_t$ again) and applied to the new cell state to finally get the right components to generate (i.e. "what is actually required from the different cells in the layer to generate the right information?").

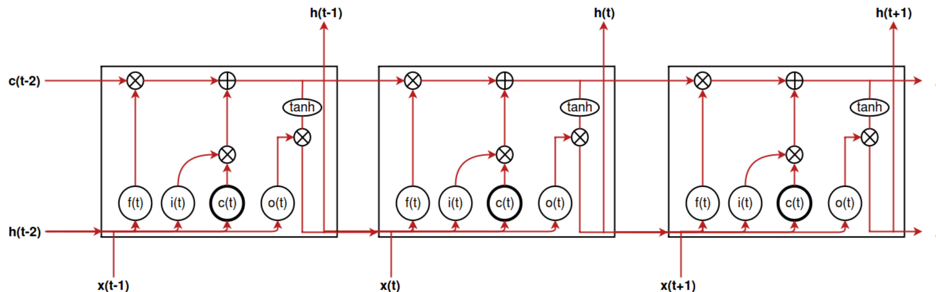

FIGURE 3.13: LSTM cell unfolded in time. The red arrows depict the inference flow of data between time-steps.

3.7 Summary

In section 3.1 we have went through the basics of what an Artificial Neural Network (ANN, NN, MLP) is, from its elemental unit (the neuron), to the full architecture of the feed-forward construction. The idea of the different layers (Input, Hidden and Output) has been explained, detailing why any hidden units are required to make a more powerful model with a toy example, which was the XOR. Also, the layer activation equation was shown and explained, looking at the components that compose it.

Following to the basics, the Back-Propagation algorithm has been discussed in section 3.2 in order to see the learning details of the neural networks. There some terminology has been introduced to be understood in later chapters when dealing with the models developed for this work. The feed forward architectures explanations conclude with a brief description of the Deep Neural Network architectures, where some past issues are exposed regarding the difficulties of training these models and some current solutions the problems are mentioned.

The last part of the chapter is focused on Recurrent Neural Networks and the way in which they are trained, and more concretely Long Short Term Memory networks and their sequence modeling capabilities are explored. Also, a detailed explanation of the LSTM gating mechanisms is given, something that is useful for the gating analysis performed in Chapter 4.

Chapter 4

Two stage Text-to-Speech with RNN-LSTM

4.1 Introduction

The main purpose of this work has been building a state of the art Text To Speech (TTS) system with Deep Learning techniques. As seen in Chapter 2, some previous works performed quite well with Deep Neural Network variants and also with recurrent architectures, either as standalone systems for mapping linguistic contents to acoustic frames, or to replace some parts of the SPSS pipeline. In this section, the process to build the full TTS system for this thesis is explained, which is fully built with RNN architectures as later will be seen. Figure 4.1 represents the general scheme of the developed work.

The system is subdivided in two main steps: training and synthesis, similar to the SPSS system seen in Chapter 2. During training there is a Speech Database available that contains contextual features and speech recordings. The contextual features are generated from analyzing the textual transcriptions of those recordings, and they will be presented in section 4.2.1. It is basically a representation of the linguistic complexities that must be derived into speech signal. Out of the recordings of that database many acoustic features are also extracted by means of a Vocoder, as explained in section 4.2.2, which are used to train two models: duration RNN and acoustic RNN. This two stage architecture was influenced by the work in Zen and Sak (2015). Both training processes are depicted in Figure 4.1 in the *Duration RNN training* and *Acoustic RNN training* blocks. Once the models are trained the weights are saved for later usage.

The second part of the system is the synthesis stage, where both trained models are used in a pipeline fashion. First, raw text is inserted by the user and the text analysis front-end converts it to contextual features. Then these features are injected into the duration model that will predict the duration of each phoneme one by one, sending its predictions to the acoustic model, which will generate the acoustic parameter trajectories for a Vocoder. In the end, the Vocoder will convert the parameterization of the speech into the waveform again.

This chapter is structured in the following manner: Section 4.2 describes in detail what data is required to train this model and how it is prepared, with a parallelized pipeline architecture to speed things up. The Two-stage-TTS modules are described in Section 4.3, and a post-processing improvement to

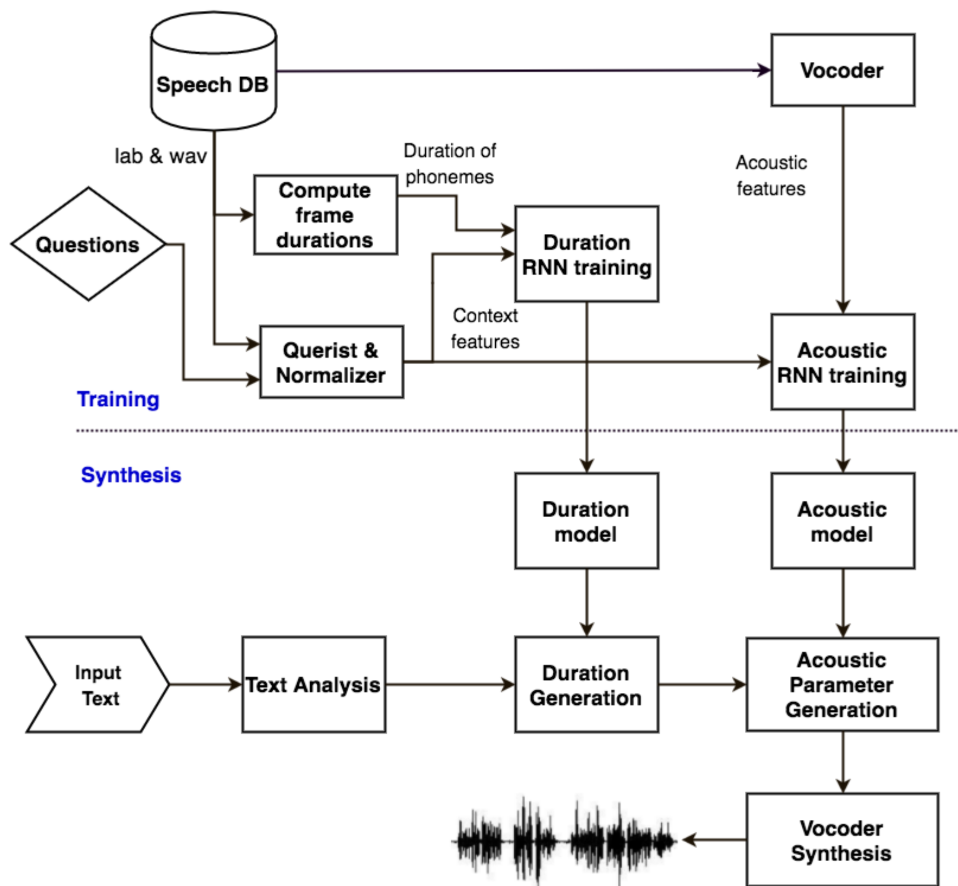

FIGURE 4.1: Schematic of the developed two stage Text-To-Speech system.

achieve better naturalness in the synthesis is applied in section 4.4. Finally, results are shown and discussed in section 4.5.

4.2 Data Preparation

In this section it is explained the way in which data is generated. This is basically the process to go from raw data to features that will be processed either as predictors or as predictions. To clarify things and avoid confusion, input features are called predictors and outputs are called predictions. First, the textual features, which will be mainly predictors, are explained. Their types and the process they go through is depicted, such that it is understood why are these features chosen for the synthesis purpose. Then, acoustic features will be described, as well as the process they go through to produce the speech stream with the Vocoder.

A parallelization mechanism to speed up the generation of features is also briefly shown. It is considered to be an important part of the system and the project, owing to the fact that it lets us make more experiments and quicker if we want to generate different sets of speakers' features. We will see in the next chapter that we require data from many speakers to be generated, so this tool is useful for that.

4.2.1 Text to Label process

As mentioned earlier, raw text is processed into a more convenient representation that we call *label*. This representation is composed of a set of contextualized prosodic and phonetic features. The features are a phonetic transcription of a few windowed phonemes, so that the synthesis of the current phoneme takes into account the surrounding phonemes for co-articulation purposes. Also, information about stressed syllables, position of the phoneme inside the current syllable, the position of the syllable in the word, etc. is embedded in these features. Table 4.1 shows the label format features/ symbols and their descriptions.

label format	
Symbol	Description
p1	phoneme identity before the previous phoneme
p2	previous phoneme identity
p3	current phoneme identity
p4	next phoneme identity
p5	the phoneme after the next phoneme identity
p6	position of the current phoneme identity in the current syllable (forward)
p7	position of the current phoneme identity in the current syllable (backward)
a1	whether the previous syllable is stressed or not (0; not, 1: yes)
a2	whether the previous syllable is accented or not (0; not, 1: yes)
a3	number of phonemes in the previous syllable
b1	whether the current syllable stressed or not (0: not, 1: yes)
b2	whether the current syllable accented or not (0: not, 1: yes)

b3	the number of phonemes in the current syllable
b4	position of the current syllable in the current word (forward)
b5	position of the current syllable in the current word (backward)
b6	position of the current syllable in the current phrase(forward)
b7	position of the current syllable in the current phrase(backward)
b8	number of stressed syllables before the current syllable in the current phrase
b9	number of stressed syllables after the current syllable in the current phrase
b10	number of accented syllables before the current syllable in the current phrase
b11	number of accented syllables after the current syllable in the current phrase
b12	number of syllables from the previous stressed syllable to the current syllable
b13	number of syllables from the current syllable to the next stressed syllable
b14	number of syllables from the previous accented syllable to the current syllable
b15	number of syllables from the current syllable to the next accented syllable
b16	name of the vowel of the current syllable
c1	whether the next syllable stressed or not (0: not, 1:yes)
c2	whether the next syllable accented or not (0: not, 1:yes)
c3	the number of phonemes in the next syllable
d1	gpos (guess part-of-speech) of the previous word
d2	number of syllables in the previous word
e1	gpos (guess part-of-speech) of the current word
e2	number of syllables in the current word
e3	position of current word in the current phrase (forward)
e4	position of current word in the current phrase (backward)
e5	number of content words before the current word in the current phrase
e6	number of content words after the current word in the current phrase
e7	number of words from the previous content word to the current word
e8	number of words from the current word to the next content word
f1	gpos (guess part-of-speech) of the next word
f2	number of syllables in the previous word
g1	number of syllables in the previous phrase
g2	number of words in the previous phrase
h1	number of syllables in the current phrase
h2	number of words in the current phrase
h3	position of the current phrase in utterance (forward)
h4	position of the current phrase in utterance (backward)
h5	Phrase modality (question, exclamation, etc.)
i1	number of syllables in the next phrase
i2	number of words in the previous phrase
j1	number of syllables in this utterance

j2	number of words in this utterance
j3	number of phrases in this utterance

TABLE 4.1: Context-dependent label format.

We call a text transcription file a utterance. When we convert the text into a label, the features are structured in the following format:

```
p1^p2$-$p3$+$p4$=$p5~p6_p7/A:a1_a2_a3/B:b1-b2-b3~b4-b5...
&b6-b7#b8-b9$b10-b11!b12-b13;b14-b15|b16/C:c1+c2+c3/D:d1_d2...
/E:e1+e2~e3+e4&e5+e6#e7+e8/F:f1_f2/G:g1_g2/H:h1=h2~h3=h4|h4...
/I:i1_i2/J:j1+j2-j3
```

So for instance, if we had a utterance with the Spanish sentence *"buenos días"*(good morning) in it, the label processing system would output the label file depicted in Figure 4.2, which follows the format aforementioned.

```
_^_-pau+b=w~1_1/A:0_0_0/B:0-0-1~1-1&0-1#0-0$0-0!0-0;0-1|_/C:0+1+3/D:SIL_0/E:SIL+1~0+1&0+0#0+0/F:AQ_2/G:0_0/H:0=0~2=1|0/I:4_3/J:5+6-1
_^pau-b+w=e~1_3/A:0_0_1/B:0-1-3~1-2&1-4#0-0$0-2!0-0;0-2|e/C:0+0+3/D:SIL_0/E:AQ+2~1+3&0+3#0+1/F:UNKNOWN_1/G:0_0/H:4=3~2=1|0/I:0_0/J:5+6-1
pau^b-w+e=n~2_2/A:0_0_1/B:0-1-3~1-2&1-4#0-0$0-2!0-0;0-2|e/C:0+0+3/D:SIL_0/E:AQ+2~1+3&0+3#0+1/F:UNKNOWN_1/G:0_0/H:4=3~2=1|0/I:0_0/J:5+6-1
b^w-e+n=o~3_1/A:0_0_1/B:0-1-3~1-2&1-4#0-0$0-2!0-0;0-2|e/C:0+0+3/D:SIL_0/E:AQ+2~1+3&0+3#0+1/F:UNKNOWN_1/G:0_0/H:4=3~2=1|0/I:0_0/J:5+6-1
w^e-n+o=z~1_3/A:0_1_3/B:0-0-3~2-1&2-3#0-0$1-2!0-0;1-1|o/C:0+1+2/D:SIL_0/E:AQ+2~1+3&0+3#0+1/F:UNKNOWN_1/G:0_0/H:4=3~2=1|0/I:0_0/J:5+6-1
e^n-o+z=D~2_2/A:0_1_3/B:0-0-3~2-1&2-3#0-0$1-2!0-0;1-1|o/C:0+1+2/D:SIL_0/E:AQ+2~1+3&0+3#0+1/F:UNKNOWN_1/G:0_0/H:4=3~2=1|0/I:0_0/J:5+6-1
n^o-z+D=a~3_1/A:0_1_3/B:0-0-3~2-1&2-3#0-0$1-2!0-0;1-1|o/C:0+1+2/D:SIL_0/E:AQ+2~1+3&0+3#0+1/F:UNKNOWN_1/G:0_0/H:4=3~2=1|0/I:0_0/J:5+6-1
o^z-D+a=a~1_2/A:0_0_3/B:0-1-2~1-1&3-2#0-0$1-1!0-0;2-1|a/C:0+1+2/D:AQ_2/E:UNKNOWN+1~2+2&1+2#1+1/F:NC_1/G:0_0/H:4=3~2=1|0/I:0_0/J:5+6-1
z^D-a+a=s~2_1/A:0_0_3/B:0-1-2~1-1&3-2#0-0$1-1!0-0;2-1|a/C:0+1+2/D:AQ_2/E:UNKNOWN+1~2+2&1+2#1+1/F:NC_1/G:0_0/H:4=3~2=1|0/I:0_0/J:5+6-1
D^a-a+s=pau~1_2/A:0_1_2/B:0-1-2~1-1&4-1#0-0$2-0!0-0;1-0|a/C:0+0+0/D:UNKNOWN_1/E:NC+1~3+1&2+1#1+0/F:SIL_0/G:0_0/H:4=3~2=1|0/I:0_0/J:5+6-1
a^a-s+pau=_~2_1/A:0_1_2/B:0-1-2~1-1&4-1#0-0$2-0!0-0;1-0|a/C:0+0+0/D:UNKNOWN_1/E:NC+1~3+1&2+1#1+0/F:SIL_0/G:0_0/H:4=3~2=1|0/I:0_0/J:5+6-1
a^s-pau+_=_~1_1/A:0_1_2/B:0-0-1~1-1&0-5#0-0$0-3!0-0;1-0|_/C:0+0+0/D:NC_1/E:SIL+1~0+4&0+3#1+0/F:SIL_0/G:0_0/H:4=3~2=1|0/I:0_0/J:5+6-1
```

FIGURE 4.2: Label file example resulting from processing the Spanish text *"buenos días"*.

This format has been widely used in SPSS, and it is an accepted well formatted description of many important characteristics to take into account for generating the speech stream. In the case of our TTS, the label generator is the front-end of Ogmios (Bonafonte et al., 2006a), which is a TTS system developed by UPC including modules for text processing, prosody modeling and speech generation. The way in which this data is injected to the neural network model will be seen in section 4.2.3.

Now that labels have been introduced, we can derive more information from them. It is so that another type of features were also used in SPSS, which were extracted from the labels contextual information. Previously in Chapter 2 the clustering method when building the HMM models was introduced. The way in which those clustering trees are built is by means of a set of questions related to the identities appearing in the labels (see Figure 2.6). These questions are designed considering the characteristics of the language we are processing. For instance we have the following questions for the current phoneme (p3) appearing in the label:

- is the current phoneme a Vowel?
- is the current phoneme Affricate?
- is the current phoneme Consonant?

- is the current phoneme Palatal?
- is the current phoneme Approximant?
- is the current phoneme Alveolar?
- is the current phoneme Glottal?
- is the current phoneme Bilabial?
- is the current phoneme Plosive?
- (...)

These questions are a very brief example extracted from the real ones, referring to the current phoneme only (p3). However, there are questions related to all the other phonemes (p1,p2,p4,p5) and Part-Of-Speech tags (d1,e1,f1). We concatenate the information given by answering these questions to the label features for our own TTS, because we also want to consider the type of content included in the label. The total amount of questions taken from the ones used in SPSS are shown in Table 4.2. There, the entities where questions are applied are depicted jointly with the amount of questions asked.

Entity	# Questions
$LL - phoneme$	33
$L - phoneme$	33
$C - phoneme$	33
$R - phoneme$	33
$RR - phoneme$	33
$L - Word - GPOS$	8
$C - Word - GPOS$	8
$R - Word - GPOS$	8

TABLE 4.2: Number of questions per Entity. LL:Left-Left, L:Left, C:Central, R:Right, RR:Right-Right

4.2.2 Acoustic parameters

The text-to-speech system built in this work does not generate the voice waveform out of the neural network itself, but it rather uses an intermediate speech generation module called Vocoder, shown in Figure 4.1. The Vocoder takes the raw speech and, by windowing it, it extracts many acoustic frames composed of features that describe the signal in a more convenient way, having good mathematical properties. This process is called encoding. On the other hand and also continuing with our TTS output, the Vocoder does the same the other way around for decoding: it takes the parametric representation of the speech, thus the acoustic frames, and converts them back to the speech signal. In this work we used Ahocoder (Erro et al., 2011), and the encoding process schematic is shown in Figure 4.3, where it can be seen how it takes the speech and generates three types of features: Mel Frequency Cepstral Coefficients, Voiced Frequency (FV for its name given by the authors in Spanish) and $\log$-F0 contour.

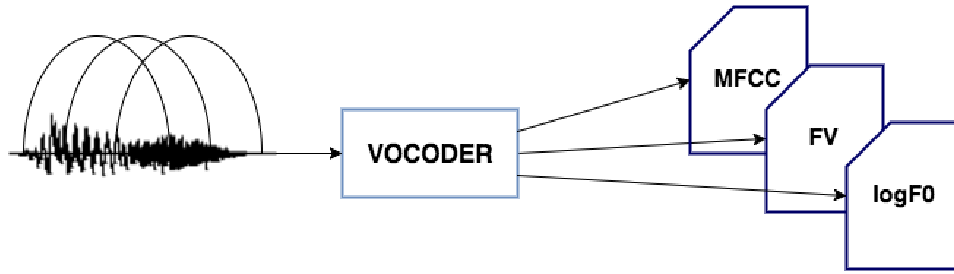

FIGURE 4.3: Schematic representation of the Vocoder for encoding the input voice with windowed frames into acoustic parameters.

Ahocoder is an Harmonics plus Noise Model (HNM), meaning it decomposes the speech frames into an harmonic part and a stochastic part (Stylianou, 1990). During the analysis stage and given an input signal the system calculates:

- F_0 estimation, being 0 if the frame is unvoiced. Once it is estimated the natural logarithm is taken, such that for all the frames we finally obtain what is called the log-F0 contour.
- The Mel Frequency Cepstral Coefficients that model the spectral envelope of the speaker at every frame. In the case of this work MFCCs of order $p = 39$ were used, which means that we have 40 coefficients.

During the decoding stage the generation of the speech is modeled by equation 4.1.

$$s(t) = \sum_i A_i(t) \cdot \cos(2\pi i f_0(t)t + \Phi_i(t)) + e(t) \tag{4.1}$$

where we can see the stochastic component e(t) and the harmonic one composed of the sum of cosines. The harmonic component captures the locally periodic part of the signal whilst the stochastic one contains all the signal events that cannot be captured by the harmonic one (aspiration noise, bursts, etc.).

The noise component is usually modeled as Gaussian noise passing through a shaping filter, and it is present always in every frame either if it is voiced or unvoiced. In the case of voiced frames however the harmonic component is also present, and its amplitudes $A_i(t)$ are obtained by sampling the MFCC envelope. The separation between the harmonic parameterization and the noise component in voiced frames during reconstruction is performed by means of a division in the spectrum at the Voiced Frequency ($f_v(t)$).

The resulting number of acoustic parameters generated by the Vocoder are then 42 per frame, where 40 are MFCC coefficients, the 41-st is the log-F0 value and the 42-nd is the Voiced Frequency. This has been a very brief introduction to the Ahocoder system, based on the work by Erro et al. (2011).

The mentioned parameterization coefficients (plus an addition term introduced when presenting the acoustic model) are then the predictions of the neural network built in this work. During synthesis we then have a set of

acoustic parameters to be passed through the Ahocoder in order to recover the voice waveform. The way in which these parameters and the linguistic ones are prepared for the network is explained in the following section.

4.2.3 Encoding and normalizing the features

Now that textual and acoustic features have been presented, the way in which they are encoded is described here. First, as mentioned earlier in Chapter 3, output predictors have to be normalized for a well-behaved back-propagation of the gradients. Because of this, it is clear that the normalization of the acoustic features will be in a min-max range of their real values, which is performed in the following way:

$$\hat{y} = \frac{y - y_{min}}{y_{max} - y_{min}} \tag{4.2}$$

Nevertheless, it is common to make a little change in this equation in the literature, which is making the output be bounded by $\{0.01, 0.99\}$ for better convergence properties:

$$\hat{y} = (0.99 - 0.01)\frac{y - y_{min}}{y_{max} - y_{min}} + 0.01 \tag{4.3}$$

All acoustic features are then normalized like so, but it is different with the textual features. They are inputs so they also have to be normalized for a proper learning process because of the non linearities of the network (LeCun et al., 2012), but they can be normalized in two different ways:

- the input is bounded between $\{0, 1\}$
- the input is z-normalized, such that it has $\mu = 0$ and $\sigma = 1$

In our case the distance features in the label, which are real values, are z-normalized:

$$\hat{x} = \frac{x - \mu}{\sigma} \tag{4.4}$$

Also, the categorical values have to take some numeric type, but the code must not include any ordering information. This means that if we had the categories (A,B,C), a way to encode them would be with some ordering, for instance (1,2,3). Nevertheless, this imposes that A and C are two extreme values in an ordered scale, meaning that, for instance, C has more value than A. The solution to this is using an orthonormal code:

- Every item is pointing to an orthogonal direction with respect to other items.
- The energy of an item is 1.

This is why we use a one-hot code, so that (A,B,C) would be encoded like (001,010,100). In this scheme, there is no ordering, the position of the bit 1

tells us which category do we have. Yet there is an exception for boolean variables, as they have only two categories that can be signaled by the presence or absence of the bit 1. Note that in the case of one-hot codes we have many bits for a symbol, therefore having B bits for a one-hot code means having B inputs in the network.

Table 4.3 describes the type of symbols we have in the label, as well as the amount of values for categorical symbols. This lets us compute the amount of input features we end up with, which is 405. Later, with the models description, it is shown how this amount is reduced to keep the current time and future information only, thus getting rid of past information and leaving us with 362 features finally.

Symbol	Type	Num. classes
p1	Categorical	35
p2	Categorical	35
p3	Categorical	34
p4	Categorical	35
p5	Categorical	35
p6	Real	-
p7	Real	-
a1	Boolean	2
a2	Boolean	2
a3	Real	-
b1	Boolean	2
b2	Boolean	2
b3	Real	-
b4	Real	-
b5	Real	-
b6	Real	-
b7	Real	-
b8	Real	-
b9	Real	-
b10	Real	-
b11	Real	-
b12	Real	-
b13	Real	-
b14	Real	-
b15	Real	-
b16	Categorical	6
c1	Boolean	2
c2	Boolean	2
c3	Real	-
d1	Categorical	47
d2	Real	-
e1	Categorical	47
e2	Real	-
e3	Real	-
e4	Real	-
e5	Real	-
e6	Real	-
e7	Real	-

e8	Real	-
f1	Categorical	47
f2	Real	-
g1	Real	-
g2	Real	-
h1	Real	-
h2	Real	-
h3	Real	-
h4	Real	-
h5	Categorical	6
i1	Real	-
i2	Real	-
j1	Real	-
j2	Real	-
j3	Real	-

TABLE 4.3: Label symbol types and amount of classes per symbol. See Table 4.1 for a description of each symbol.

As we will see soon (section 4.3), in the acoustic prediction there are two additional inputs derived from the duration model. Those inputs are the duration of the current phoneme to synthesize, and the relative position of the current frame within the phoneme. We will see in detail what do they mean exactly, but a thing to point out here is that duration is log-normalized and then the max-min (between $\{0, 1\}$) is applied. On the other hand, the relative duration is normalized by the absolute duration. Equations 4.5 and 4.6 express these operations. Note that the relative duration is computed with the absolute duration in the truth time dimension, not the log-compressed range.

$$\hat{d} = \frac{\ln d - \ln d_{min}}{\ln d_{max} - \ln d_{min}} \tag{4.5}$$

$$\hat{r_d} = \frac{r_d}{d} \tag{4.6}$$

The reason to make the log-compression will be seen with the explanation about the duration model.

4.2.4 Parallelizing the pipeline: Speeding up the data generation

The label generation is made with Ogmios from Bonafonte et al., 2006a as mentioned before. Once all text is converted to label files, which is a quick task, all WAV recordings have to be processed to get the Ahocoder (Erro et al., 2011) features. This process is time-consuming, especially given the many files of long duration available, as shown in the durations histogram in Figure 4.4.

To accelerate the Ahocoding, a parallelization approach has been taken. The data generation system is then composed of the following components:

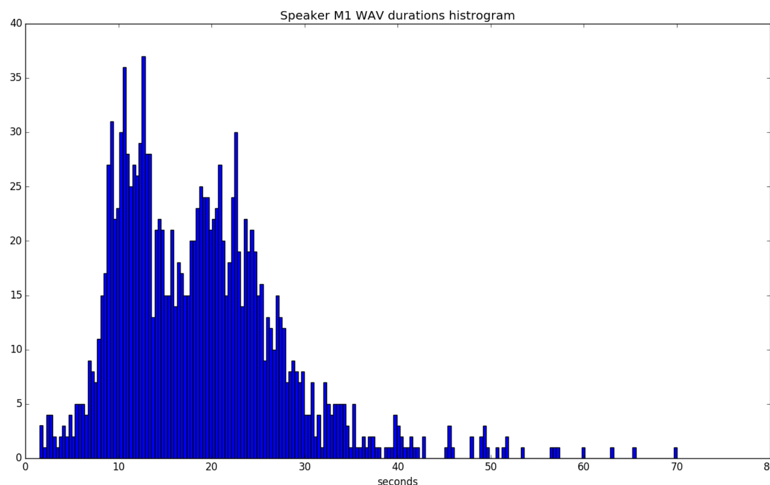

FIGURE 4.4: Histogram of WAV files' durations for speaker M1. It is shown in a scale of seconds.

- FIFO Queue: to keep the WAV files to be processed ready to go into the computation flow.
- Pipeline: A pipeline is composed of two modules:
  - WAV Trimmer: trim the first and last WAV silence regions to a maximum length of 100 ms each. This is done to avoid having an overhead of silence samples that can bias the network learning procedure, as it could get to learn how to produce the silence to much at the expense of doing worse with other phonemes. Moreover, the length of the silences at the beginning and end of recordings are more erratic than the ones inside the sentence, which were tied to a prosodic pattern.
  - Ahocoder: make the spectral estimation through Ahocoder.
- Datagen: the data generator, which converts all the resulting lab files and acoustic parameters into the tables that the TTS will process.

The data generation task is fast enough to not require parallelization, as it is only gathering all data into the table structure with some intermediate transformation (i.e. the aforementioned log-compression of some features). The architecture shown in Figure 4.5 is thus the one for accelerating the data generation process. There we can see how, having the WAV and label files, a process queue is built to hold up to N parallel ahocodings. Later, all those results are used by the data generator to build the tables and files:

- Training split for duration prediction:
  - 362 predictors and 1 prediction
- Test split for duration prediction:
  - 362 predictors and 1 prediction

- Validation split for duration prediction:
  - 362 predictors and 1 prediction
- Training split for acoustic prediction:
  - 364 predictors and 43 predictions
- Test split for acoustic prediction:
  - 364 predictors and 43 predictions
- Validation split for acoustic prediction:
  - 364 predictors and 43 predictions
- Acoustic statistics for normalization
- Duration statistics for normalization

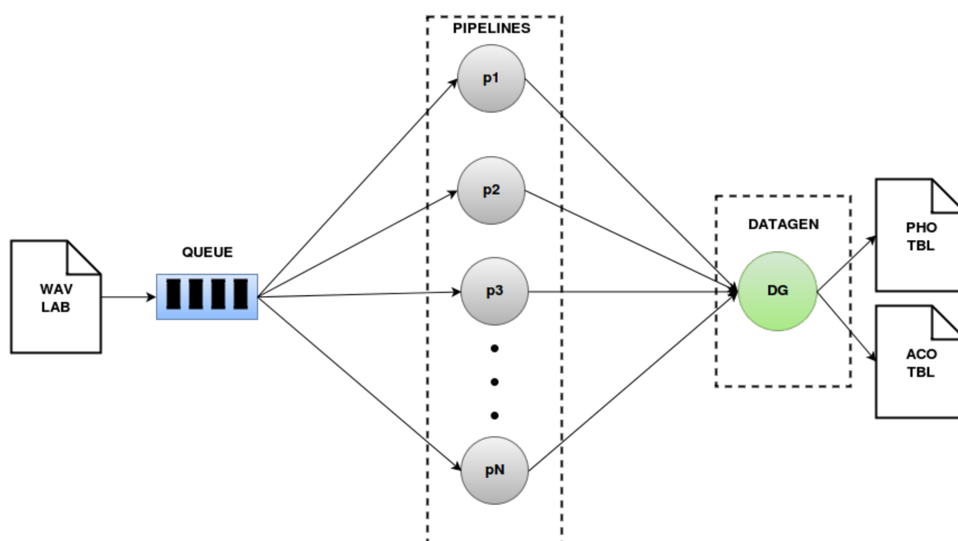

FIGURE 4.5: Data generation architecture with parallelized pipeline. Every pipeline processes a tuple (lab,wav) and accumulates the result to be given to the data generator.

Some tests have been made in order to evaluate the acceleration provoked by this system. For this, we picked the two speakers: M1 (male) and F1 (female). See section 4.5.1 for further details of the speakers. Then, we took 100 wav/label files for each speaker and ran the data generation pipelines with 1 and 32 parallel processes. The histograms for the duration of the 100-files-subsets per speaker can be seen in Figure 4.6. There we see that the mode of the histograms (both of them) is around 15 seconds, but there are many files ranging from 15 to 45 seconds.

The speed improvement can be clearly appreciated in Figure 4.7. In the most extreme case (i.e. the male voice), the elapsed time got reduced from 1 hour to 3 minutes.

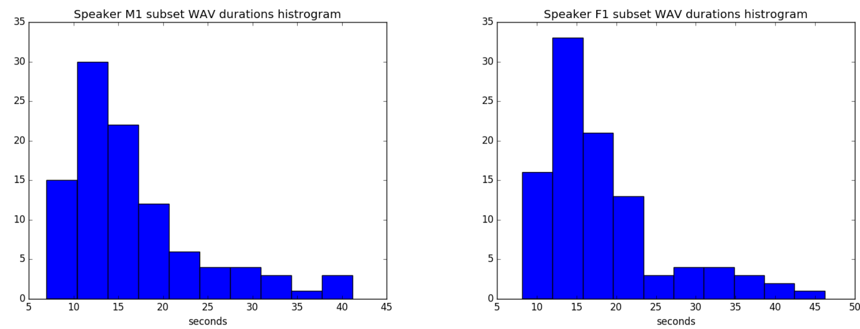

FIGURE 4.6: WAV files' durations histograms for M1 and F1 subsets of 100 files each.

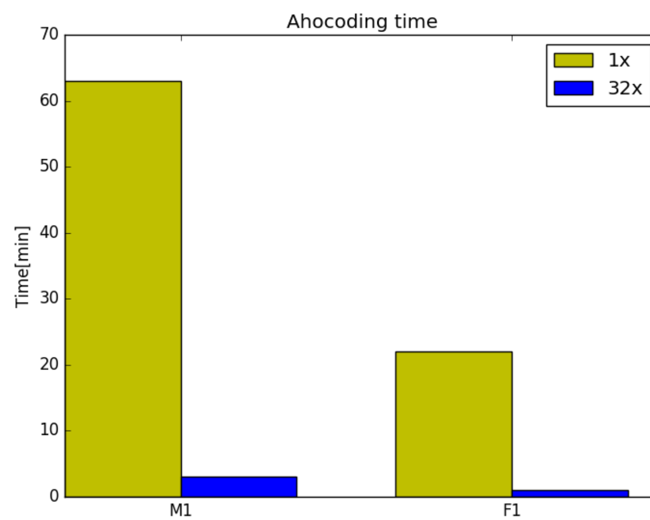

FIGURE 4.7: Results of data generation acceleration for speakers M1 and F1. M1-1x: 63 min. M1-32x: 3 min. F1-1x: 22 min. F1-32x: 1 min.

4.3 Two-stage RNN-LSTM model

The are two types of information required to produce a voice with good quality:

- The prosodic information: It considers intonation, phoneme duration, pauses between words, etc. characteristics that can make a huge effect on the voice naturalness.
- The acoustic information: Spectral estimation processed by the Vocoder system to generate the waveform. A good estimation is required for naturalness and also intelligibility.

The prosodic prediction is the first problem we tackle in this TTS design, and to be more specific we begin with special focus on the phoneme duration prediction. We basically need to know the amount of frames to be generated with the Vocoder, and then those frames will be generated out of the acoustic prediction system. This is where the "Two-stage" name comes from:

1. Predict the duration for the current phoneme out of the encoded input linguistic features.
2. Predict the acoustic frame coefficients, for as many frames as dictated by the duration prediction, also taking the linguistic features.

Beware of two things at this point: there is no specific pause prediction system, because the pause annotations are given by the front-end that generates the labels. Secondly, the F0 contour which carries the intonation behavior is not predicted at this stage but with the acoustic model.

Each of the two stages is performing a mapping, where the chosen base model for both tasks is a Recurrent Neural Network (RNN), and more concretely a Long Short Term Memory (LSTM), yet the two tasks end up having different architectures (we will see both later) as we have two different problems, despite working in a pipeline fashion.

The Recurrent Neural Network is a well suited model for these tasks, because it grants the following properties:

- It keeps track of the sequential context, as it reads the input phonemes by time-steps (phoneme by phoneme). Each time-step is thus an encoded label.
- In the case of the acoustic prediction, having a recurrent output layer improves the continuous prediction between frames (Zen and Sak, 2015), getting even better results than predicting static and dynamic features, as it was used to be done with SPSS (see Chapter 2).

In the acoustic prediction we want to work with frames (like the Vocoder does). Figure 4.8 exemplifies the hypothetical analysis for a phoneme that lasts less than 3 windows. It helps in the reasoning for the way in which

duration is predicted. It could be done by predicting the amount of frames that the current phoneme lasts, owing to the fact that the acoustic RNN will work frame-wise. Although predicting the amount of frames as an integer is a possibility, it imposes an error for the duration RNN, and it is that we show it examples biased to a quantization error. In Figure 4.8 there is a red region containing a not-current-phoneme region that fell inside the third window, and in an frame-wise duration prediction fashion that would be mistaken with a rounding error of 3 windows.

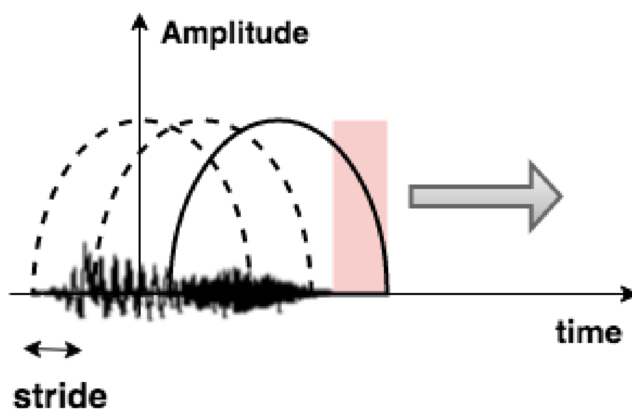

FIGURE 4.8: Vocoder sliding window example for an arbitrary phoneme. The stride is the increment in time taken by the next window $\Delta t = t_{i+1} - t_i$. The red zone shows a hypothetical empty zone out of the phoneme, so the frame duration is including information from the next phoneme or a silence. The grey arrow is the direction of the windowing analysis.

This brings the idea of making the duration RNN predict the real amount of time that the current phoneme lasts (normalized), and with this we avoid making bigger errors in the duration prediction stage produced by unnecessary round operations. Thus, the output of the duration model is a linear feed forward layer, also called Fully Connected layer (FC), such that:

$$y = \mathbf{w} \cdot \mathbf{x} + b$$

Some experiments were made to train the model with a recurrent output. This idea comes from the acoustic model, as will be seen soon, and we wanted to see if there was any improvement with this task, but there was not any besides increasing the number of parameters of the network.

Moreover, it has been mentioned how the RNN models keep track of the past context, going phoneme by phoneme, and this makes us discard some unnecessary information embedded in the classic label format. Hence features referring to past-context are removed, as commented previously in section 4.2.3, going from 405 initial features to 362. These conform what is called the linguistic inputs, which is a 362-dimensional vector that is fed into the duration RNN.

An important issue here is the way in which the duration is normalized. Previously we mentioned that there is a log-compression of the real duration per phoneme. This is to smooth the effect of the outliers that the model may encounter during training.

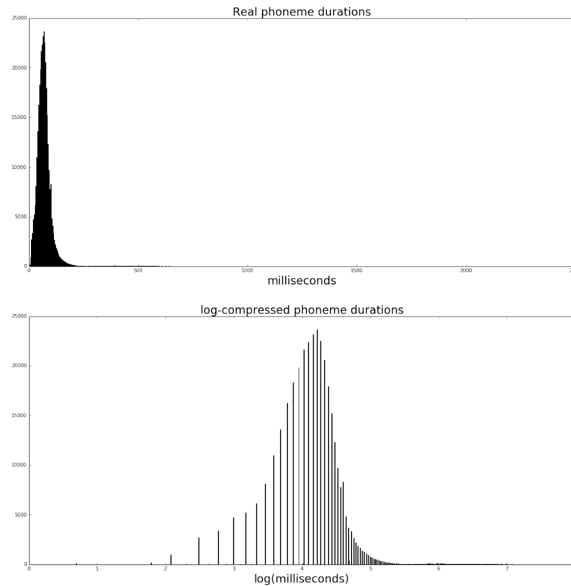

FIGURE 4.9: Duration histograms for speaker M1 phonemes. Top plot shows the real durations in milliseconds, bottom plot shows the log-compressed durations.

Figure 4.9 depicts the log-compression result in the histograms for the male speaker (M1) phonemes. In the real durations there are many examples provoking a long-tailed distribution of the data, something that distorts the regression training with the Mean Squared Error (MSE) minimization. This is why the logarithm is applied. To be more specific, the natural logarithm has been applied in all the experiments for this work. This is then finally normalized in a min-max range, as mentioned in section 4.2.3, making the values fall in the range $\{0.01, 0.99\}$. To get the duration in milliseconds during prediction we just take the exponential of the prediction, after denormalizing the min-max range.

Let's analyze why are the outliers compressed with this method: the sum of squares error that is used to train this model for the regression purpose comes from the supposition that the target data (i.e. the duration examples shown to the network for back-propagation of the errors) follows a Gaussian distribution, and the distribution of the duration values then looks closer to a Gaussian after compressing the long tail we have seen. This means that when the logarithm is applied the outliers turn into inliers. To justify this transformation we can see what would be the effect of outliers during training if we analyze the behavior of MSE in equation 4.7 (Bishop, 1995).

$$E = \sum_{t} \sum_{n=1}^{N} |\hat{y}_n(\mathbf{x}^t; \mathbf{w}) - y_n^t|^2 \tag{4.7}$$

In our case there is only one output to compute the cost so we can express it like the sum over the training samples:

$$E = \sum_t |\hat{y}(\mathbf{x}^t; \mathbf{w}) - y^t|^2 \tag{4.8}$$

Figure 4.10 shows the cost for a single sample, related to equation 4.8. Observing $|\hat{y} - y|^2$ against $|\hat{y} - y|$ we can see the effect of a big outlier by means of the derivative of the function, which is very significant, as it means a very high error and a correction towards it during minimization.

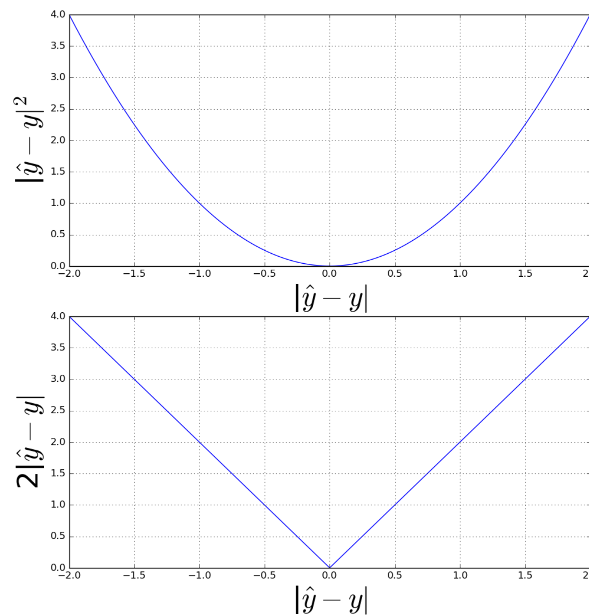

FIGURE 4.10: Plot of $|\hat{y} - y|^2$ against $|\hat{y} - y|$ and the derivative with respect to the error:$2|\hat{y} - y|$.

Figure 4.11 illustrates an example of the effect that an outlier may produce over our model, where the highly distant sample deviates the line a lot. This is why our main interest is reducing this effect avoiding the long-tailed behavior.

Now that the duration model has been discussed the acoustic model is explained, which is the second stage of the full TTS system. This model is also built with Recurrent Neural Network architectures, as mentioned previously, yet the output layer is also recurrent in this case as aforementioned. The normalization of the features for this stage is a min-max for all the outputs, which are:

- Mel Cepstral Coefficients.
- Voiced Frequency.
- log-F0 contour.
- Voiced/Unvoiced flag.

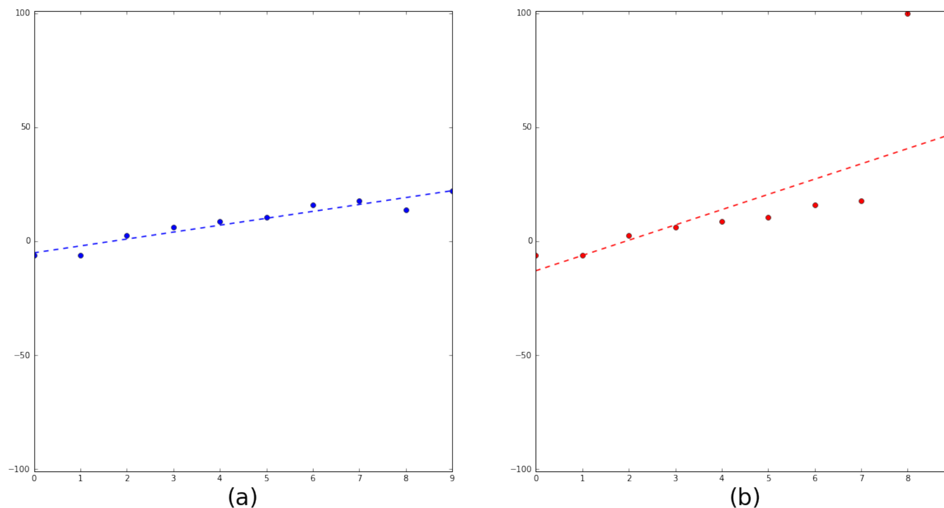

FIGURE 4.11: Example of a regression estimation where we see how the outlier produces a high deviation over the correct estimation. (a) Gaussian distributed data with $\mu = 0$ and $\sigma = 4$ over the line $y = 1.2x$. (b) Outlier artificially inserted to the data in (a).

We have seen that the F0 contour is one of the acoustic parameters to be predicted, but this parameter has a special behavior depending on whether the current frame contains voiced or unvoiced sounds. For voiced frames it behaves as a continuous signal, but for unvoiced frames its value is zero (thus indicating that there is no periodic behavior in the frame). There is an example of pitch contour in Figure 4.12 to observe this mixture of discrete and continuous values. However, the network predicts the $\log$ of the F0 at each frame, because it is the type of data with which the Ahocoder deals.

It is then important to get rid of the discrete symbol somehow to be able to normalize the pitch with the min-max range, because we will have a very frequent outlier with a very large negative value (Ahocoder encodes the unvoiced symbol with a value of -1000000 to indicate it goes to $-\infty$). Otherwise, if we had the unvoiced symbol in the training set, the min-max operation would compress all the continuous values to a tiny range, which wouldn't let the network learn the real distribution of the pitch. The way to solve this is by means of an interpolation of the original pitch (also shown in Figure 4.12), where the following operation has been performed during the unvoiced frames:

- If the unvoiced frame is at the beginning of the contour where no previous voiced value appeared, put the first next voiced F0 value as a constant value
- If the unvoiced frame is at the end of the contour where there are no more voiced values put the last previous voiced F0 value as a constant value (inverse of the first operation)
- In all intermediate unvoiced regions, interpolate linearly in the $\log$-domain between right previous and right next voiced values. Equation 4.9 shows the operation.

$$\log F_0^i = \log F_0^p + (\log F_0^n - \log F_0^p) \cdot \frac{i-p}{n-p} \tag{4.9}$$

Where n is the next voiced value's frame index, F_0^n is the next voiced value, p is the previous voiced value's frame index, F_0^p is the previous voiced value and F_0^i is the i-th pitch value we want to get at frame index i.

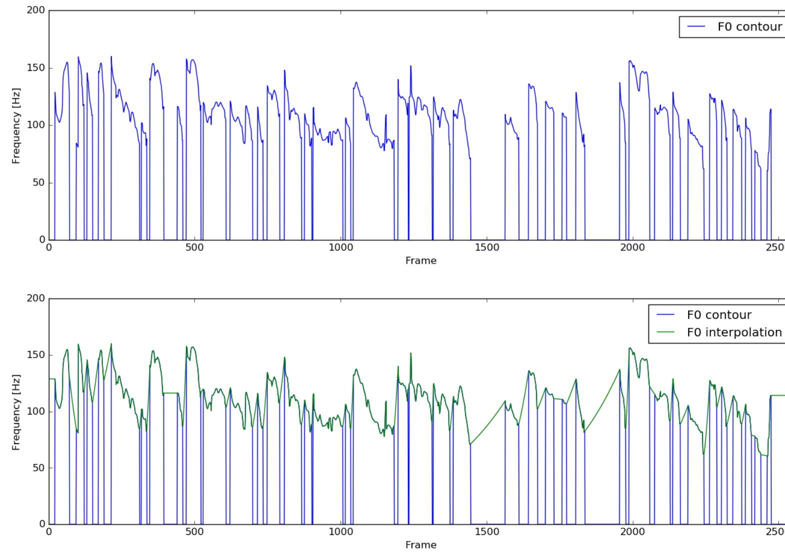

FIGURE 4.12: Top: F0 contour example for male speaker M1. Bottom: F0 contour (blue line) example for male speaker M1 with interpolation contour (green line).

On the other hand, the acoustic model must determine when is it that there is an unvoiced frame in order to mask out the interpolated values, which are false information given for the sake of good learning. There is then the voiced/unvoiced flag for this task, and it is a learned output as well.

Finally, all other parameters are normalized in a min-max fashion with the reduced range $\{0.01, 0.99\}$, but the Voiced Frequency parameter (discussed in section 4.2.2) is also $\log$-compressed to avoid a long tailed distribution. The original distribution of values can be appreciated in the histogram in Figure 4.13.

In order to predict all the acoustic parameters we use the same linguistic inputs as before to take into account not only the phoneme, but also the context that surrounds it. In addition, we must take care with the duration that has already been predicted, so that it becomes an input of the acoustic stage with a min-max normalization, as expressed in equation 4.5. There is also an input (the relative duration) expressing what is the current position in the predicted frame duration. This is a normalized index computed in equation 4.6, so that the network has a clue about in which part of phoneme is it generating contents at every time-step. It is at every time-step that this value gets updated with the same stride with which the Ahocoder encoded the data, normalized by the full phoneme duration. Both of these duration features are concatenated to the linguistic inputs and then fed into the acoustic model.

The general scheme of the two-stage TTS system is shown in Figure 4.14. As introduced previously, vanilla Recurrent Neural Networks do not tend

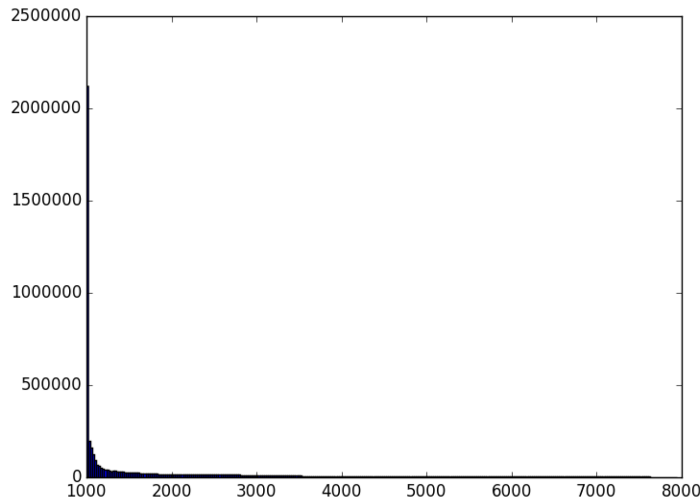

FIGURE 4.13: Histogram of voiced frequency values in the training data.

to work very well to find long-term dependencies, and this is why we use LSTM in this model.

The acoustic model has some Embedding (EMB) layers at the input. These are Fully Connected layers with $\tanh$ activation functions, which may help with the first mapping of the mixed sets of features better than injecting all directly to the LSTM layer.

The output layer is an LSTM layer with 43 output cells to preserve a smooth transition between acoustic predictions (as mentioned previously).

Finally, a comment on how many time-steps happen for a utterance might be clarifying for the reader, which basically shows approximately the amount of time spent during generation (we don't consider other de-normalizing computations nor possible roundings to go from time to frames): In the duration prediction stage there is one time-step per phoneme (up to N phonemes). In the acoustic prediction stage each phoneme has a duration t_n in frames. The total number of time-steps can then be estimated as:

$$T_{gen} \approx N + \sum_{n=0}^{N-1} t_n \tag{4.10}$$

4.4 A post-processing technique for better naturalness

This section is about an issue inherent to the network learning mechanism which worsens the quality of the generated speech with respect to the natural one. When doing regression, the network is trained with the MSE criterion, which is supposing a Gaussian distribution of the output data (as seen earlier in this chapter) and which tends to predict the mean of the distribution. Hence, the network does not take into account the variance of the distributions at the output, such that it cannot capture the variability effect

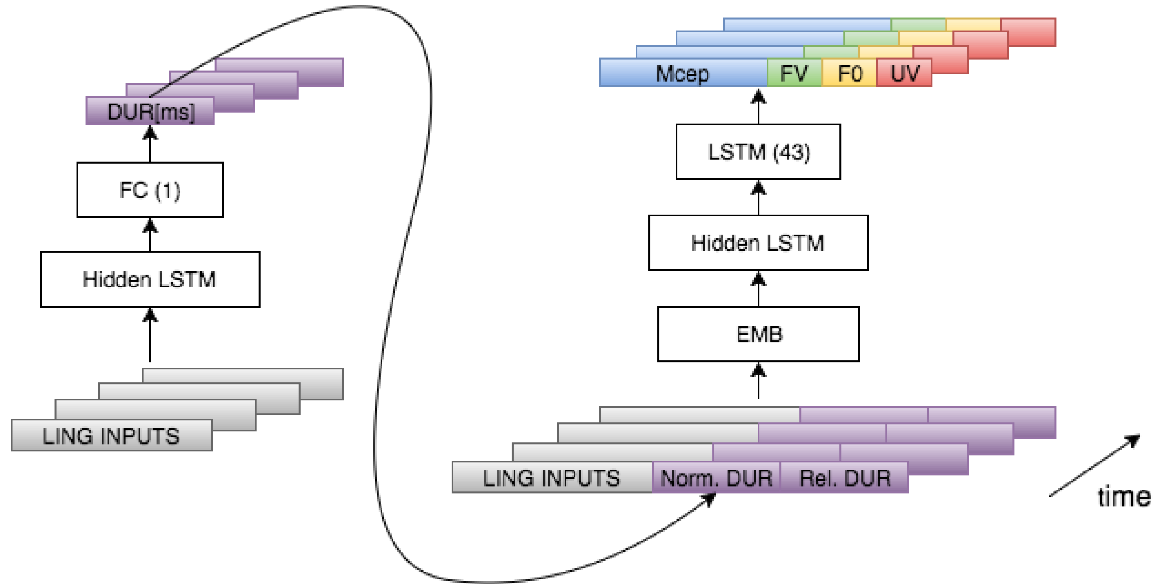

FIGURE 4.14: Final setup of the architecture, where duration prediction and acoustic prediction work together in a pipeline fashion.

(or noisy effect) of the natural speech, which turns out to be an important factor to achieve a good level of naturalness.

To see this effect produced by the network more clearly the following analysis is carried out: The full training set is inferred through the acoustic model to get the predicted acoustic parameters, and then for each phoneme in the set we compute the ground-truth cepstral standard deviations of each coefficient, and we do the same for the predictions. Beware that the pause phoneme "pau" is discarded. Then for a certain phoneme equation 4.11 is applied to obtain the curves shown in Figure 4.15. There it is clear how, as the coefficient order increase, the quotient of standard deviations also gets bigger values for all the phonemes, which means that the network makes predictions with less variability for each i increment. The best we could get would be a flat behavior with value 1, so that the network predictions would have the same variability as the real data. It does not happen though, and this is something to be corrected by means of a post-filtering technique.

$$\frac{\sigma^{i}_{ground-truth}}{\sigma^{i}_{prediction}} \qquad \text{for} \qquad i = \{0, 1, \cdots, 39\} \tag{4.11}$$

Post-filtering then increases the variance of the network predictions, thus getting closer to the original speech characteristics. Erro (2016) proposed a post-filtering technique that enables the application of different post-filtering factors to low and high frequencies. In this work we apply another approach with which we achieve already better naturalness (compared to the original speaker) when we post-filter the network prediction with some

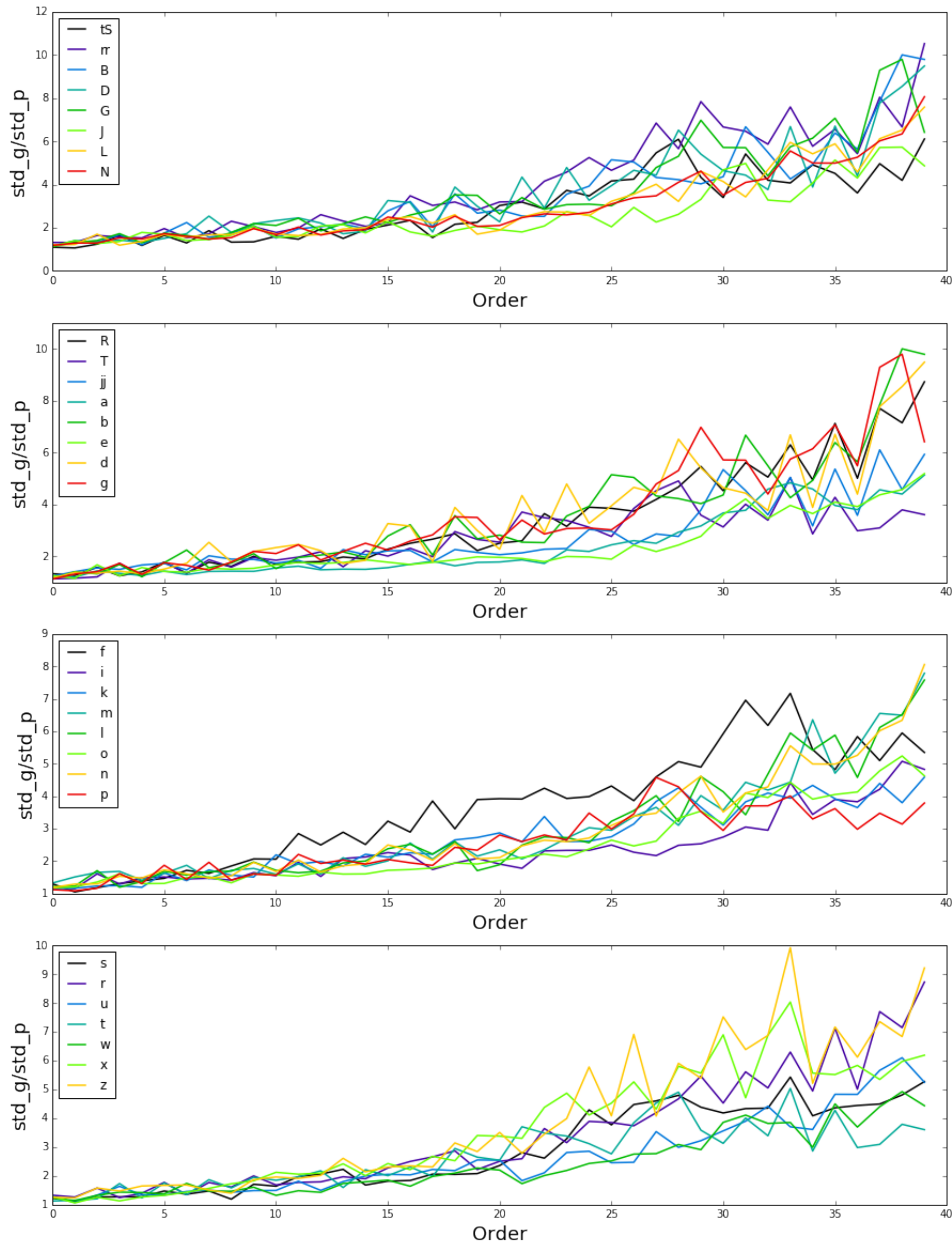

FIGURE 4.15: Ratio $\frac{\sigma_g^i}{\sigma_p^i}$ per phoneme and coefficient, where σ_g^i stands for ground-truth standard deviation at i-th coefficient, and σ_p^i stands for prediction standard deviation at i-th coefficient. There are 32 phonemes analyzed.

multiplicative factor p_i over each predicted cepstral coefficient c_i. The factor then increases the variance for the i-th coefficient (with $p_i > 1$). This is achieved with a multiplicative correction curve applied with the following methodology based on Sorin, Shechtman, and Pollet (2011):

$$\begin{aligned} \hat{c_0} &= c_0 \\ \hat{c_i} &= c_i \cdot p_f^i \quad i = \{1, \cdots, 39\} \end{aligned} \tag{4.12}$$

Where p_f is the post-filtering factor, set finally to 1.04 empirically based on our preliminary perceptual tests, and the first cepstral coefficient is not modified, but the following 39. Figure 4.16 depicts the comparison between some post-filtering curves and the geometric mean of the curves shown in Figure 4.15. There it is clearly depicted how our perceptually chosen curve with $pf = 1.04$ is the down-side envelop of the geometric mean, thus never going over the quotient.

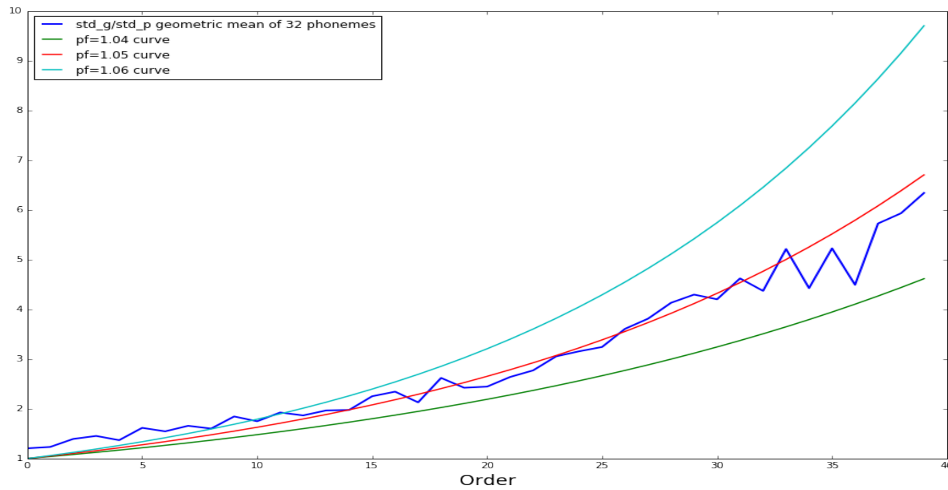

FIGURE 4.16: Geometric mean of $\frac{\sigma_g}{\sigma_p}$ compared to the post-filtering curves for values $pf = 1.04$, $pf = 1.05$, $pf = 1.06$.

Figure 4.17 shows the result of applying the aforementioned post-filtering to the curves in Figure 4.15, and then the geometric mean is computed for comparison purposes with the one shown in Figure 4.16. The resulting curve is much flatter than the one without post-filtering, thus making the network make predictions with a similar variance per output cepstral coefficient.

The results can also be appreciated in time for some cepstral coefficients, so for instance the 5-th, 10-th and 15-th are analyzed for a couple utterances, taking approximately the first 600 frames and comparing them to the ground-truth frames. Figures 4.18, 4.19 and 4.20 show how the post-filtered signals achieve a better representation of the natural signals.

4.5 Experimental Design and Results

Now that all the properties of the model have been discussed results are shown and explained, such that we can have an idea of the performance

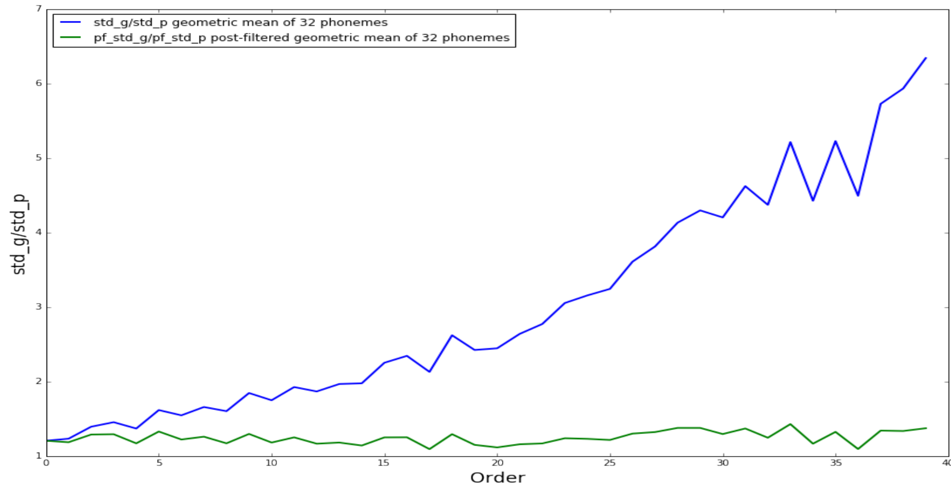

FIGURE 4.17: Geometric mean of $\frac{\sigma_g}{\sigma_p}$ before and after post-filtering with $p = 1.04$. Green: post-filtered. Blue: raw prediction.

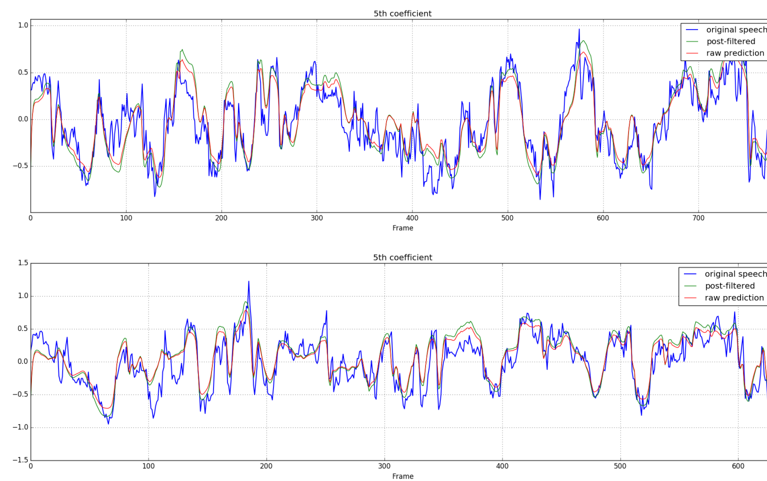

FIGURE 4.18: Evolution in time of the 5th cepstral coefficient for two test files. Blue: natural speech. Green: post-filtered prediction. Red: raw prediction.

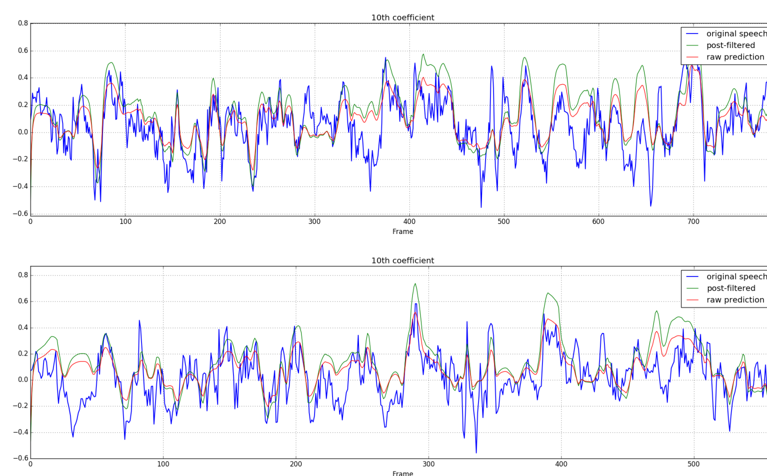

FIGURE 4.19: Evolution in time of the 10th cepstral coefficient for two test files. Blue: natural speech. Green: post-filtered prediction. Red: raw prediction.

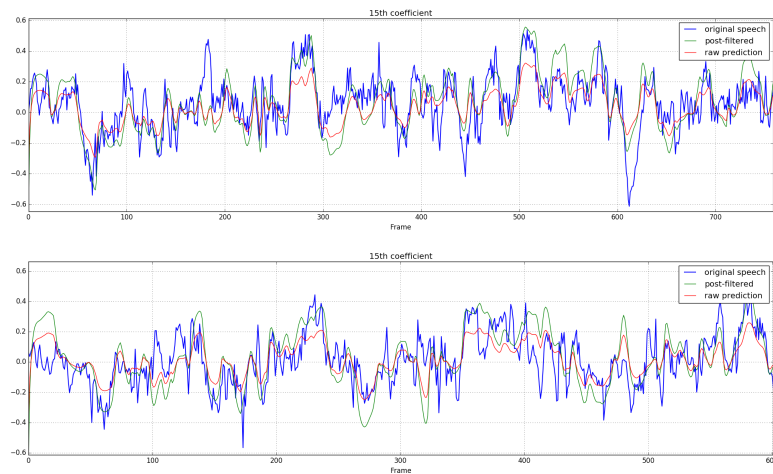

FIGURE 4.20: Evolution in time of the 15th cepstral coefficient for two test files. Blue: natural speech. Green: post-filtered prediction. Red: raw prediction.

of the model, also comparing the differences in changing the topology of the network in both the duration and the acoustic stages. Changing the topology means trying to find a good combination of number of layers, neurons, learning rate, etc. which are called the hyper-parameters of the model, as introduced in Chapter 3. Sometimes this process might require a bit of intuition and patience, owing to the fact that nowadays the number of hyper-parameters can be huge depending on the model we chose, but a reduced set of choices can lead to a good result. Hence, a set of different models will be tested in section 4.5.2 and some objective measurements will be taken on the resulting predictions to compare the relative performance between the different options. This process will finally lead us to obtain the subjective results for the chosen final architecture.

All the models were developed with the Keras (Chollet, 2015) framework, which is an object oriented library for prototyping Deep Learning models quickly.

4.5.1 Database

All the experiments were performed with the so called M1 speaker of the TCSTAR (Bonafonte et al., 2006b) database. This is a professional male speaker reading different types of contents. From that speaker, a total of 1453 utterances were taken to be processed as labels. This means we have approximately 8 hours of speech for this speaker's experiments, which turns to be an amount of 286K duration examples and ≈ 4M acoustic examples. Also, the following data splits were defined:

- Training: 80% of total data
- Validation: 10% of total data
- Final Test: 10% of total data

Performing a k-fold Cross Validation is an attractive solution for obtaining good data splits and more confidence in the results, but it seems unfeasible

to do this and also test different architectures with the current computation time required by our models.

4.5.2 Objective Evaluationt

To see the effect of different architectures we made an objective evaluation by means of specific metrics for each kind of predicted feature. In the case of duration, we chose the Root Mean Squared Error (RMSE) in millisecond scale, as it is a well-accepted error measure in the literature for speech synthesis tasks, defined as:

$$RMSE\,[ms] = \sqrt{\sum_{t=0}^{N-1} (dur_t - \hat{dur_t})^2} \tag{4.13}$$

where N is the number of test phonemes used for the evaluation. It is important to mention that silence phonemes were predicted but not included in the evaluation computations because they have a large variance and they would distort the prediction of the regular phonemes duration. Table 4.4 shows the result for different architectures. All those models had Dropout between the Output Layer and the previous Hidden Layer with $p = 0.5$ (probability of being activated). Also, when the output layer is recurrent (LSTM) the activation is sigmoid (the outputs are normalized to work in the linear region) to gain stability whilst training. Moreover, the batch size was 64, the maximum sequence length to propagate the gradient through time was 10 time-steps, the optimizer was Adam (see Chapter 3), with learning rate $lr = 0.001$ and, during training, a validation set was used to do early-stopping, having a maximum of 100 epochs per model. The early-stopping mechanism had a patience of 10 epochs per model, which means that if the validation loss does not improve at a certain point during training along 10 epochs, the training is aborted and the best validated model is the one stored, and it is also the one with which results are computed.

Embeddings	Hidden LSTM	Output Type	Num. params.	RMSE [ms]
0	**1×256**	**FC**	**634K**	**18.51**
0	1×64	FC	110K	18.71
0	1×256	LSTM	635K	18.73
1×256	1×256	FC	618K	18.77
2×256	1×512	FC	1.7M	18.95
2×128	1×256	FC	457K	19.01
0	1×64	LSTM	110K	19.06
0	1×1024	FC	5.7M	19.34
0	2×256	FC	1.1M	20.04
0	2×256	LSTM	1.2M	22.77

TABLE 4.4: Comparison of different architectures for the duration model with their objective result. FC: Fully-Connected layer. The best performing model is in bold text.

We can see how increasing the number of parameters tends to give worse results when we go over 256 for our search, probably owing to over-fitting

effects as the amount of data is not very high. Also, going under this amount of units does not provide better result either, thus losing some representation capabilities for the provided data, though the difference is not high. Even though, we picked the best performing model as we considered it has a reduced set of parameters considering the state of the art models. Some embedding layers were also tested with no success for this model. The same happened when inserting a recurrent output. It is worth mentioning that this is not the most exhaustive search that could be done, but it is considered to be good enough to get a first approach for the purpose of this work.

Regarding the acoustic model other tests were performed for the same purpose, and the results can be seen in Table 4.5. In this case we have the following metrics for each kind of predicted feature:

- Mel Cepstral Distortion (MCD) for the MFCC predictions in Decibel [dB] scale.
- RMSE of the F0 prediction in Hertz [Hz] scale.
- Accuracy metric for the Voiced/Unvoiced flag prediction in $\%$.

Some authors claim that the MCD (Mashimo et al., 2001) is correlated to subjective evaluations (Kubichek, 1993) and it is defined as:

$$MCD = (10\sqrt{2})/(T \ln 10) \sum_{t=0}^{T-1} \sqrt{\sum_{n=0}^{39} (c_{t,n} - \hat{c}_{t,n})^2} \tag{4.14}$$

where T is the number of test frames, $c_{t,n}$ are the real cepstral coefficients and $\hat{c}_{t,n}$ are the predicted cepstral coefficients. The MCD is computed without applying post-filtering. The RMSE of the F0 is computed as the duration one:

$$RMSE\,[Hz] = \sqrt{\sum_{t=0}^{T-1} (f_{0t} - \hat{f}_{0t})^2} \tag{4.15}$$

Finally, we consider the Accuracy of the Voiced/Unvoiced flag prediction, which is defined as:

$$Acc[\%] = \frac{TP + TN}{TP + FP + TN + FN} \times 100 \tag{4.16}$$

Where TP stands for True Positives, TN are the True Negatives, FN are the False Negatives and FP are the False Positives.

The silence frames were also dropped for the acoustic evaluation, and the training conditions for all the acoustic models were the same as in the duration models except that the batch size was 256 and the maximum sequence length to back propagate the gradients was 30. Here we can see how including some embedding layers decreases the error for some metrics, possibly due to the mix of data types that we have at the inputs (duration and contextual features). Also, we can see how increasing the number of cells and

layers gives the lowest results, as we have way more examples than the ones for the duration model (see section 4.5.1). Nonetheless, it is clearly seen that there is no much variation in the acoustic prediction results for the tested parameters. We ended up picking the one with lowest error in all measurements again, like in the duration model search, and this fits perfectly the purpose of this work as the subjective results will suggest.

Embeddings	LSTM	Params.	F0 RMSE[Hz]	MCD [dB]	UV Acc[%]
2×256	**2×512**	**3.9M**	**14.90**	**6.09**	**95.00**
2×256	1×512	1.8M	14.90	6.17	94.86
2×256	2×256	1.2M	15.00	6.09	94.86
0	2×256	1.2M	15.03	6.22	94.60
2×256	1×256	736K	15.22	6.18	94.96
3×512	1×256	1.55M	15.99	6.29	94.70

TABLE 4.5: Comparison of different architectures for the acoustic model with their objective results. LSTM: Hidden LSTM layer and units. Params: Number of parameters of the network. Embeddings: Number of input Fully Connected layers for first projections of the data. F0 RMSE: Root Mean Square Error of the F0. MCD: Mel Cepstral Distortion. UV Acc: Accuracy of Voiced/Unvoiced flag prediction. The best performing model is in bold text.

4.5.3 Subjective Evaluation

Objective tests performed well to compare different architectures and to get some clue of how well does the TTS work. To get a more in-depth evaluation of the model however, a subjective is conducted to evaluate the naturalness of the TTS developed in this work and also to make a comparison with the other currently used TTS techniques. The platform and synthesized examples can be listened in *UPC TTS Benchmark* (2016). In the test 17 listeners were given 5 sentences to each one, randomly selected from a set of 15 short sentences. For every sentence, the listeners evaluate 6 different versions generated with the following 6 different systems:

- Natural voice: original speaker recording.
- Ahocoded version: the original recording was parameterized with the Ahocoder and constructed back into the waveform to evaluate the amount of naturalness that the developed TTS loses already in the waveform generation part, which is extrinsic to the neural network.
- LSTM raw: Raw prediction from the two stage LSTM TTS without applying any post-filtering.
- LSTM pf: Prediction from the two stage LSTM TTS with a post-filtering factor of $pf = 1.04$.
- US: Unit Selection system of UPC (Bonafonte et al., 2006a).
- SPSS: Statistical Parametric Speech Synthesis generated with the HTS framework (Zen et al., 2007).

The users are then asked to evaluate the hidden natural voice with a 100 and to set the other ones inside the $0 - 100$ scale of naturalness (thus they have to set values relative to that of the natural voice) (ITU-R, 2003). The participants can listen the different recordings as many times as required to make comparisons between the different systems.

All the TTS systems predict the duration, pitch and the acoustic parameters. As opposed to the acoustic objective evaluations, here there is no forced alignment to evaluate the full TTS capability. In order to proceed with the comparisons a normalization technique was applied to all the recordings based on ITU-T P.56 Recommendation (ITU-T, 2011). In this way all files are equalized to have the same power level and we make sure that there is no difference because of a masking effect over noise artifacts (for instance caused by the multiplying factor of the post-filtering curve, that would increase the energy of the signal otherwise).

Figure 4.21 and Table 4.6 show the results of the subjective test. Figure 4.21 is a box-plot, where each system's data distribution is drawn in quartiles. The boxes extension is the distance between the first and the third quartiles, and the red line depicts the median of the data. The vertical dashed lines extending up and down from the boxes are called the whiskers. In the edge case that the first and third quartiles are equivalent, the whiskers extension will be set to the min-max range of values in the distribution (for instance in the Natural case). Otherwise, the whiskers show the range 1.5 times the first or third quartile, as in the Ahocoded case. The points outside of the whiskers range are outliers found in the distribution of the data resulting from evaluation.

In the box-plot we can see that the results are good for the TTS developed in this work, as it is the one right after the Ahocoded voice in terms of naturalness. We can note there that the post-filtering technique indeed improves the perception of naturalness, thus confirming the effectiveness of this method. It is important to see that the Ahocoder already introduces some artifacts that diminish the naturalness of speech, getting a mean evaluated value of 82.65. The Ahocoder score is the best that our model could achieve, so we set it as a reference for main comparisons. The two stage TTS system with post-filtering achieves a mean value of 60.80, so 21.85 points is the mean loss in naturalness introduced by the neural network with respect to the Ahocoder. In addition to this, we can also see that the mean value of the TTS without post-filtering is 47.13, which means that the mean improvement achieved by adding the post-filtering mechanism is 13.70.

The case of Natural speech is the one with highest naturalness, as expected. Actually, the distribution is so condensed near 100 that the quartiles coincide.

In the case of US system we can see how the variance is quite higher than in other systems, and this is mainly because the US performs quite well for very studied contexts in the dataset of speech samples, but when there is a rare context appearing in the test text to be synthesized the system performs a lot worse than normally (Hunt and Black, 1996), and thus the resulting variance in the evaluation is higher because of those noticeable failures.

The SPSS used for this comparison performed clearly worse, but it is important to mention that no post-filtering methodologies were used in this

case to combat the over-smoothing effect that also appears in HMM-based synthesis.

For both US and SPSS systems the already available systems were used, but perhaps they could be optimized to get a closer result to that of our optimized TTS.

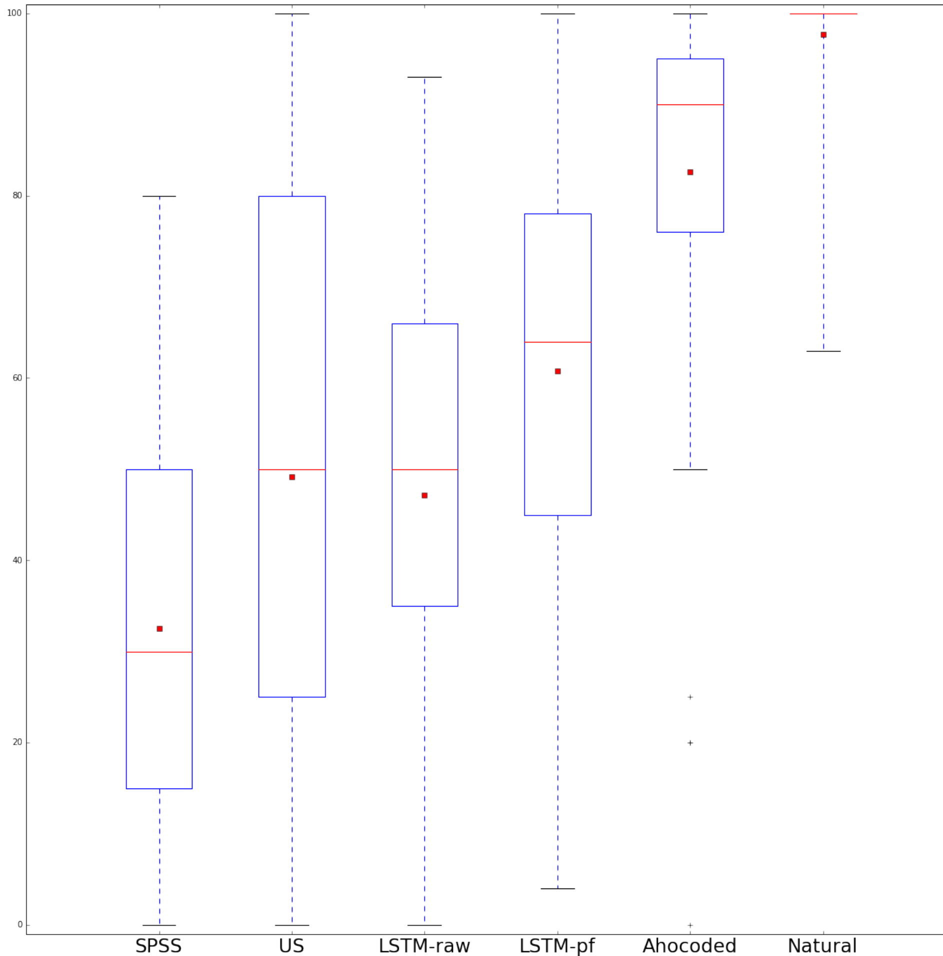

FIGURE 4.21: Box-plot of the subjective naturalness test results relative to real human voice. SPSS: Statistical Parametric Speech Synthesis. US: Unit Selection. LSTM-raw: Two-stage TTS without post-filtering. LSTM-pf: Two-stage TTS post-filtered with $pf = 1.04$. Ahocoded: Natural speech parameterized with the Ahocoder and reconstructed. Natural: real human voice. Red line: median. Red dot: mean.

System	μ	σ
Natural	97.69	6.61
Ahocoded	82.65	19.85
LSTM raw	47.13	24.63
LSTM pf 1.04	60.80	22.20
US	49.20	30.80
SPSS	32.54	21.21

TABLE 4.6: Statistics of the subjective results for the 6 systems.

4.5.4 Gate activations analysis

Once the acoustic model is trained we can look at its inner structure to check what is activated through time depending on the input sequence of phonemes. Here an analysis of the input, output and forget gate activations is shown graphically for an arbitrary test file (see section 3.6 in Chapter 3 for a description of the gating mechanisms). The chosen sentence is *"Llamó desde la recepción con su voz lúgubre."*, where the first LSTM hidden layer and the output layer gates have been analyzed. Figures 4.22, 4.23 and 4.24 depict the heatmap of activations of 20 LSTM cells in the hidden layer. These plots can look confusing as we see a lot of activities happening through the different cells (plus only 20 cells are shown from the total amount of 512, so we don't have the full picture for better visualization purposes). In order to get a closer look at what's going on, Figure 4.25 illustrates the averaged activities for the three types of gates over all the hidden units.

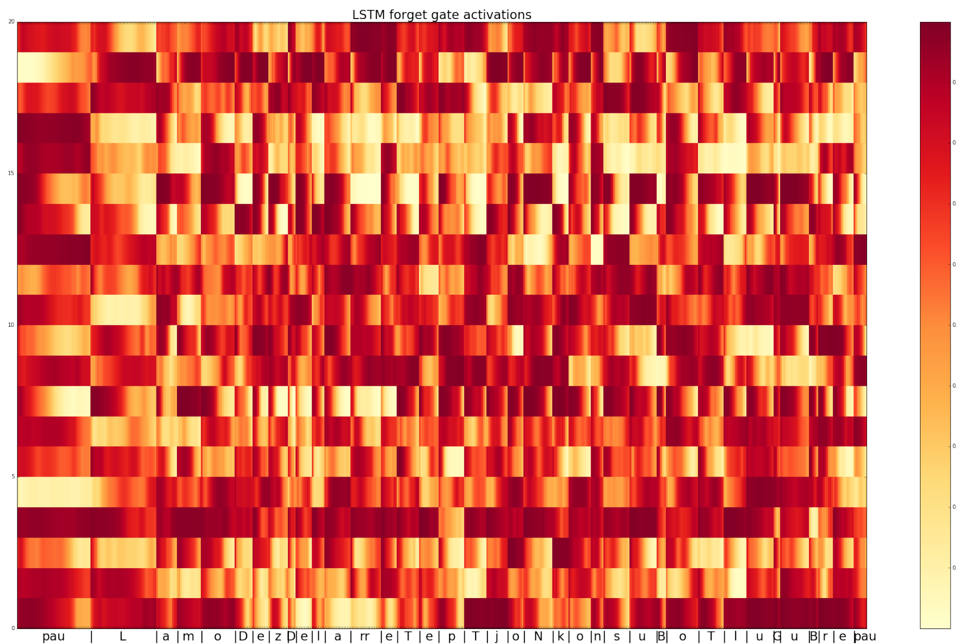

FIGURE 4.22: Activations map for the forget gates of the first 20 hidden cells in the first LSTM hidden layer. Red regions are high activations (thus preserve the past) and yellow means forget the past.

Something quite interesting can be observed in Figure 4.25: the forget gate removes more content as the input phoneme is still repeating, though it is done in a patient manner (close to a linear behavior). However, when a new phoneme gets into the network there is a peak in the gate, thus it preserves as much as possible the information about its past. On the other hand, the input gate follows what looks like the inverse behavior, meaning that it lets in as much new information as possible (maybe to try to get new clues when the changes in input features are not quite relevant). This "inverse" behavior with respect to the forget gate was already mentioned in the work by Wu and King (2016). There the authors proposed an interesting efficient recurrent architecture based on this fact, where they only keep a forget gate as a memory control mechanism and get rid of the output gate,

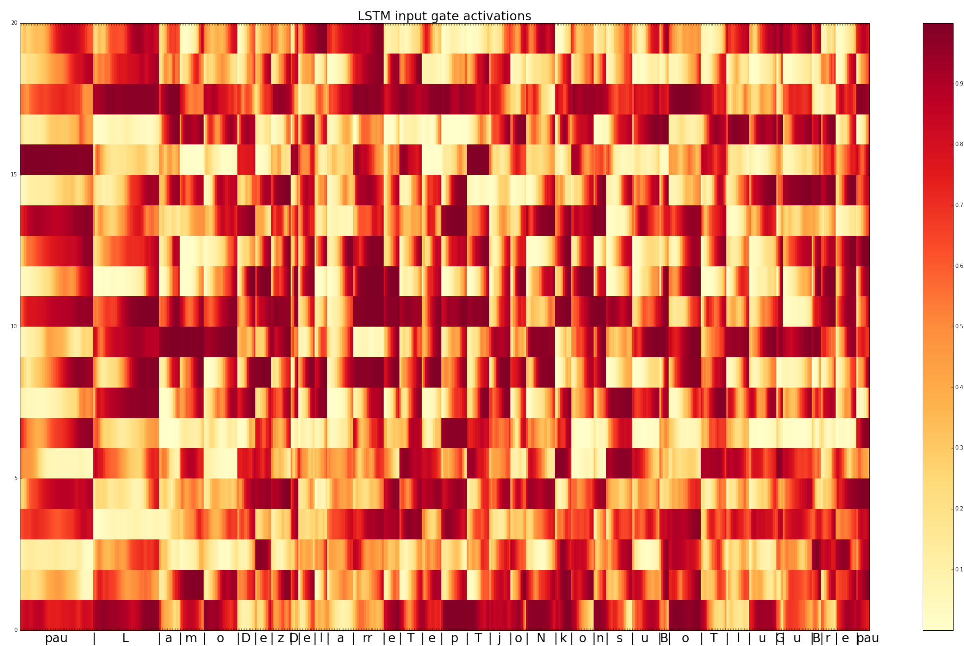

FIGURE 4.23: Activations map for the input gates of the first 20 hidden cells in the first LSTM hidden layer. Red regions mean updating the cell state a lot with the new candidate. On the other hand, yellow means not letting the information in.

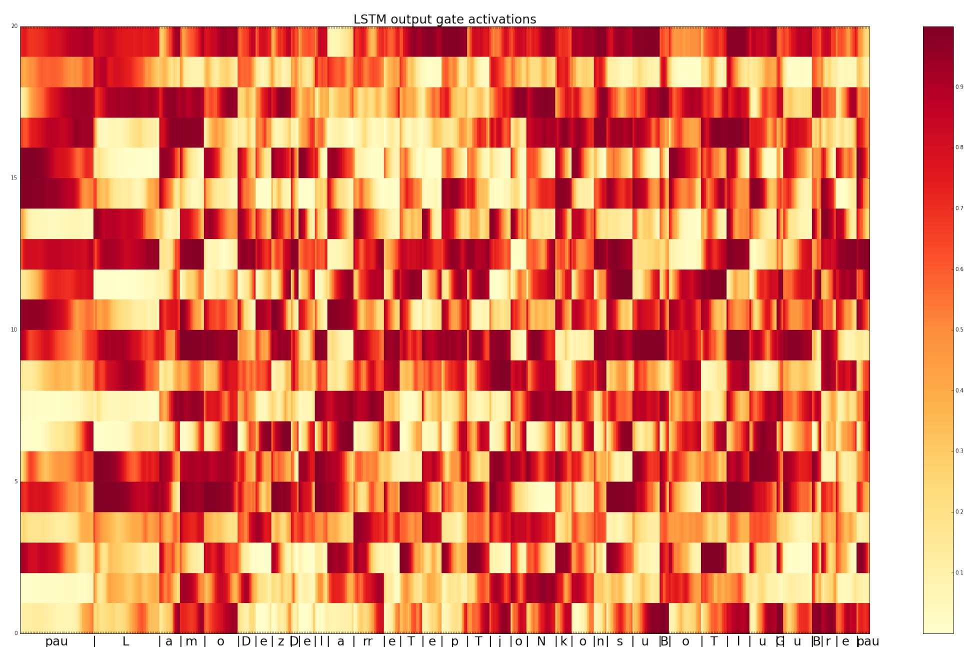

FIGURE 4.24: Activations map for the output gates of the first 20 hidden cells in the first LSTM hidden layer. Red regions mean letting the cell state flow out of the cell.

but preserving a special type of input gate which is: $\mathbf{i}_t = 1 - \mathbf{f}_t$, whilst the forget gate $\mathbf{f}_t$ is still parameterized with the set of learnable weights.

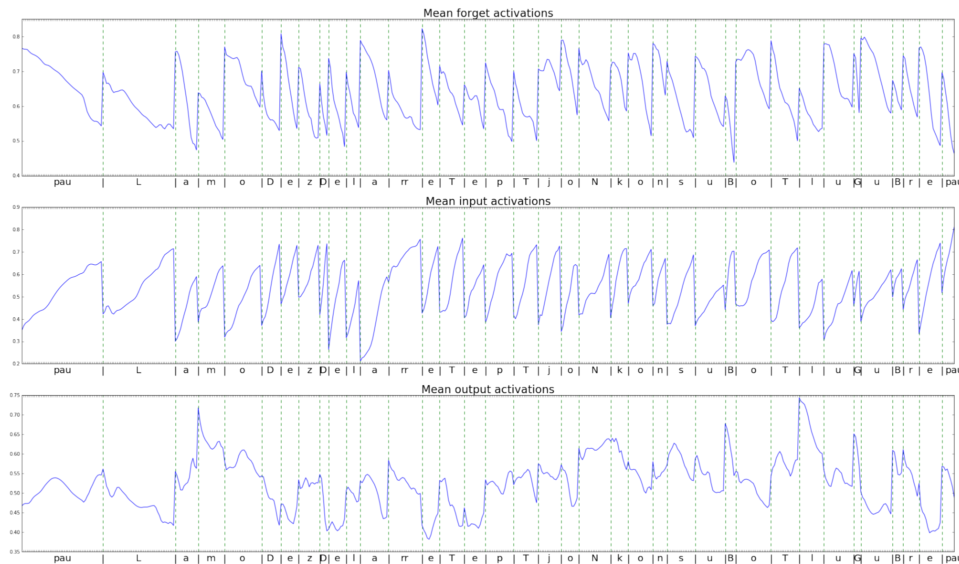

FIGURE 4.25: Average activations for the first LSTM hidden layer with 512 cells. Input gate, forget gate and output gate are shown. Green dashed lines are the phoneme boundaries.

In Figure 4.26 it is shown the comparison between the learned input gate activations $\mathbf{i}_t$ and the type of input gate proposed in Wu and King (2016) $1 - \mathbf{f}_t$. There we can indeed see that the curves are correlated.

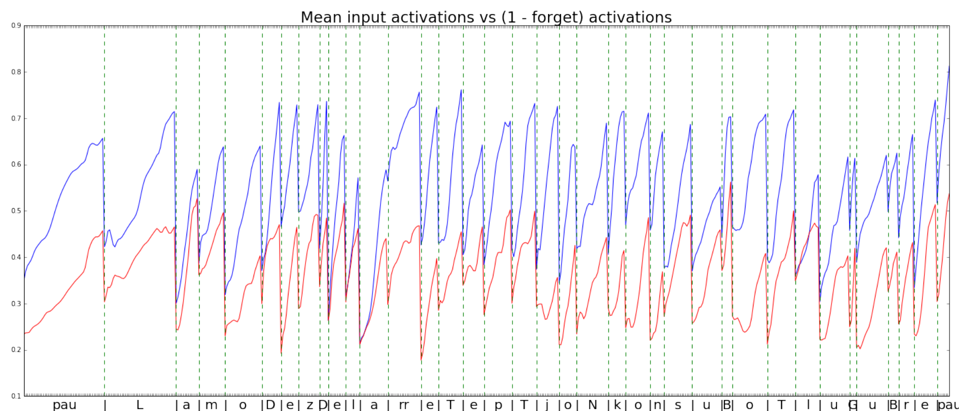

FIGURE 4.26: Averaged activations of the input gates for the first hidden LSTM layer in blue line and $1 - \mathbf{f}_t$ average activation in red.

The same analysis can be done in the output recurrent layer, where a different behavior is observed. The heatmaps for the 43 output LSTM cells gate activations are shown in Figures 4.27 4.28 and 4.29. There it is depicted the behavior of the cepstral predictions (the first 40 rows beginning at the bottom of the figure), the voiced frequency, the pitch and the top row is the voiced/unvoiced flag. In the three heatmaps it can be appreciated how the voiced/unvoiced prediction get extreme values (either close to one or to zero) as expected, and an important thing to note is that those values match their value to the type of phoneme pretty accurately, so for instance phonemes like "pau", "T", "k", "s", that are unvoiced, get a low level in the

output gate which is blocking the cell output to not make a prediction there. Also, a closely related behavior happens in the voiced frequency value as expected, which has a wider range of intermediate values but is correlated to the type of phoneme injected.

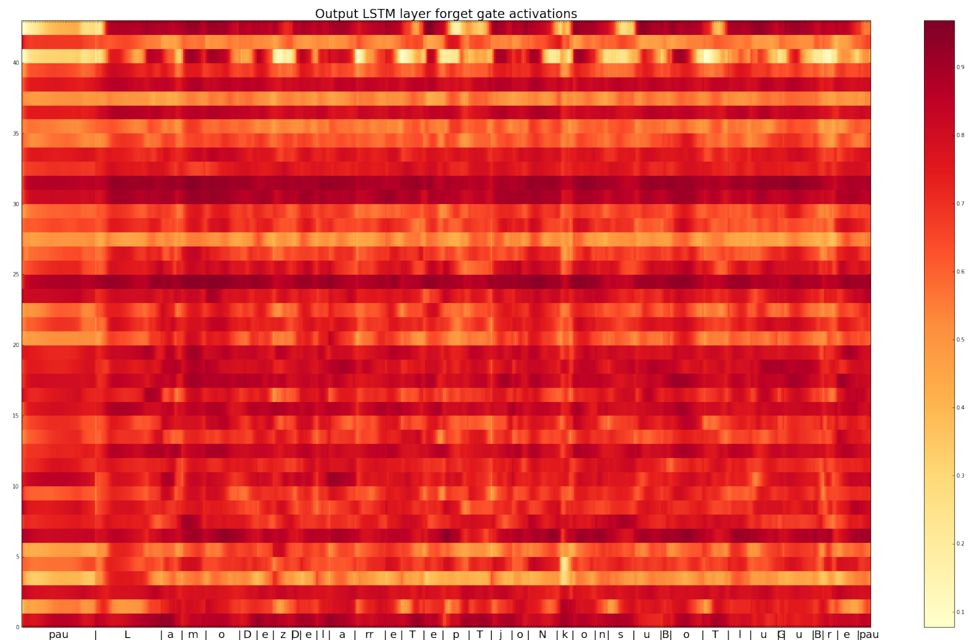

FIGURE 4.27: Activations map for the forget gates of the 43 output LSTM cells. Red regions mean letting the cell state flow out of the cell.

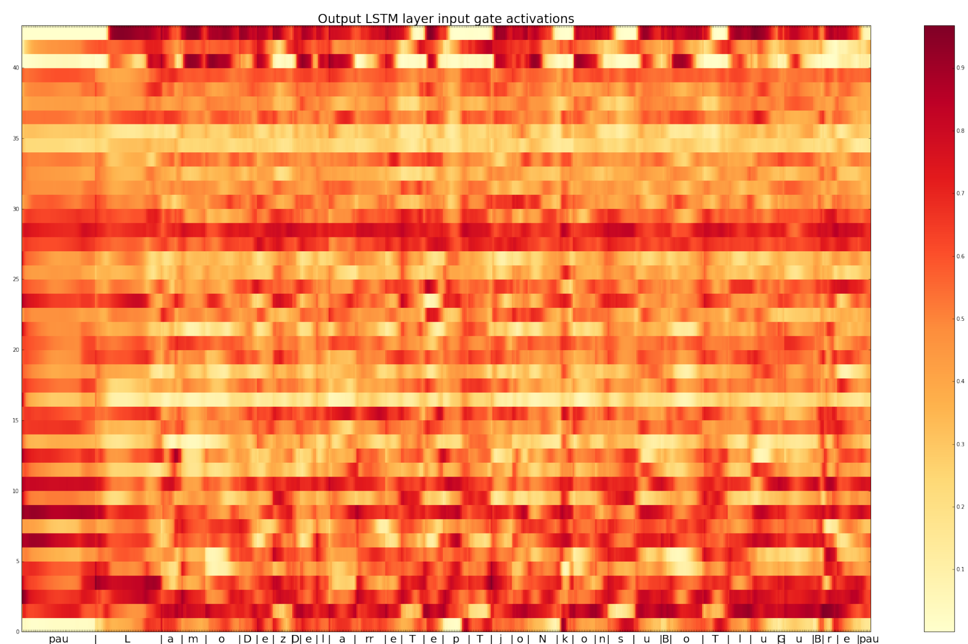

FIGURE 4.28: Activations map for the input gates of the 43 output LSTM cells. Red regions mean letting the cell state flow out of the cell.

Figure 4.30 depicts the averaged gate activations in a similar analysis to that of Figure 4.25 for the hidden layer. Here it is appreciated how the output activation is usually high due to the functionality that this layer is performing, but there appear some valleys in the unvoiced regions which

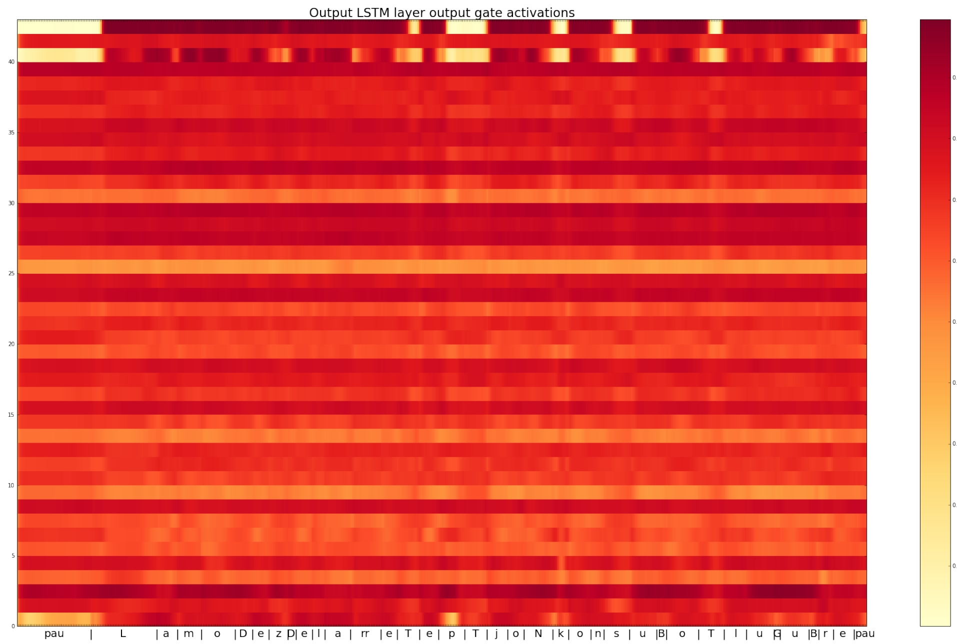

FIGURE 4.29: Activations map for the output gates of the 43 output LSTM cells. Red regions mean letting the cell state flow out of the cell.

are probably due to the loss of energy, so some output has to be blocked, as well as the U/V and FV values commented previously. The behavior of the input and forget gates seem to be more erratic than the linear decaying found in the hidden layer, but both gates seem to be again closely related as previously seen in Figure 4.26.

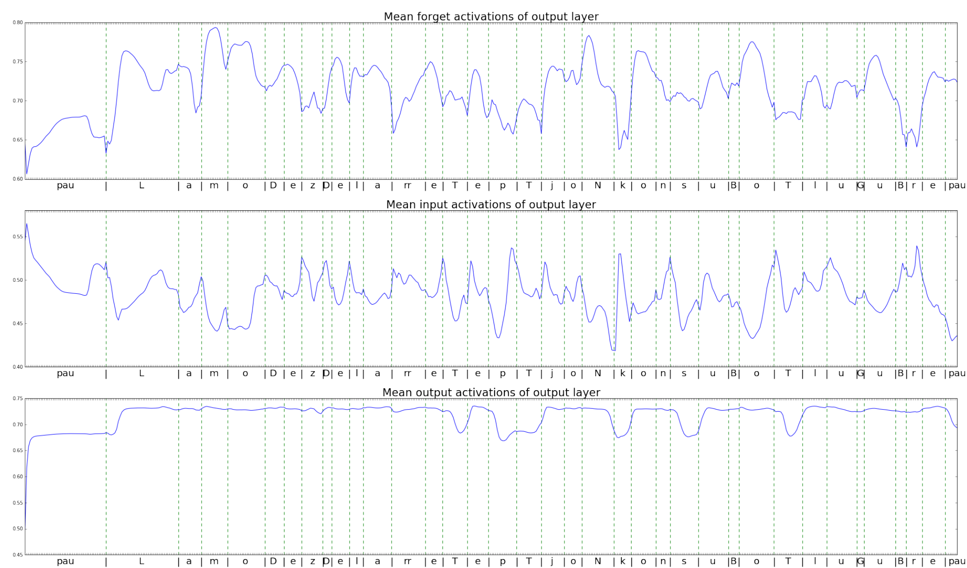

FIGURE 4.30: Average activations for the output LSTM layer with 43 cells. Input gate, forget gate and output gate are shown. Green dashed lines are the phoneme boundaries.

4.6 Discussion

A TTS made from scratch with RNN-LSTM architectures has been shown in this chapter. First, the types of data to be used as predictors and predictions have been introduced in section 4.2, where a parallelization pipeline developed for the project has also been explained. The way in which those features are encoded and normalized for the sake of a good learning process during the network training has been explained in detail.

Then in section 4.3 the TTS system is shown, which is composed of two models forming a twofold stage methodology, which are the duration model and the acoustic model. The first one predicts the amount of acoustic frames to be generated by the second, thus having an interconnection between the two stages to generate the acoustic stream of features. We have seen how these predicted features have to be decoded by a Vocoder, as introduced. In section 4.4 we introduce a post-filtering mechanism that is applied to the acoustic prediction in order to enhance the naturalness of the generated speech. The origin of the issue degrading naturalness is shown together with the correction method.

This chapter concludes with the results shown in section 4.5, where we have seen some brief architecture search to tune the different parameters of the network in an objectively evaluated search. The experiments validate the architecture we use for final evaluation and synthesis, although in the case of the acoustic model the tested topologies are not suffering big changes in the objective results. In addition, there is a subjective test performed to evaluate the naturalness achieved by the developed TTS compared to natural speech and other existing TTS systems. The results suggest two main things: first, the TTS developed in this work is the one with the best rated naturalness in comparison to the other existing technologies available: Unit Selection and Statistical Parametric Speech Synthesis.It should be noted that in international evaluations (King, 2014) Unit Selection or hybrid systems US-SPSS are the most successful approaches raising the best results. Therefore, the comparison performed is a first reference but has to be taken carefully because the evaluations were not made with the best US nor SPSS models. Secondly, the post-filtering technique is effective in terms of human perception by a large margin compared with the system without applying it. In the end of the results section there is an analysis of the insights of the LSTM layers, where the activations of each gate (input, forget and output) are pictured to get some clues about the correlation they have with the inputs and predictions in the acoustic model. We have seen the clear relationship between the input and forget gates for the acoustic parameters generation task, as stated in the work by Wu and King (2016), and also the behavior of the output layer that always tries to let predictions get out, though it produces a slight closing effect on unvoiced frames.

Chapter 5

Multiple Output Acoustic Mapping

5.1 Introduction

In this chapter it is introduced a new model derived from the previous one, called the Multi-Output (MO). Here, the previous Text To Speech (TTS) acoustic mapping is extended so that one model reproduces several speakers' speech. With this approach we can tackle three problems with a single deep RNN-LSTM model:

- Making multiple speaker models out of the same structure.
- Make speaker adaptation with new data on top of this model.
- Make speaker interpolation out of the learned representations.

The system is then capable of holding many speaker models inside the same shared structure, so that every user shares its characteristics with the others, thus reducing the required number of parameters per speaker and letting them interact in the lower layers, where the linguistic information processing is performed, as seen in Chapter 4. With this aim in mind, we wanted to figure out whether or not it was helpful for the different speakers to see others' data, although every speaker output is independent from each other and they are only tied to the same inputs. There was a proposal of a similar approach by Fan et al., 2015 with DNNs performing multi-task learning, but in our case we work with RNN architectures, with a different training procedure and also with a different speaker adaptation architecture.

5.2 Data Preparation

The data generation method presented in Chapter 4 is really helpful here, because we need to generate the data for the different speakers in an agile way. We worked with six voices from the TCSTAR (Bonafonte et al., 2006b) project, where four of them contain expressive speech, and two neutral voices from interface (Hozjan et al., 2002). We balanced the data per user, such that all of them have approximately the same amount of samples to train, i.e. 20 minutes of speech per speaker. There are four male voices (M1, M2, M3, M4) and four female voices (F1, F2, F3, F4). The F3 data is

separated from the other ones because it is used for the speaker adaptation experiment. We will focus the results on the M1, F1 and F3 speakers, all of them having 4 minutes of samples for testing and 4 minutes for validating.

5.3 Multi-Output architecture for acoustic mapping

The proposed architecture is depicted in Figure 5.1, and this is the model we proposed in our work (Pascual and Bonafonte, 2016a). There are two first Fully Connected layers serving as a bottleneck for the sparse inputs. These intend to get a deeper knowledge about the input data, which is formed by the mixed set of multiple types of features shown in Chapter 4. There is a first stage of LSTM hidden layers, processing every transformed input set of features at each time step, and deriving the results to the output branches. Dropout is performed between the hidden recurrent layer and the output layers to mitigate any over-fitting. Each output branch belongs to a different speaker, so at prediction time we inject the linguistic parameters to the model to obtain every speaker's speech parameters at the output. It is clear that this model is an extension of the one shown in Chapter 4, putting $(N-1)$ additional output layers.

Every output branch is independent of each other, so it propagates its own error through the whole shared structure at training time without the need of padding any data for the other outputs. The intuition behind this architecture is that, whilst every output branch is trained, it shares the first linguistic mappings with other branches. This might lead to an improvement in the final acoustic mapping of every speaker in comparison to the speaker model trained in an isolated manner, because we add more information during training time to get to correlated predictions at the different outputs. For the experiments of this model, the first two Fully Connected layers had 128 hidden units each one with $\tanh$ activation functions. Moreover, there is one shared hidden LSTM layer containing 256 memory cells, also with $\tanh$ activation functions, and the dropout applied is 0.5. Finally, the output layers were like the ones for the single speaker models, 43 units to produce the acoustic predictions in a regression fashion after a sigmoid activation function (see Chapter 4 for more information about the acoustic features predicted). The LSTM units had the forget gate bias initialized to one for better performance, as specified in Jozefowicz, Zaremba, and Sutskever (2015).

The training procedure was a challenge for this model, owing to the fact that we had two options:

- Have the same data aligned for all speakers, which is very hard to achieve because of the fact that the speakers have different prosodic pauses, durations, etc.. in addition to silences, which have varied lengths as well. This made the simultaneous back-propagation of all branches gradients very hard to achieve.
- Back-propagate each speaker individually and sequentially, dividing the training process into rounds. This solution doesn't require a direct mapping of a linguistic input to all the output speakers, and thus the

data is not required to be aligned for the N outputs. This is the way in which the training has been implemented in this work.

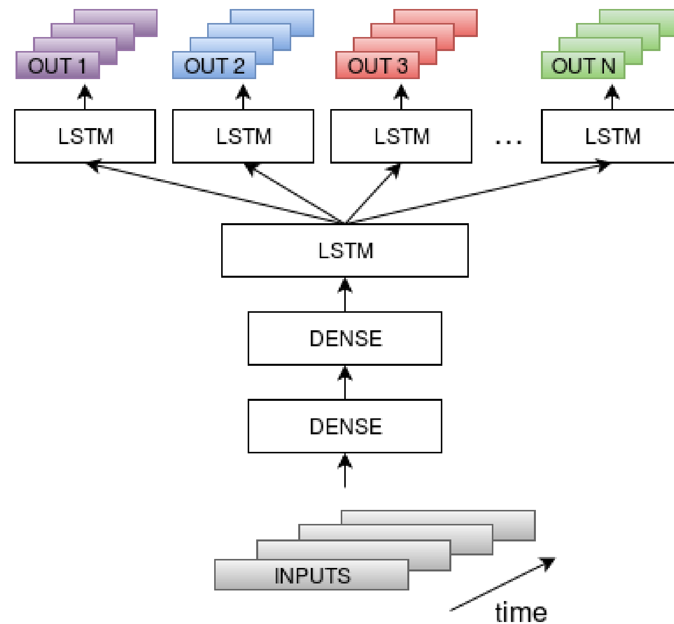

FIGURE 5.1: Proposed architecture using regular feed forward (dense) layers and recurrent LSTM layers. There are N outputs belonging to N different speakers.

In the training stage, every speaker error is back-propagated independently and sequentially, such that a speaker ID is randomly selected in every round (assigning a turn to the speaker), completing an epoch when all rounds of speakers have been seen. A round is a sequence of back-propagated mini-batches, having N mini-batches per round as we have N speakers, each speaker having its turn inside the round, and when a speaker mini-batch is back-propagated inside the round we move on to the next randomly picked one until all speakers are processed and we can shuffle the IDs again. Figure 5.2 depicts the training procedure for a round in the epoch, where the turn of every mini-batch is in brackets. Note that with this method we do not require to have the same transcriptions per speaker, as mentioned earlier, and also every mini-batch is related to only one speaker.

As a comparison with the previous proposed single speaker model, in the case that we had $N = 6$ speakers, we know that a single acoustic model required 3.9M parameters, so a straightforward calculation can be done to know the amount of parameters required for representing 6 speakers with single models: $6 \cdot 3.9$M $= 23.6$M. On the other hand and thanks to the shared structure, the multi-output model only requires 4.4M parameters, which is the 18.69% of what the individual models needed.

5.4 Multi-Output architecture for speaker adaptation

In order to do speaker adaptation we insert a new output branch for speaker $N+1$, and we try two training mechanisms for adapting our multi-output

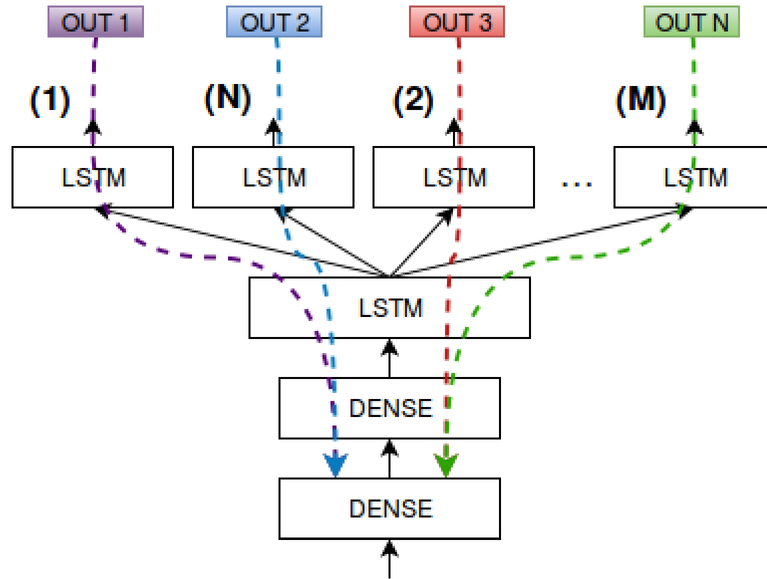

FIGURE 5.2: Exemplified training round for the N mini-batches. Dashed lines represent the corresponding output error back-propagation. The numbers in brackets express the order of that mini-batch inside the round.

model to the new speaker. For the purpose of checking how we can insert a new speaker into the architecture with minimal changes and best results, we have the speaker F3 separated. When the multiple output model is trained, we get what we call the pre-trained multiple speaker weights (the ones in all layers just before the output LSTMs) and attach the new output branch to the model, back-propagating only the new speaker. We make the adaptation experiment varying the amount of data available for the new speaker in batches of 25%, 50% and 100% of the total available data, this way we check the possible change in the quality of the new voice depending on the amount of data available. Moreover, we mentioned the requirement of a simple adaptation model, where we change the least things possible, so we try freezing the shared layers to not update them during the back-propagation of the new speaker, training only the new output branch as a possible approach. The scheme for this model is shown in Figure 5.3.

5.5 Multi-Output architecture for speaker interpolation: α-model

This task is about creating a new speaker out of the mixture of an M-sized subset of the N pre-trained speakers. This means that, once the multi-output model is available with the pre-trained N output branches, we find a way to mix the different speaker representation in a complete synthetic way, without any data of a new speaker. The way in which this was done is by means of another layer, which we called the α-interpolation layer. The architecture proposed is the one in Figure 5.4 (Pascual and Bonafonte, 2016b). There the α-interpolation layer is attached to M branches once the multi-output is trained, where: $M \leq N$.

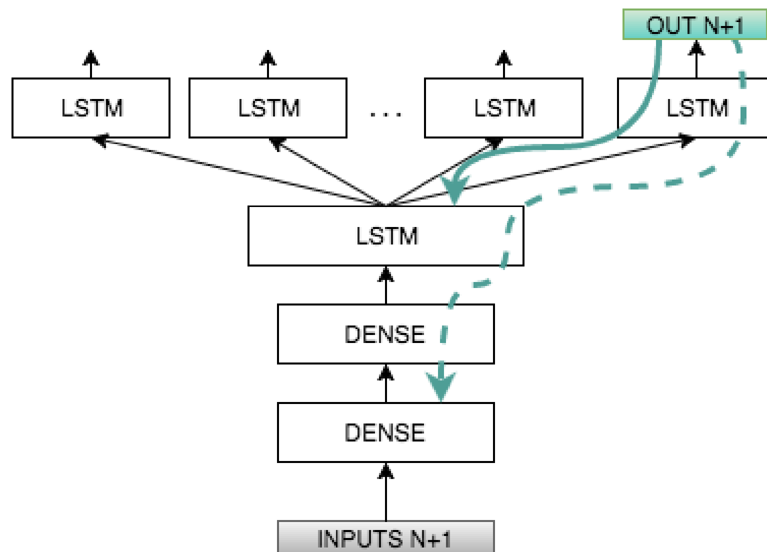

FIGURE 5.3: Speaker adaptation system by fine-tuning a pre-trained Multi-Output model. The new layer can be trained in two ways: solid-line: 1) fine-tune only new branch with frozen model in the lower layers. 2) Fine-tune the whole model, thus propagating the error until the first hidden layer.

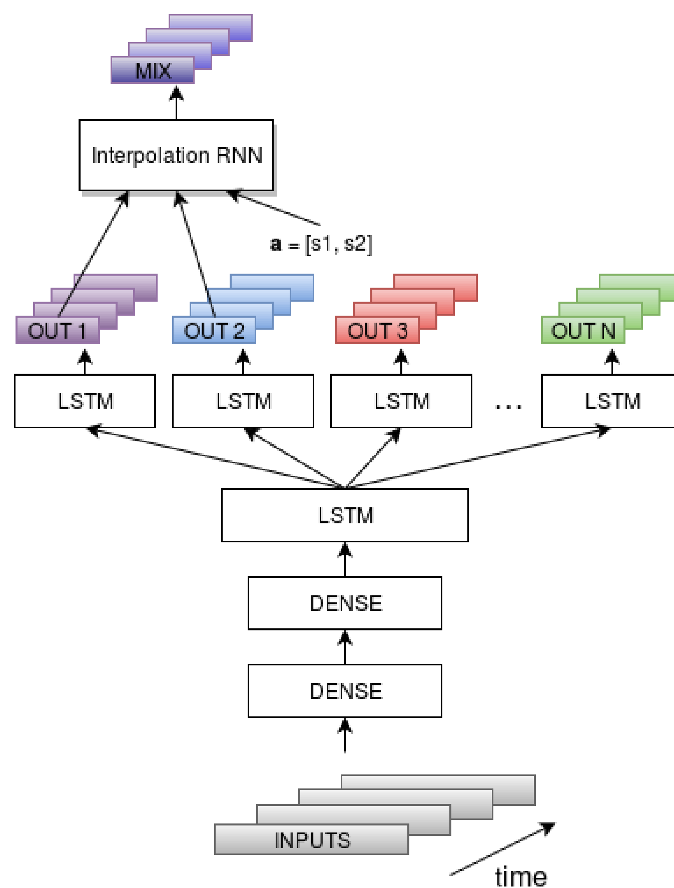

FIGURE 5.4: Speaker interpolation system by training a new mixing layer, the α-layer. The new layer uses the input α codes to learn to interpolate between the extreme examples given.

The α-layer is trained by freezing the multi-output model weights. As mentioned previously, the layer has M speaker branches as input, thus raising $M \times O + M$ input units, where O is the acoustic vector dimension. The M added inputs come from another input vector which is inserted to control the weight that every speaker has in the interpolation, called the α vector (which gives its name to the layer), which is a one-hot code of dimension M. During training, linguistic inputs are injected into the multi-output model and then the inference is made, which turns to be the inputs for the interpolation layer, which also get the one-hot α concatenated, expressing the identity of the current shown speaker at the interpolation layer output as mentioned before. The same training data used to train the multi-output model is used for the M speakers. Figure 5.5 depicts the training procedure for an example with 3 speakers and for 3 batches.

This methodology then expects the layer to learn not only each extreme case (i.e. each one-hot case shown during training), but it is also expected to infer intermediate values for the acoustic outputs, and as it is seen in section 5.6.3, it actually learns to interpolate the features.

During synthesis, the α code is replaced by a probability distribution among the speakers, thus expressing the percentage to synthesize for every speaker, so for instance we could have the following α-vector in synthesis:

$$\alpha = (0.5, 0.5, 0, 0)$$

This is a code for an interpolation layer made out of $M = 4$ speakers, and it means that the mixture is made with 50% of speaker 1 and 50% of speaker 2, while the other two remain 0, so no information from them is required to generate the acoustic predictions.

This type of code is the first approach for making an interpolation architecture that will be improved in a future line of research with identity related to acoustic characteristics of speakers, such as the *i-vectors*.

5.6 Results

The results for the different architectures proposed are explained here, so that we see the effect of:

- How is the overall quality of the multi-output model compared to the individual models? Are the speakers distorting each other or are they helping each other?
- How does the speaker adaptation perform when we back-propagate different amounts of the available data? How does affect the back-propagation through the whole model in comparison to freezing the base model and updating only the new output? How does it do comparing to the single speaker model of the new speaker?
- Regarding the interpolation, is the α-layer learning a suitable and meaningful intermediate range of values for the different speakers? How is the $M \geq \mu$ affecting the resulting prediction of μ speakers?

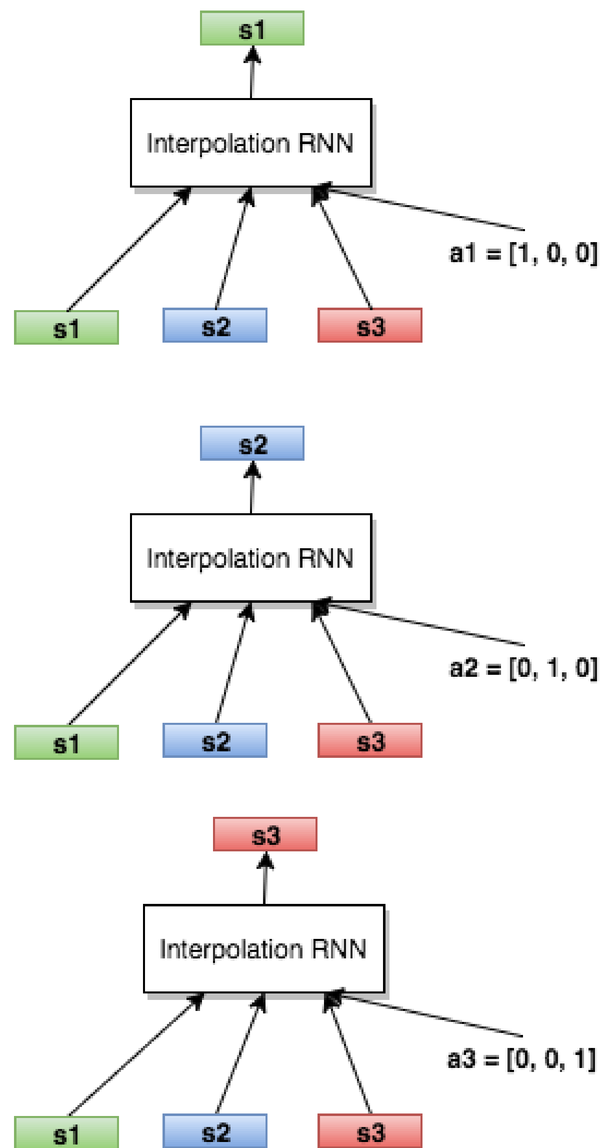

FIGURE 5.5: α-interpolation training method for an example with $M = 3$ for 3 batches of examples. The one-hot code expresses the identity of the currently shown speaker. Each s_m is the output prediction of the corresponding multi-output branch for the m-th speaker.

The following sections analyze these questions given the objective and subjective results obtained during the course of this work.

5.6.1 Results: Multi-output

We make a first analysis by looking at the training loss evolution of the different speaker outputs, and concretely focusing on two speakers: M1 and F1. To establish a reference, we trained M1 and F1 with a single output architecture and multiple output one. The results can be seen in Figure 5.6, where the 7 speaker learning curves are shown, depicting that all output converge with a noisy behavior, given by the training methodology where every speaker distorts each other's learning process for mini-batches of data.

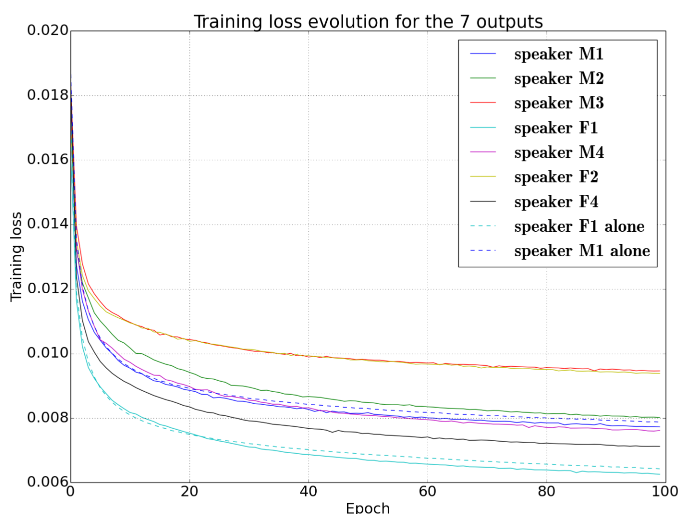

FIGURE 5.6: Training loss evolution comparison. Speakers F1 and M1 decrease the learning cost when trained with other speakers altogether.

There we can see how speakers F1 and M1 get to a lower training loss when they are trained with the multiple output mechanism. This is normally related to a better training procedure where they reach a better point in the optimization.

To really see this effect, we first make an objective evaluation by means of specific metrics for each kind of predicted feature, as it was previously done in Chapter 4. Concretely we used the Mel Cepstral Distortion (Mashimo et al., 2001) (MCD), the RMSE of the predicted F0 (Hertz scale) and the Accuracy in UV flag prediction. Table 5.1 shows the results for F1 and M2, which show the alone models (i.e. the speaker trained with its own acoustic model isolated from others) and the mixed models (i.e. the speaker trained in a multi-output fashion with the other 6 speakers). The *alone* results for M1 differ a little bit from the ones obtained in Chapter 4 because we have different data splits, which at same time come from a much less amount of samples.

The objective results suggest the improvement in the features estimation when the speaker models are trained in the multi-output fashion, as they

share knowledge in the lowest layers of the network to transfer learning about the final mappings.

Model	MCD[dB]	F0[Hz]	UV[%]
M1 alone	7.6	14.4	92.3
M1 mixed	7.2	13.8	94.2
F1 alone	7.0	17.3	95.2
F1 mixed	6.5	17.3	96.2

TABLE 5.1: Objective evaluation for M1 and F1 trained alone with a single output model and together with other speakers (mixed) in the multiple output architecture.

A subjective evaluation has been carried out as well with a preference test made by 16 subjects. For both F1 and M1 speakers, 5 sentences are selected and evaluated. The listeners can choose a declining score between two synthesized utterances; one generated by the single output model and another one by the multiple output one. Listeners then find five options available from -2 (multiple output is much preferred) to 2 (single output is much preferred). The type of system per utterance is hidden for the listeners and randomly ordered.

The results are depicted in Figure 5.7. It can be seen that the testing subjects have all rather preferred the multiple output model in most of the cases. We also made a Wilcoxon test for the subjective evaluation to find out how statistically meaningful are these results, obtaining the following p-values: $p_{F1} = 5.3 \cdot 10^{-7}$ and $p_{M1} = 2.2 \cdot 10^{-5}$.

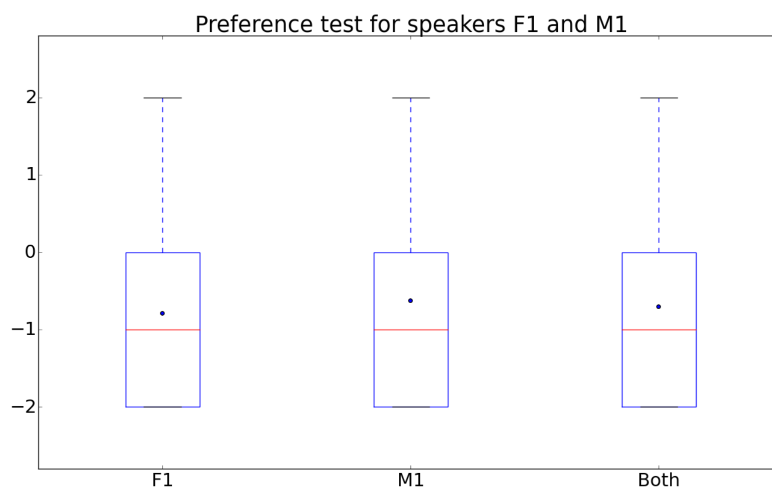

FIGURE 5.7: Box plot of preference test scores. Scores range from -2 (multiple output model is preferred) to 2 (single output trained model is preferred). Both is the summary of all the answers, joining both speaker results. Red lines: medians. Blue dots: means.

5.6.2 Results: Adaptation

In this section we analyze the effect of an adaptation layer constructed on top of the MO architecture. In Figure 5.8 we can see how the validation cost for the F3 speaker improves when we add more data, something we could expect, as it learns better with the more data it gets to fine-tune the new output branch. It is interesting to see how freezing the shared layers and training only the new output branch we get to a very similar result, and more smoothly.

Table 5.2 summarizes the objective evaluation for this fine-tuning, getting a good result with respect to the single output model of F3 when only the last layer is trained on top of the shared parts of the model. The error values are quite high in comparison with the previous ones (F1, M1), because this speaker was taken from an expressive subset of data, being it quite different from that of F1 and M1.

An informal listening test suggested that the adaptation sounded close to the original speaker, thus validating this approach.

Model	MCD[dB]	RMSE F0[Hz]	UV[%]
F3 alone	8.11	28.07	9.00
F3 fine-tuned full 100% data	7.96	26.96	7.74
F3 fine-tuned frozen 100% data	7.90	26.44	6.73

TABLE 5.2: Objective evaluation for F3 as an adaptation subject. Full: all layers are fine-tuned. Frozen: only new output branch is fine-tuned.

5.6.3 Results: α-interpolation

Finally, we designed the experiments to evaluate the α-interpolation proposal with 2 speakers out of the 6 mentioned previously from the TCSTAR database. For the interpolation, two configurations were trained as will be seen, $M = (2, 6)$. This means that the interpolation was carried out between speakers F1 and M1, but there is also a configuration where the α-layer is trained will all the MO speakers, thus $M = 6$.

An informal subjective test clearly showed that increasing M improved the naturalness of the output speech although only 2 speakers are interpolated in the evaluation.

Objective tests have been performed to evaluate the performance of the α interpolation. These consist in analyzing the evolution of the MCD between the interpolation output and each of the M branches, and also the evolution of the F0 RMSE. Figures 5.9 and 5.10 show these results, where the α variation is made for speaker F1, so it is α_{F1} , speaker M1 has $\alpha_{M1} = (1 - \alpha_{F1})$ and all others are $\alpha_m = 0$. We may refer to $\alpha = 0.5$ for the point at $\alpha_{F1} = \alpha_{M1} = 0.5$.

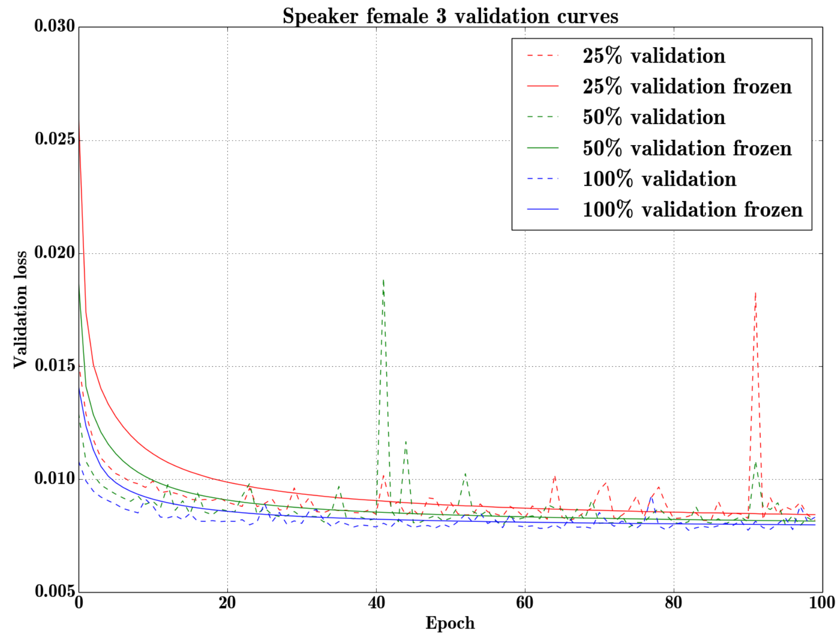

FIGURE 5.8: Validation loss evolution comparison of different batch sizes, with frozen shared layers and fine-tuned shared layers.

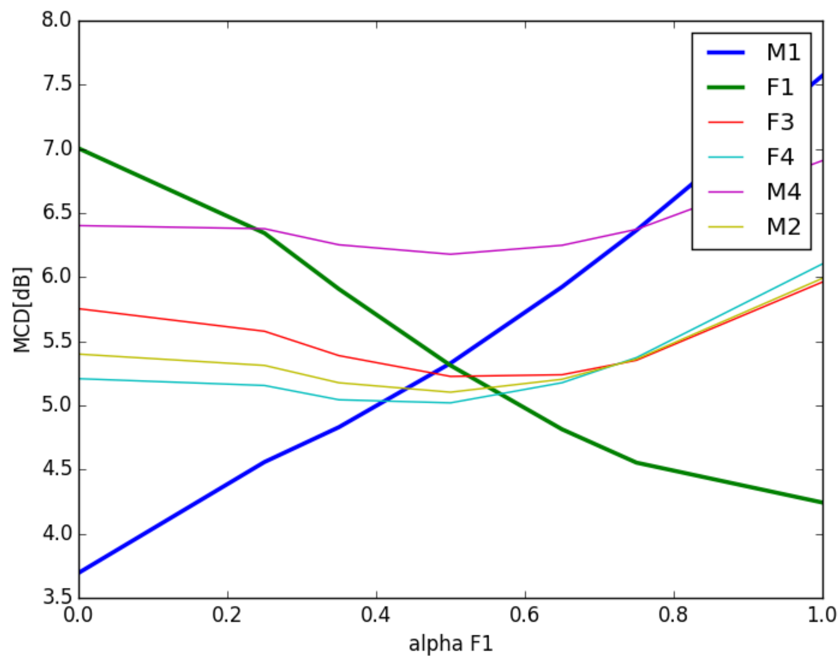

FIGURE 5.9: MCD when varying α values. The variation is made for speaker F1 and it is $(1-\alpha)$ for M1. All others speakers remain 0. $M = 6$.

From the curves we see how, although we only show to the network the extreme values with an orthogonal code, it learns the intermediate representations effectively. The MCD values vary smoothly between the interpolated speakers F1 and M1, whilst other speakers' MCD remain with a short variation. It is interesting the fact that the crossing point is very close to $\alpha = 0.5$.

Note that the values may differ from those in Table 5.1 because the distances are not computed to natural speech but to the multi-output predictions in this case.

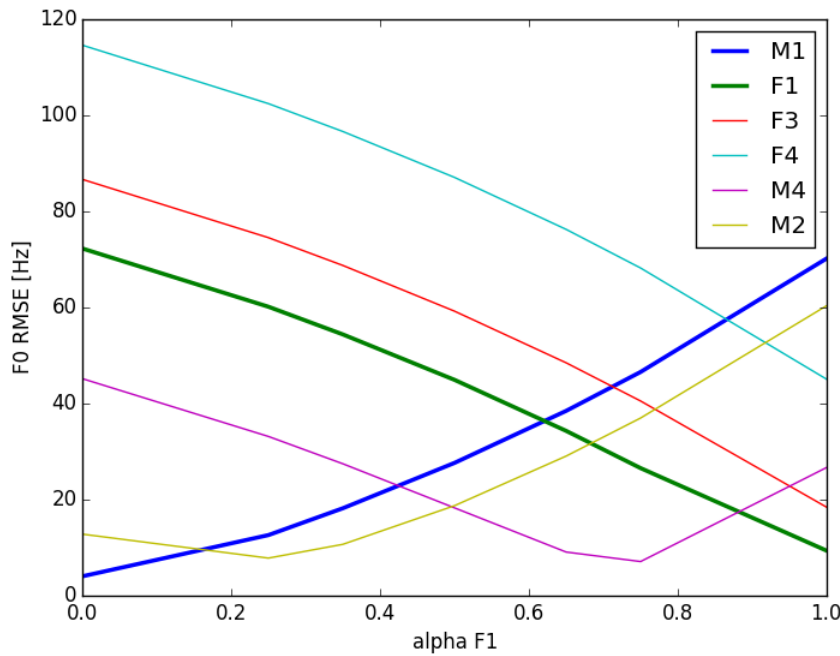

FIGURE 5.10: F0 RMSE when varying α values. The variation is made for speaker F1 and it is $(1 - \alpha)$ for M1. All others speakers remain 0. $M = 6$.

Regarding the F0 RMSE evolution, there is a biasing of the crossing point, which shows us how the F0 prediction is biased towards the male speaker, as it is the one getting less error for $\alpha = 0.5$. These interpolation results are coherent with perceptual impression. As previously mentioned, increasing M helped in the naturalness of the 2-speaker interpolation, so an analysis of the F0 distributions is also made for the cases $M = 2$ and $M = 6$. These analysis are shown in Figures 5.11 and 5.12 respectively.

First, we can confirm the biasing towards the male speaker when $\alpha = 0.5$ (M1 50% F1 50%) in both cases. Nevertheless an important difference is the fact that training the layer with a higher M increases the distributions variance, which turns out to be a less monotonous sound at the output, and thus more natural.

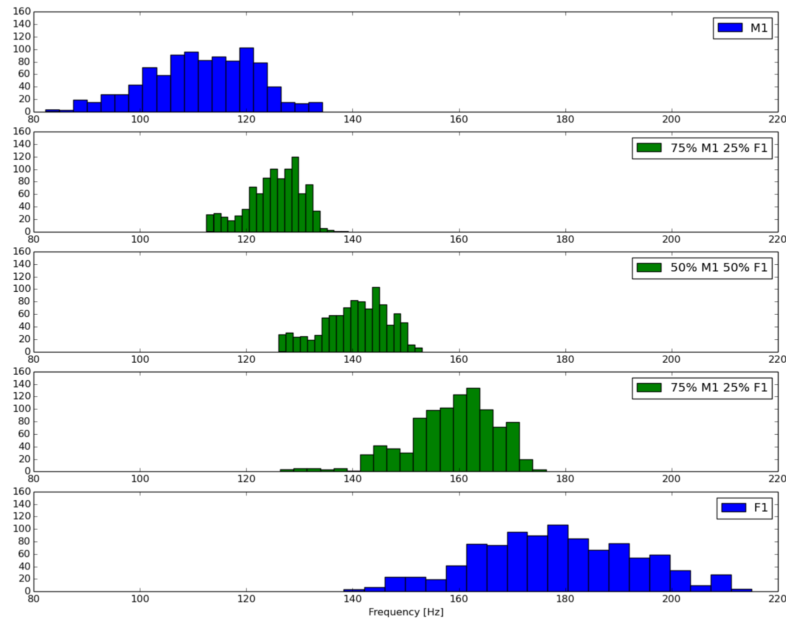

FIGURE 5.11: F0 Histograms: original M1 and F1 speakers in blue. $\alpha_{F1} = (0.25, 0.5, 0.75)$ and $\alpha_{M1} = (0.75, 0.5, 0.25)$ interpolations in green. $M = 2$.

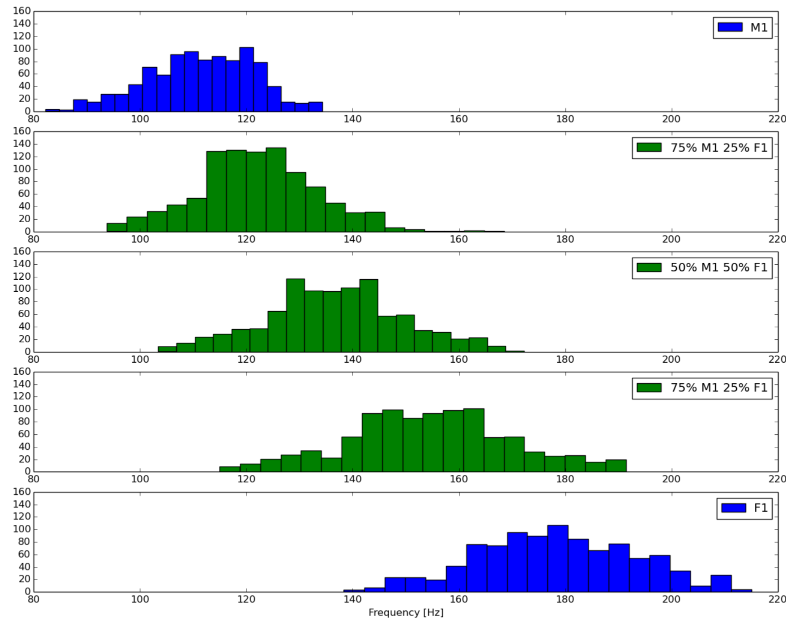

FIGURE 5.12: F0 Histograms: original M1 and F1 speakers in blue. $\alpha_{F1} = (0.25, 0.5, 0.75)$ and $\alpha_{M1} = (0.75, 0.5, 0.25)$ interpolations in green. $M = 6$.

5.7 Discussion

A novel architecture for modeling the acoustic mapping for many speakers at once has been proposed and studied in section 5.3. The model lets us represent N speakers with a very reduced set of parameters in comparison to making N separated models, once per speaker. Moreover, we have seen how the different speakers help each other in the learning process by transferring their knowledge in the linguistic mappings done in the earlier layers. Objective and subjective evaluations confirm the learning advantage that the different speakers obtain by being trained jointly with the multi-output architecture.

On top of this, a speaker adaptation technique is presented in section 5.4, where a new output branch is attached to the multi-output architecture and two back-propagataion approaches have been studied. The objective evaluation showed the improvement of this technique over training the speaker isolated as well.

Finally, an interpolation model has been built also taking advantage of this novel architecture by means of the so called α-layer, which is trained with an orthogonal code expressing speaker identities during training and speaker portions during synthesis. The results suggest that the layer can effectively learn intermediate ranges of speaker representations by only being trained with extreme cases (each speaker's voice examples). Furthermore, inserting as many speakers as possible to the α-layer increases the variability of the predictions in the output, which turns out to be a more natural result.

Chapter 6

Conclusions

This chapter is devoted to make a review of this work, discussing the implemented architectures and achieved results. Lines of future work are also explained such that the reader can get to see the possibilities opened by this contribution. In the end of the chapter the research contribution of the work is shown.

6.1 Thesis Review

A speech synthesis system made from scratch with RNN-LSTM architectures has been built in this work. The system is based on a two stage architecture, where first the duration of the phonemes to generate are predicted, and then the acoustic parameters to be passed to a Vocoder are generated frame by frame up to the corresponding duration. The types of features with which the network makes the predictions are explained in detail: the context labels that contain phonetic and prosodic information about the input text and the acoustic features predicted. To generate the data used by the neural networks a parallelized framework is made, such that the Vocoding and text-to-label processes are sped up to be able to deal with many speakers' data quickly and with large amounts of data. It has also been shown how the acoustic model suffers an over-smoothing effect of the generated parameters, because it tends to predict the means and not the variances for the inherent behavior of the regression training function, the MSE. A post-filtering mechanism is then applied in the generation of acoustic parameters to overcome this issue and enhance the naturalness of the synthesis.

After presenting all the methodologies behind the Text-To-Speech (TTS) system there are different types of evaluations made for the duration and acoustic models: some brief architecture search to tune the different parameters of the network in an objective way is performed, with which we obtained a deeper architecture in the acoustic model than in the duration one. The duration model architecture seeking process shows that it suffers from variability in the results when we vary the network depth (amount of hidden layers) and width (amount of hidden units/cells), so the variation of topology in this case is meaningful to achieve better results. On the other hand the acoustic model is not perturbed very much and the results are just slightly different when tuning the model for the chosen parameters. Nonetheless. the best performing model in objective terms was picked as the representative one in both cases.

The TTS is also evaluated subjectively to see how good it performs in comparison to natural speech, and at the same time the model is compared to different current TTS systems. The results suggest two main things: first, the TTS developed in this work is the one with the best rated naturalness in comparison to the others we had. Secondly, the post-filtering technique is effective in terms of human perception by a large margin compared with the system without applying it.

After the TTS results are shown there is also an analysis of the LSTM gate activations in the acoustic model for two layers: the first hidden layer and the output layer. The three gate activations (input, forget and output) are visualized for an arbitrary test sentence. From this analysis we get some clues about the correlation of the gates with the inputs and predictions for the speech synthesis task, where we see that the network reacts sharply in the hidden layer whenever a new input phoneme is given to it. We have also found a clear relationship between the input and forget gates for the acoustic parameters generation task, as stated in the work by Wu and King (2016). Another interesting fact derived from this analysis is that the output gate of the output layer seems to block part of the outputs when unvoiced phonemes are going to be predicted, something we could expect because the unvoiced phonemes do not have F0, and besides the voiced frequency parameter and the voiced/unvoiced flag are set to the lowest value in these cases. Given these facts, we corroborate how the network accepts less input values in those regions and also tries to block any output information flowing from it.

The last part of the work has been the proposition of a novel architecture that can hold many speaker acoustic models inside the same shared structure, thus sharing knowledge among speaker linguistic representations in the lowest layers. The architecture is called Multi-Output acoustic model (Pascual and Bonafonte, 2016a), and it is based on having a set of shared hidden layers that perform the first linguistic mappings and then they diverge into many output branches, where each of these pertain to a speaker acoustic parameterization. A proper training algorithm for this system is developed such that there is no requirement to have the same training input/output pairs for all the speakers in the database, which give us way more freedom to use the data of completely different recordings. In the objective and subjective results it is proved that this methodology helps the individual speaker models to learn better representations that lead to better generated speech.

We also worked in a first speaker adaptation approach on top of the Multi-Output acoustic model, where we just need to insert another output branch on top of the pre-trained system and fine-tune it without modifying the whole structure, thus preserving the multiple output base model and lowering the footprint in memory to get a new speaker model. This approach worked better than training the new speaker in an isolated way.

Finally, we developed a speaker interpolation model as well, where we need to insert another output layer (the α-layer) on top of M pre-trained speaker branches from the Multi-Output, and fine-tune it without modifying the whole structure. We have seen how, when we show the network

extreme cases representing the different speakers by means of an orthogonal identity code, it is able to infer intermediate values. Furthermore, training the α-layer with all the available speakers helped the interpolation to increase the variance, thus producing more natural speech.

6.2 Future work

The future lines of work envisaged from the research and development performed here cover mainly three topics in the case of the TTS:

- Alternative representations of input linguistic features.
- Supression of the Vocoder stage to generate waveforms directly.
- End-to-end learning from text-to-speech to avoid the two-stage paradigm.

First, getting alternative representations of the input linguistic features can lead to representations that fit better to problems like expressive speech generation, or even multiple language speech synthesis. Therefore, a future line of work will be looking for alternative linguistic representations to those of the pre-defined labels that were used in the HMM-based SPSS. These features derived from previous technologies are quite naive (phonetic context, stress and some features about sentence modality and length), hence it will be interesting to explore if there is room for improvement making the network able to extract semantic information out of the text. This could produce more expressive speech, continuing with the line of research proposed in Jauk, Bonafonte, and Pascual (2016).

Regarding the acoustic parameter generation, working with hybrid systems has given good results (King, 2014), thus exploring models that mix the Unit Selection technology, which produces high quality speech, with the generalization mechanisms of the neural networks will be another interesting line of research.

So far it has been discussed a type of architecture that uses a first stage to predict the duration of the phonemes and another one to predict the acoustic parameterization. This is done because we go from a phonetic representation of the text to the acoustic representation of the phonemes. Another possible approach that will be explored will be an encoder-decoder model similar to that of Cho et al. (2014a) used in Neural Machine Translation (NMT), with which the raw input text will be analyzed and compacted into an intermediate representation called context vector in the encoder stage. Once the context vector is captured it is expanded into the destination sequence of predictions in the decoder stage, where this sequence is composed of the acoustic parameters. Furthermore, the use of an attention mechanism (Bahdanau, Cho, and Bengio, 2014) for this type of architecture will be used to impose a soft-alignment between the source sentence of words and the destination acoustic parameters. This would be an end-to-end TTS system, where the input examples formed by text and output examples formed by speech recordings do not need any previous alignment. It is the network then that learns to map the right slices of the input data to the right destination parameters automatically.

Regarding the interpolation technique, a future research that can be conducted is substituting the orthogonal codes for a representation closer to the acoustic feature space, such as the *i-vector*.

6.3 Research Contribution

A full software framework has been built in this project to make the different TTS architectures aforementioned. Also, the following publications were made out of techniques developed in this thesis:

- Multi-Output RNN-LSTM for multiple speaker speech synthesis and adaptation. Santiago Pascual and Antonio Bonafonte. EUSIPCO 2016.
- Acoustic Feature Prediction from Semantic Features for Expressive Speech using Deep Neural Networks. Igor Jauk, Antonio Bonafonte and Santiago Pascual. EUSIPCO 2016.
- Multi-output RNN-LSTM for multiple speaker speech synthesis with α-interpolation model. Santiago Pascual and Antonio Bonafonte. Submitted to ISCA SSW9.

Bibliography

Achanta, Sivanand, Tejas Godambe, and Suryakanth V Gangashetty (2015). "An investigation of recurrent neural network architectures for statistical parametric speech synthesis". In: *Proc. Interspeech*.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2014). "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473*.

Bengio, Yoshua (2012). "Practical recommendations for gradient-based training of deep architectures". In: *Neural Networks: Tricks of the Trade*. Springer, pp. 437–478.

Bengio, Yoshua, Nicolas Boulanger-Lewandowski, and Razvan Pascanu (2013). "Advances in optimizing recurrent networks". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, pp. 8624–8628.

Bishop, Christopher M (1995). *Neural networks for pattern recognition*. Oxford university press.

Bonafonte, Antonio et al. (2006a). "Ogmios: The UPC text-to-speech synthesis system for spoken translation". In: *TC-STAR Workshop on Speech-to-Speech Translation*, pp. 199–204.

Bonafonte, Antonio et al. (2006b). "TC-STAR: Specifications of language resources and evaluation for speech synthesis". In: *Proc. of LREC Conf*, pp. 311–314.

Chen, Sin-Horng, Shaw-Hwa Hwang, and Yih-Ru Wang (1998). "An RNN-based prosodic information synthesizer for Mandarin text-to-speech". In: *IEEE transactions on speech and audio processing* 6.3, pp. 226–239.

Cho, Kyunghyun et al. (2014a). "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078*.

Cho, Kyunghyun et al. (2014b). "On the properties of neural machine translation: Encoder-decoder approaches". In: *arXiv preprint arXiv:1409.1259*.

Chollet, François (2015). *Keras*. `https://github.com/fchollet/keras`.

Chung, Junyoung et al. (2014). "Empirical evaluation of gated recurrent neural networks on sequence modeling". In: *arXiv preprint arXiv:1412.3555*.

Coto-Jiménez, Marvin and John Goddard-Close (2016). "LSTM Deep Neural Networks Postfiltering for Improving the Quality of Synthetic Voices". In: *arXiv preprint arXiv:1602.02656*.

Deng, Li and Dong Yu (2014). "Deep learning: Methods and applications". In: *Foundations and Trends in Signal Processing* 7.3–4, pp. 197–387.

Erro, Daniel (2016). "Two-Band Radial Postfiltering in Cepstral Domain with Application to Speech Synthesis". In: *IEEE Signal Processing Letters* 23.2, pp. 202–206.

Erro, Daniel et al. (2011). "Improved HNM-Based Vocoder for Statistical Synthesizers." In: *INTERSPEECH*, pp. 1809–1812.

Fan, Yuchen et al. (2015). “Multi-speaker modeling and speaker adaptation for DNN-based TTS synthesis”. In: *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*. IEEE, pp. 4475–4479.

Fernandez, Raul et al. (2014). “Prosody contour prediction with long short-term memory, bi-directional, deep recurrent neural networks.” In: *Interspeech*, pp. 2268–2272.

Glorot, Xavier and Yoshua Bengio (2010). “Understanding the difficulty of training deep feedforward neural networks”. In: *International Conference on Artificial Intelligence and Statistics*, pp. 249–256.

Hastie, Trevor et al. (2005). “The elements of statistical learning: data mining, inference and prediction”. In: *The Mathematical Intelligencer* 27.2, pp. 83–85.

Hochreiter, Sepp (1998). “The vanishing gradient problem during learning recurrent neural nets and problem solutions”. In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02, pp. 107–116.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). “Long short-term memory”. In: *Neural computation* 9.8, pp. 1735–1780.

Hozjan, Vladimir et al. (2002). “Interface Databases: Design and Collection of a Multilingual Emotional Speech Database.” In: *LREC*.

Hu, Qiong et al. (2014). “An investigation of the application of dynamic sinusoidal models to statistical parametric speech synthesis.” In: *INTERSPEECH*, pp. 780–784.

Hu, Qiong et al. (2015). “Fusion of multiple parameterisations for DNN-based sinusoidal speech synthesis with multi-task learning”. In: *Proc. Interspeech*, pp. 854–858.

Hunt, Andrew J and Alan W Black (1996). “Unit selection in a concatenative speech synthesis system using a large speech database”. In: *Acoustics, Speech, and Signal Processing, 1996. ICASSP-96. Conference Proceedings., 1996 IEEE International Conference on*. Vol. 1. IEEE, pp. 373–376.

ITU-R (2003). *ITU-R BS.1534-1,Methods for the objective assessment of intermediate quality level of coding systems.*

ITU-T (2011). *ITU-T P.56, Objective measurement of active speech level.*

Jauk, Igor, Antonio Bonafonte, and Santiago Pascual (2016). “Acoustic feature prediction from semantic features for expressive speech using Deep Neural Networks.” In: *2016 24th European Signal Processing Conference (EUSIPCO) (EUSIPCO 2016).*

Jozefowicz, Rafal, Wojciech Zaremba, and Ilya Sutskever (2015). “An empirical exploration of recurrent network architectures”. In: *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pp. 2342–2350.

Kang, Shiyin, Xiaojun Qian, and Hsiang-Yun Meng (2013). “Multi-distribution deep belief network for speech synthesis”. In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, pp. 8012–8016.

King, Simon (2014). “Measuring a decade of progress in text-to-speech”. In: *Loquens* 1.1, e006.

Kingma, Diederik and Jimmy Ba (2014). “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980.*

Kubichek, Robert F (1993). “Mel-cepstral distance measure for objective speech quality assessment”. In: *Communications, Computers and Signal*

Processing, 1993., IEEE Pacific Rim Conference on. Vol. 1. IEEE, pp. 125–128.

LeCun, Yann A et al. (2012). "Efficient backprop". In: *Neural networks: Tricks of the trade*. Springer, pp. 9–48.

Ling, Zhen-Hua et al. (2015). "Deep learning for acoustic modeling in parametric speech generation: A systematic review of existing techniques and future trends". In: *IEEE Signal Processing Magazine* 32.3, pp. 35–52.

Lu, Heng, Simon King, and Oliver Watts (2013). "Combining a vector space representation of linguistic context with a deep neural network for text-to-speech synthesis". In: *Proc. ISCA SSW8*, pp. 281–285.

Mashimo, Mikiko et al. (2001). "Evaluation of cross-language voice conversion based on GMM and STRAIGHT". In:

Orr, Genevieve B and Klaus-Robert Müller (2003). *Neural networks: tricks of the trade*. Springer.

Pascual, Santiago and Antonio Bonafonte (2016a). "Multi-output RNN-LSTM for multiple speaker speech synthesis and adaptation". In: *2016 24th European Signal Processing Conference (EUSIPCO) (EUSIPCO 2016)*.

— (2016b). "Multi-output RNN-LSTM for multiple speaker speech synthesis with α-interpolation model". In: *9th ISCA Workshop on Speech Syntehsis*. Sunnyvale, CA, USA.

Qian, Yao et al. (2014). "On the training aspects of deep neural network (DNN) for parametric TTS synthesis". In: *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, pp. 3829–3833.

Recurrent Neural Networks Tutorial (2015). `http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/`. [Online; accessed February-2016].

Sorin, Alexander, Slava Shechtman, and Vincent Pollet (2011). "Uniform Speech Parameterization for Multi-Form Segment Synthesis." In: *INTERSPEECH*, pp. 337–340.

Srivastava, Nitish et al. (2014). "Dropout: a simple way to prevent neural networks from overfitting." In: *Journal of Machine Learning Research* 15.1, pp. 1929–1958.

Stylianou, I (1990). "Harmonic plus Noise Models for Speech, combined with Statistical Methods, for Speech and Speaker Modification (PhD, Signal et Image, ENST Paris, Paris)". In:

Sutskever, Ilya et al. (2013). "On the importance of initialization and momentum in deep learning." In: *ICML (3)* 28, pp. 1139–1147.

Swietojanski, Pawel and Steve Renals (2014). "Learning hidden unit contributions for unsupervised speaker adaptation of neural network acoustic models". In: *Spoken Language Technology Workshop (SLT), 2014 IEEE*. IEEE, pp. 171–176.

Tokuda, Keiichi et al. (2000). "Speech parameter generation algorithms for HMM-based speech synthesis". In: *Acoustics, Speech, and Signal Processing, 2000. ICASSP'00. Proceedings. 2000 IEEE International Conference on*. Vol. 3. IEEE, pp. 1315–1318.

Tutorial on Deep Belief Networks (2010). `http://deeplearning.net/tutorial/DBN.html`. [Online; accessed February-2015].

Tutorial on Denoising Autoencoders (DA) (2010). `http://deeplearning.net/tutorial/dA.html`. [Online; accessed February-2015].

Tutorial on Restricted Boltzman Machines (RBM) (2010). http://deeplearning.net/tutorial/rbm.html. [Online; accessed February-2015].
Understanding LSTM Networks (2015). http://colah.github.io/posts/2015-08-Understanding-LSTMs/. [Online; accessed February-2016].
UPC TTS Benchmark (2016). http://veu.talp.cat/neural_eval/. [Online; accessed June-2016].
Uria, Benigno, Iain Murray, and Hugo Larochelle (2013). "RNADE: The real-valued neural autoregressive density-estimator". In: *Advances in Neural Information Processing Systems*, pp. 2175–2183.
Uria, Benigno et al. (2015). "Modelling acoustic feature dependencies with artificial neural networks: Trajectory-RNADE". In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, pp. 4465–4469.
Valentini-Botinhao, Cassia, Zhizheng Wu, and Simon King (2015). "Towards minimum perceptual error training for DNN-based speech synthesis". In: *Proc. Interspeech*.
Wu, Zhizheng and Simon King (2015). "Minimum trajectory error training for deep neural networks, combined with stacked bottleneck features". In: *Proc. Interspeech*, pp. 309–313.
— (2016). "Investigating gated recurrent networks for speech synthesis". In: *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, pp. 5140–5144.
Wu, Zhizheng et al. (2015a). "A study of speaker adaptation for DNN-based speech synthesis". In: *Proceedings interspeech*.
Wu, Zhizheng et al. (2015b). "Deep neural networks employing multi-task learning and stacked bottleneck features for speech synthesis". In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, pp. 4460–4464.
Ze, Heiga, Andrew Senior, and Mike Schuster (2013). "Statistical parametric speech synthesis using deep neural networks". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, pp. 7962–7966.
Zen, Heiga and Hasim Sak (2015). "Unidirectional long short-term memory recurrent neural network with recurrent output layer for low-latency speech synthesis". In: *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*. IEEE, pp. 4470–4474.
Zen, Heiga and Andrew Senior (2014). "Deep mixture density networks for acoustic modeling in statistical parametric speech synthesis". In: *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, pp. 3844–3848.
Zen, Heiga, Keiichi Tokuda, and Alan W Black (2009). "Statistical parametric speech synthesis". In: *Speech Communication* 51.11, pp. 1039–1064.
Zen, Heiga et al. (2007). "The HMM-based speech synthesis system (HTS) version 2.0." In: *SSW*. Citeseer, pp. 294–299.