UNIVERSITY OF CRETE

DEPARTMENT OF PHYSICS

MSc THESIS

Modeling of Speech Signals using Recurrent Neural Networks

Aikaterini Papadaki

Advisor: *Dr.* Yannis Pantazis - Foundation for Research and Technology Hellas

UNIVERSITY OF CRETE
DEPARTMENT OF PHYSICS

MSc THESIS

Defended by
Author AIKATERINI PAPADAKI

Modeling of Speech Signals using Recurrent Neural Networks

COMMITTEE

Dr.	Yannis Pantazis	-	FOUNDATION FOR RESEARCH AND TECHNOLOGY HELLAS
Prof.	Iannis Kominis	-	UNIVERSITY OF CRETE
Prof.	Kostantinos Makris	-	UNIVERSITY OF CRETE

Date of the defense:
16/10/2020

Abstract

Speech is the primary means of communication between humans. A speech signal waveform corresponds to the variation of air pressure over time. Over two hundred years ago there have been early efforts to produce synthetic speech, using mechanical apparatus. Recently, ongoing recent activities have been focused on the artificial production of human speech, or else *speech synthesis*. Particularly, there have been developed *vocoders*, analyzers and synthesizers of human voice signals, based on mathematical models. Furthermore, recently neural vocoders, vocoders based on artificial neural networks, have introduced a new area in speech synthesis.

Goal of this study is to build a neural vocoder which combines deep learning with prior knowledge of the inherent sinusoidal nature of speech, in contrast with the majority of available neural based vocoders which discard signal processing in favor of neural networks. Particularly, the origin of the present study is modeling complex multi-component AM and FM sinusoidal waves with the property to represent speech signals, employing Recurrent Neural Networks (RNNs). The target is to develop a light neural vocoder, faster than WaveRNN, which achieves state-of-the-art performance. In the context of this thesis, we implement a variant of the WaveRNN model and we present the generated state-of-the-art results. Furthermore, we examine the proposed model's performance using synthetic as well as real speech signals.

Acknowledgements

First of all, I would like to thank my supervisor Dr. Yannis Pantazis for his guidance and support during my Master's thesis. He introduce me in the field of speech processing, advised me to attend courses and accord me useful studying resources. From the first day, he considered that nothing is granted and he explained me the next step patiently on the blackboard. I believe he is a great researcher and I am glad for our collaboration. Also, I would like to thank the Speech Laboratory, at the Computer Science department, for the computational resources. Furthermore, I would like to special thank all my master's classmates. During this year we spend a lot of time together, helping each other and having lots of fun inside and outside of the university. Of course I want to thank my friends Naya, Niki, Myrto, Dunker, Plat, Pit and Thomas. Finally, I thank my family, for growing me up and supporting my decisions. I feel really thankful for my parents and my siblings who constantly encourage me.

Contents

1 Introduction

Sinusoidal models exploit the quasi-periodic properties of voiced and unvoiced speech [13]. According to sinusoidal modeling, speech can be represented as a sum of sinusoids whose frequency is multiple to the fundamental frequency. In this study, we propose to develop a neural based sinusoidal system. In figure Fig. 1.1 there is a graphical representation of the proposed neural vocoder.

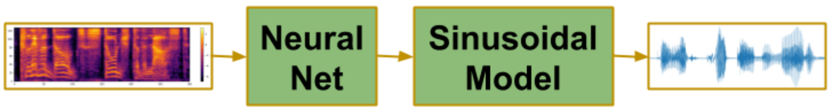


Figure 1.1: Proposed Lighter Neural Vocoder.

Aim of the thesis is to build a neural vocoder exploiting the sinusoidal nature of speech signals. Using Recurrent Neural Networks (RNNs) we target to develop a light neural vocoder, faster than WaveRNN [5] which achieves state-of-the-art performance. Current neural-based vocoders, such as WaveRNN, have millions of trainable parameters, thus the demand of computational resources and large high quality recording collections is increased. However, we target to significantly reduce the total number of vocoder parameters making it faster, without sacrificing the quality of the produced speech and requiring large recording sessions.

We used synthetic as well as real speech signals to examine the proposed model performance. Specifically, the first step was to implement the model on synthetic mono-component sinusoidal signals while the second step was to test our models accuracy for synthetic multi-component sinusoidal signals. Last but not least, we used real audio clips. Particularly, we implemented the decomposition of the speech signals using Filter Bank analysis and subsequently tested the proposed model under the assumption that each signal component is a mono-component sinusoidal signal. We extracted and analyzed the results both in time and frequency domain, as well as calculated the Relative Mean Squared Error(RelMSE) between the original and the generated signal.

Additionally, we examined the state-of-the-art performance of a WaveRNN variant model. Similarly, we used both a time and a frequency representation of the implementation results.

The present Master's thesis is organised as described below. First, in Chapter 2 there is a brief introduction on speech synthesis and a literature review on neural based speech synthesis. Moreover, there is an insight into the sinusoidal speech modelling analysis and into the Mel-Scaled spectrogram as a vocoder feature. In Chapter 3 we introduce Artificial Neural Networks and a special category of neural networks the Recurrent Neural Networks.

Continuing, in Chapter 4 we take a closer look at the architecture of a single output WaveRNN model. We implement a variation of the model and we present the extracted state-of-the-art results. Chapter 5 describes in detail the mathematical background and the structure of the proposed model as well as the implementation procedure. Specifically, we clarify how to extract from the data set the required information, so as to define the input, the output and the loss function of the neural model. In Chapter 6 there are represented and examined the results of our proposed model implementation on synthetic as well as real speech signals. In conclusion, Chapter 7 resumes the scope and the results of this study, while gives some directions for further research.

Additionally, appendix A explains the computational tricks required for the training and inference of the neural model, while appendix B contains the main Python codes we developed in the context of this study.

2 Speech Synthesis

2.1 General on Speech Synthesis

Speech signal waveform correspond to the variation of air pressure over time. Undoubtedly, it is the primary means of communication between humans. Aiming to improve our ability to communicate, there have developed techniques to measure, analyze and transform speech signals to other forms. The most popular case of this is the analog telephone handset. A telephone handset converts the time-varying air pressure modification caused by the speaker's lips into at time-varying electric voltage signal, which can be transmitted and re-transformed into speech pressure signal by the receiving handset.

Furthermore, over two hundred years ago there have been early efforts to produce synthetic speech, using mechanical apparatus. The previous decades, the area of digital speech processing has been greatly developed and gathered a lot of attention. A digital ,or else, a *discreet time speech signal* can be captured by taking samples of the air pressure over time with a specific sampling rate. Further more, speech signal *analysis/synthesis* systems have been advanced on the production of discrete-time speech signals. Specifically, in analysis, speech waveform underlying parameters are extracted, both in temporal and in spectral resolution, while in synthesis the waveform is reconstructed based on the parameter evaluation. Discrete time speech signal processing and analysis/synthesis systems have a great variety of applications laying on many different aspects like speech modification, speech enhancement, speaker/language recognition and speech coding.

A long-standing dream of human-computer interaction is to allowing people converse with machines. Ongoing recent activities have been focused on the ability of computers to understand natural speech and on the artificial production of human speech, or else *speech synthesis*. *Text to Speech (TTS) Synthesis* facilitates the human-machine communication by converting arbitrary text to speech signals.

For a long time the main methods of Text-to-Speech conversion were the *Concatenative TTS* and *Parametric TTS*. Particularly, in Concatenative TTS complete utterances are formed from a large database of short speech fragments, recorded from a single speaker. However, synthesizing speech using this method is time consuming while modifying the voice to a different speaker or emotion, requires to record a whole new database.

On the other hand, Parametric TTS is a more statistical method that addresses the limitations of concatenative TTS. Using digital speech signal processing techniques the text is processed to extract firstly *linguistic features*, such as phonemes and duration, and secondly *vocoder features*, such as the spectrogram and the fundamental frequency. Vocoder features enclose inherent characteristic of human speech and along with the linguistic features are fed into a *Vocoder*. A *Vocoder* is an analyzer and synthesizer of human voice signals. In this case it is a mathematical model based on Hidden Semi-Markov processes which estimates speech parameters, such as phase, speech rate and intonation, and finally generates the speech signal waveform.

2.2 Neural Based Speech Synthesis

Recently, the intensive research activity around the field of *Neural Networks* and *Deep Learning* has added a new perspective to the problem of speech synthesis. Innovative TTS systems based on neural network models constitute a major breakthrough in terms of high quality synthetic speech. Specifically, in speech synthesis the term quality implies speech characteristics such as naturalness, intelligibility and speaker recognizability.

The majority of state-of-the-art neural TTS systems are organized into two separate neural networks, as illustrated in figure Fig. 2.1. Specifically the first neural model maps a sequence of characters to a sequence of spectral vectors. The most successful text-to-spectra model has been proposed by DeepMind and is called Tacotron [2], [3] . The second neural model is actually a neural vocoder, that takes as input the spectral representation and generates the conditional speech signal. The WaveNet [4] was the first neural vocoder and established a new era on high quality speech production, while the WaveRNN [5] and LPCNet [6] followed.

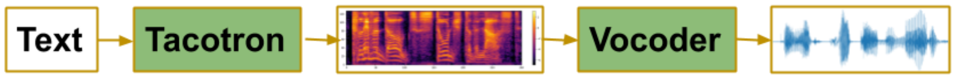


Figure 2.1: Typical Neural TTS System.

The above neural vocoders are autoregressive models. The term *autoregressive* implies models where the output speech signal sample depends linearly on previous sample values, resulting in slow speech waveform generation. To that end the above mentioned neural vocoders are *slow inference*. On the other hand, the WaveGlow [7] and MelGAN [8] are fast inference , however generate lower speech quality, in comparison to autoregressive neural vocoders.

Text to Speech Synthesis is a highly active field which has gathered a lot of attention. There is a great variety of applications and recent developments. Some examples include variants of WaveRNN model such as universal vocoders [9] as well as systems capable of generating highly intelligible speech in the presence of noise [10], indispensable for modern mobile devices. Some approaches have employed the inherent nature of speech, such as source-filter neural synthesis [11]. Also, the sinusoidal modeling of speech due to its flexible structure has been used for research activities around speech transformations [12].

In the following sections we will get insights into the sinusoidal speech modelling and the Mel-scaled spectrogram as a vocoder feature, elements that constitute the proposed vocoder (Fig. 1.1).

2.3 Sinusoidal Speech Modelling

In this paragraph, there is a brief summary of sinusoidal speech modeling [14]. To begin with, in the context of sinusoidal modeling, a speech signal $s(t)$ can be represented as the sum of *amplitude-modulated* (AM) and *frequency modulated* (FM) sinusoidal waves given by

$$s(t) = a_0(t) + \sum_{k=1}^{K(t)} 2a_k(t)cos(\varphi_k(t)) = \sum_{k=-K(t)}^{K(t)} \alpha_k(t)e^{i\varphi_k(t)}, \tag{2.1}$$

where $\alpha_k(t)$ is the time-varying amplitude and $\varphi_k(t)$ is the time-varying phase of the k^{th} signal component. The amplitude $\alpha_k(t)$ and phase $\varphi_k(t)$ components are slow varying quantities. Additionally, it is noteworthy that the number of components $K(t)$ is a time-varying quantity. This is an indispensable property for the representation of *non-stationary* signals, such as speech and music.

In *frame-by-frame* sinusoidal signal analysis, the signal $s(t)$ is divided into temporal segments called frames and denoted by

$$s_l(t) = s(t - t_l)w(t), \tag{2.2}$$

where t_l corresponds to the center of the frame and $w(t)$ to the analysis window function with $t \in [-T_l, T_l]$. The window function typically vanishes at the boundaries of the frame it is applied on. This structure aims to alleviate discontinuities, as well as to eliminate the interference between the components.

Frame-by-frame sinusoidal signal analysis was developed under the assumption that each frame is constructed by stationary components. Particularly, each frame is a *superposition* of sinusoidal waves with constant amplitudes and constant frequencies. To that end each frame is modeled as

$$h_s(t) = \sum_{k=-K}^{K} \alpha_k e^{i2\pi f_k t} w(t), \tag{2.3}$$

where $t \in [-T, T]$, K is the local number of components, f_k and a_k are the local frequency and local amplitude of the k^{th} sinusoidal component of the particular frame, respectively. The subscript l that denotes each particular frame, as in equation Eq. 2.2, is disregarded for simplicity.

A considerable body of literature in the field of speech signal processing get insights into the estimation of the unknown sinusoidal parameters. *In this study the evaluation of those unknown sinusoidal parameters is implemented employing neural network based tools.*

2.4 Mel-Scaled Spectrogram

In the previous sections we mentioned the Mel-scaled spectrogram as a vocoder feature. In this paragraph, we will have a brief discussion about the Mel-scaled spectrogram origin and an insight into its generation procedure.

So as to produce the Mel-scaled spectrogram of a signal the first step is to divide the signal $s(t)$ into short frames $s_i(n)$, where n indicates the number of samples included in the frame and i the number of frames. Particularly, an audio signal corresponds to a complex dynamic timeserie which value changes constantly. Therefore, for simplicity we can assume that on short time scales the audio signal statistically stationary. The duration of a typical frame is usually around 20-40ms. A shorter frame leads deficient number of signal samples for reliable spectral estimate, while a longer frame may capture shifted signal. After slicing the signal into frames, a window function $h(n)$ is applied on each frame. Typically, a window vanishes at the boundaries of the frame, such as the bell-shaped Hann window

$$h(n) = \frac{1}{2}\bigg(1 + cos(\frac{2\pi n}{N})\bigg), \tag{2.4}$$

where 0 $\leq$ n $\leq$ N-1 , N is the window length.

Continuing, the next step is to estimate the power spectrum of each frame. The periodogram-based power spectrum imitates the function of a cochlea and identifies which frequencies are present in a specific frame. Specifically, the *cochlea* is an ear organ which vibrates at different spots, depending on the sound frequency. To that end the cochlea informs the brain of the incoming sound frequencies values.

Particularly, the periodogram estimate of the power spectrum is given by the squared absolute value of the complex valued Fourier Transform of each frame. The K-point Discrete Fourier Transform (DFT) of each signal frame $S_i(k)$, or else the Short Time Fourier Transform (STFT) of the signal, generates the signal frequency spectrum and is given by the following formula

$$S_i(k) = \sum_{n=1}^{N} s_i(n)h(n)e^{-i2\pi kn/N}, \tag{2.5}$$

where 1 $\leq$ k $\leq$ K , i denotes the frame number, h(n) is an N sample long window function and K is the length of the DFT. Note that, windows functions reduce spectral leakage. Their necessity lays to the assumption made by Finite Fourier Transform (FFT) that the signal is infinite. To that end the periodogram-based power spectrum for each speech signal frame is given by

$$P_i(k) = \frac{1}{N}|S_i(k)|^2, \tag{2.6}$$

The final step to computing the Mel-scaled spectrogram of the signal is to design and apply to the power spectrum a Mel-scaled Filter Bank. The Mel-scale is used to imitate the human ear perception of sound, which is non-linear. Equations Eq. 2.7 convert Hertz (f) to Mel (m) and inverse Mel (m) to Hertz (f)

$$\begin{aligned} m &= 2595log_{10}\left(1 + \frac{f}{700}\right) \\ f &= 700(10^{m/2595}). \end{aligned} \tag{2.7}$$

A Mel-scaled filter bank is more discriminative at lower frequencies and less discriminative at higher frequencies , as illustrated in figure Fig. 2.2.

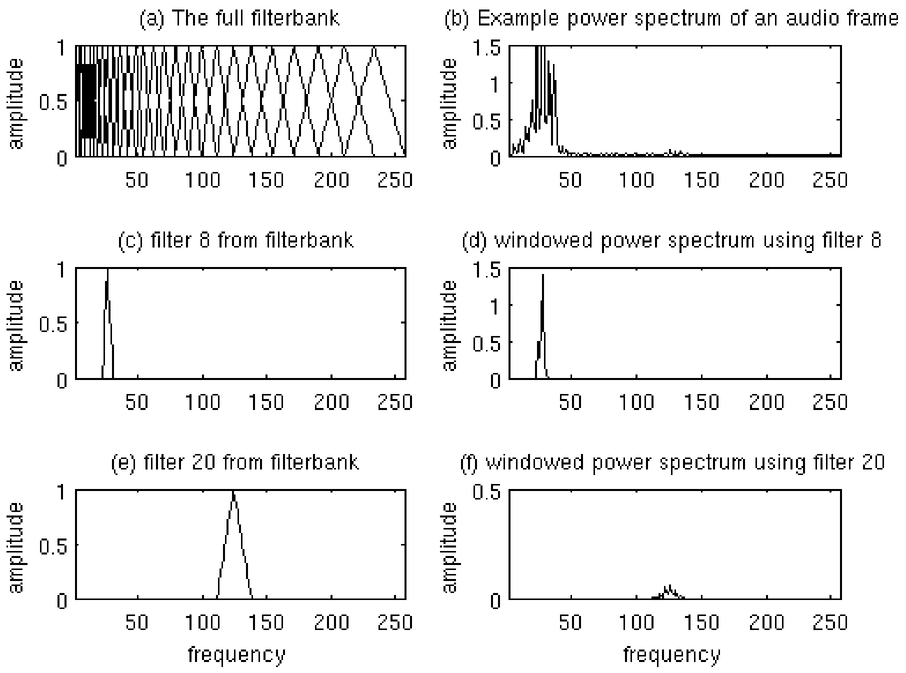


Figure 2.2: Plot of Mel Filter-bank and windowed power spectrum, [1].

Particularly, in the filter bank each filter can be mathematically modeled by the following equation

$$H_m(k) = \left\{ \begin{array}{lr} 0, & \text{for } k < f(m-1) \\ \frac{k-f(m-1)}{f(m)-f(m-1)}, & \text{for } f(m-1) \leq k \leq f(m) \\ \frac{f(m+1)-k}{f(m+1)-f(m)}, & \text{for } f(m) \leq k \leq f(m+1) \\ 0, & \text{for } k > f(m+1) \end{array} \right\}$$

Eventually, we can extract the frequency bands that constitute the *Mel-Scaled Spectrogram* (Fig 2.3), by applying the Mel-scaled filter bank on the power spectrum.

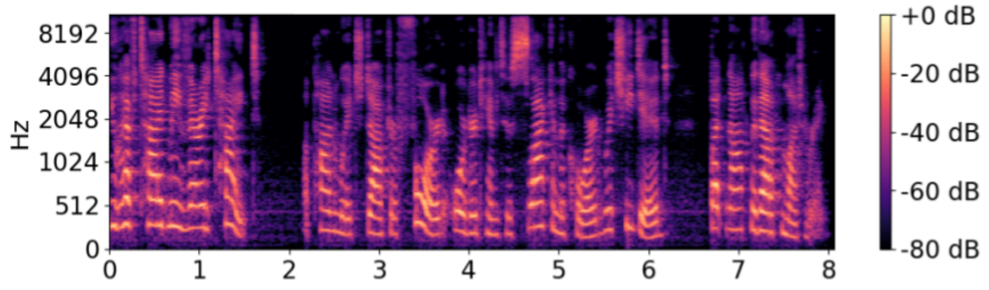


Figure 2.3: Example of a Mel-Scaled Spectrogram.

3
Recurrent Neural Networks (RNNs)

3.1 ANN Overview

Artificial Neural Networks (ANN) is an information processing paradigm that is inspired by the way the biological nervous system, such as brain, process information [16]. It is a technique that teaches computers a behaviour that rises naturally to humans, to learn by example. An Artificial Neural Network model is composed of a large number of highly interconnected processing elements, called neurons. These elements work as a unison so as to face a particular problem.

In this paragraph, we will introduce Artificial Neural Networks starting with the the Perceptron, the first structrure of an artificial neuron which was inspired by the biological neuron. We will describe the structure of a neuron and discuss how it is able to learn, by examining the back-propagation and gradient decent algorithms.

3.1.1 Perceptron

The biological neurons are the nerve cells that constitute a nervous system, like the brain. They are able to receive information from the external of the cell word, process those input information and finally transmit the resulted output.

In Fig. 3.1 there is a graphical representation of a biological neuron. It consists of four parts, the dendrites,the soma, the axon and the synapses. The dendrites are hair-like tubular extensions around the neuron that are able to receive incoming electrical signals, the soma contains the cell nucleus, which executes necessary biochemical transformations, the axon is a long, thin, tubular structure responsible for transmitting information and the synapses are complex cell branches that are capable of connecting neurons to each other. As a whole, at a biological neuron the dendrites collect input electrical signals provided by the synapses of other neurons, the soma processes those incoming information and derives an output signal, which is transmitted to other neurons through the axon and the synapses.

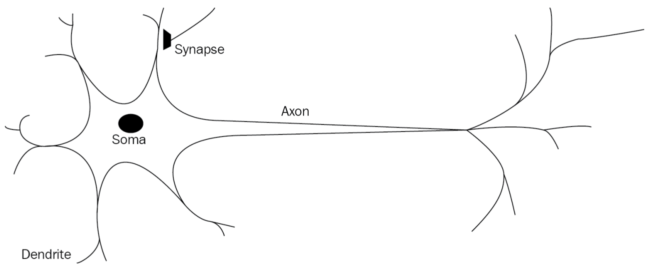


Figure 3.1: A biological neuron. It consists of four parts, the dendrites,the soma, the axon and the synapses.

The first ANN model was inspired by the biological neuron. It was invented in 1957 by Frank Rosenblatt and was called Perceptron. The Perceptron is an artificial neuron that mathematically models a biological neuron, imitating the procedures that take place at each of its parts. At Fig. 3.2 there is a biological neuron connected to an artificial neuron, representing their similarity.

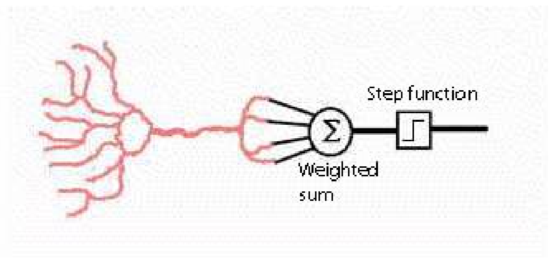


Figure 3.2: A biological neuron connected to an artificial neuron.

In the biological neurons electrical signals are transmitted through the axons to dendrites. Imitating this procedure, in the perceptron these electrical signals are represented as numerical values. The strength of each input electrical signal is modulated, at the synapses between the dendrite and axons. Correspondingly, in the perceptron this operation is mathematically modeled as the multiplication of each numerical input value with a weight. When the total strength of all the input signals exceeds a certain threshold, the neuron fires an output signal. In a perceptron the output is determined by calculating the weighted sum of all the inputs and applying a step function on the sum. Similarly to biological neurons, this output is fed to other perceptrons.

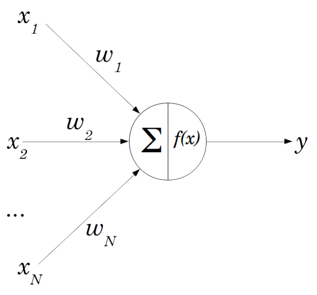


Figure 3.3: A Perceptron. $x_1, x_2, ..., x_N$ the input to the neuron values, $w_1, w_2, ..., w_N$ the synaptic weights, *sum* the weighted sum of the inputs, $f(x)$ the step function and y the output of the neuron.

In the above figure Fig. 3.3, there is a perceptron representation, where $x_1, x_2, ..., x_N$ are the input to the neuron values, $w_1, w_2, ..., w_N$ are the synaptic weights, *sum* is the weighted sum of the inputs, $f(x)$ is the step function and y the final output of the neuron. The weights w are multiplied with each input x, determining the strength of a particular node. The neuron's output y is the step function of the sum of these products plus a bias b value :

$$y = f(\sum_{i=1}^{N} w_i * x_i + b_i) \tag{3.1}$$

There are used different types of step functions, depending on the problem an ANN model is trying to solve. A step function is also called activation function, since it decides whether a neuron should be activated or not. Additionally, applying an activation function includes non-linearity to the neuron's output. Since, Artificial Neural Networks are considered to be "Universal Function Approximators"[], non-linearity is necessary.

3.1.2 Feed-Forward Neural Networks

Perceptrons arranged in layers can form a neural network. In Fig. 3.4, there is presented computational graph of a feed-forward neural network with one hidden layer. Particularly, a feed-forward neural network consists of the input layer, a number of hidden layers and the output layer. The input layer recieves the incoming inputs and the output layer produces the outputs. Since, the middle layers have no interaction with the external of the network environment, they are called hidden.

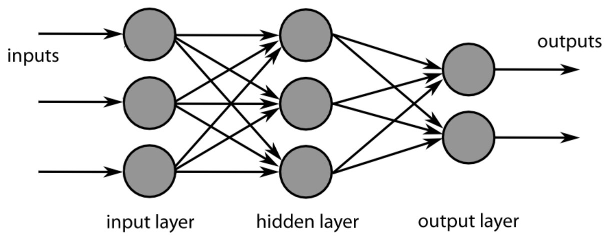


Figure 3.4: A feed-forward neural network, with one single hidden layer.

On one single layer, there is no connection among perceptrons. However, on one single layer each perceptron is connected to every perceptron, on the next layer. To that end, the incoming information is travelling from one layer to the next, and hence it is "fed forward" to the network.

3.1.3 Learning in Feed-Forward Networks

There are different types of learning in neural networks, depending on the nature of the problem the network is trying to solve. Feed-forward networks use supervised learning, a type of machine learning where the network tries to find a function that best maps an input to an output, based on pairs of input and output example values.

These examples of input-output pairs are called training samples and the whole learning procedure *training*. The training samples that are provided to the network, consist of an input vector x and its desired output vector y. During the training procedure of supervised learning, the input vector x is fed into the network and the network generates an output vector $\widetilde{y}$. The output vector $\widetilde{y}$ is compared to the desired output vector y. The squared difference of the generated output vector $\widetilde{y}$ and the desired output vector y defines the *loss function* J Eq. 3.2 :

$$J = \sum_{i=1}^{N} |\widetilde{y}_i - y_i|^2 \tag{3.2}$$

The value of J quantifies the degree of divergence from the desired result. In Eq. 3.4, there is the expression of the loss function J in terms of the input values x_i, the weights w_i, the bias b_i and the desired output values y_i, of each training sample:

$$J(w, b) = \sum_{i=1}^{N} |(w_i * x_i + b_i) - y_i|^2 \tag{3.3}$$

The next step, after feeding to the network the training samples and defining the loss function is the *backpropagation* of errors. Backpropagation is a procedure which aims in finding the global minimum of the loss function J. This goal is approached following the *gradient descent* algorithm. According to gradient descent algorithm, decreasing the value of the weights w_i in the direction of the gradient of the loss function J, finally leads to the most rapid minimization of J. To that end, after feeding the network with all the training samples and computing the loss function, we go backwards and update all the weight w_i and bias b_i values.

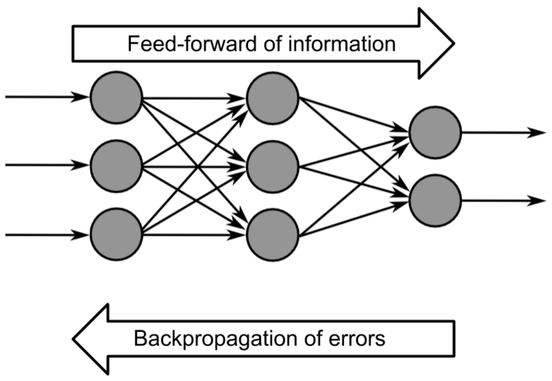


Figure 3.5: A feed-forward neural network.

Each weight w_i and bias b_i is updated using the following formulas :

$$w_i = w_i - \alpha \frac{\partial J}{\partial w_i} \tag{3.4}$$

$$b_i = b_i - \alpha \frac{\partial J}{\partial b_i} \tag{3.5}$$

, where α is a model hyperparameter. It defines the learning speed, or in other words how big the step towards the minimum of J will be.

This procedure is repeated for many cycles. Repetitively, the training samples are fed to the network and going backwards the weight and bias vectors are updated based on the derivatives of the loss function, so as to gradually decrease J to the minimum possible value.

3.2 Recurrent Neural Networks

As discussed in the previous section, a simple feed forward neural network is able to learn efficiently the relation between a set of inputs and outputs. However, in the case of series type inputs, the efficiency of a feed forward neural network is insufficient.

In a series type set of input training data, each element has a relation with the other elements of the sequence and probably an influence on its neighbour elements. The mechanism implemented by a feed forward neural network, is not made for capturing sequential relationship across inputs. It does not use previous sequence elements to generate predictions for the future sequence elements. Such problems where the order of events is important for predicting next events are addressed to a special category of neural networks, the Recurrent Neural Networks (RNN). The architecture and function of RNNs will be examined in the sections below.

3.2.1 Vanilla Recurrent Neural Networks

Recurrent Neural networks are a special category of neural networks, which are able to recognize sequence patterns. They allow information to persist during training process using loops.

In Fig. 3.6 there is a computational gragh of a recurrent neural network, where x is the input state, h the hidden state, o the output state and U, V, W are the weights of

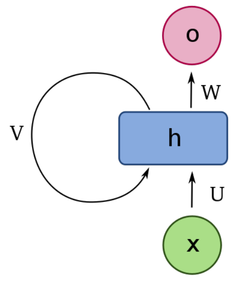


Figure 3.6: A Recurrent neural network, x the input state, h the hidden state, o the output state and U, V, W the weights, image by Ixnay.

the network. The loop allows information to flow from the one step to the next, letting the hidden state h carry information from all the previous steps.

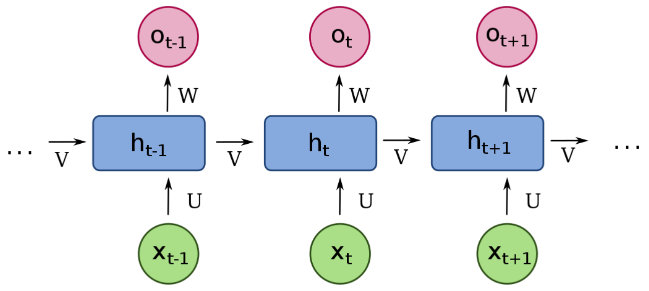


Figure 3.7: A Recurrent neural network unfolded, image by Ixnay.

In the above figure Fig. 3.7 there is an equivalent representation of a recurrent neural network, this time displayed with the loop unrolled. Practically, it is a neural network which is not only fed the current input x_t, but also the hidden state of the previous step h_{t-1}. Specifically, the output of a RNN at a specific step o_t is given by the following set of formulas Eq. 3.6:

$$h_t = f_h(U * x_t + V * h_{t-1} + b_h) \tag{3.6}$$

$$o_t = f_o(W * h_t + b_o) \tag{3.7}$$

, where f_h the hidden state activation function, b_h the hidden state bias, f_o the output state activation function and b_o the output state bias.

This chain-like nature reveals that recurrent neural networks are intimately related to sequences and lists. The chain-like architecture of recurrent neural network eflects their intimate relation to sequence data. There are various different forms of sequence data. For instance audio is a natural sequence, which can be split and fed into a RNN, Fig. 3.8.

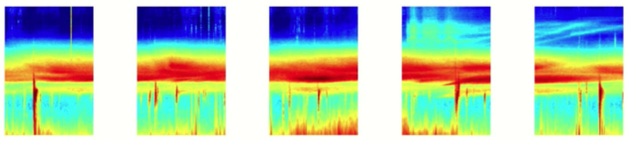


Figure 3.8: A split audio spectrogram, image by Michael Phi.

3.2.2 Long Short term Recurrent Neural Networks

Although Recurrent Neural Networks are able to handle sequential data, as a RNN processes more steps of a sequence, retaining information from previous steps becomes more difficult. This difficulty is known as *short-term memory* problem and is due to the nature of back-propagation, which was described in the previous section. Particularly, the short-term memory of the network is caused by the appearance of *vanishing gradients*, during the back-propagation through time procedure. Gradually, the gradient values exponentially shrink, as it propagates to previous steps of the input sequence. Small gradients mean small adjustments in the neural networks weights, causing the network to not learn and forget the information provided from the early steps of the sequence.

In terms of solving the short-term memory problem, there were created the Long Short term Recurrent Neural Networks (LSTMs). Due to an internal mechanisms called gates, LSTMs can learn which data in a sequence is important to keep or throw away, regulating the flow of information inside the network. At the following paragraphs, we will take a closer look on their architecture and function.

All types of recurrent neural networks have a chain-like form of repeating modules of neural network. In vanilla RNNs, this repeating modules have the simple structure of a single activation function layer.

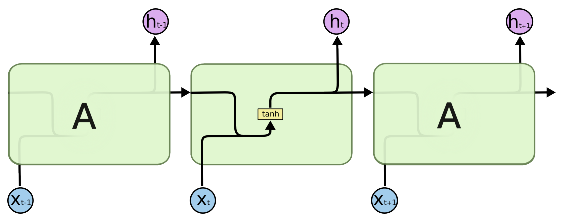


Figure 3.9: The repeating module in a vanilla RNN, image by Christopher Olah.

In long-short term memory RNNs this chain like repeating module consists of four interacting neural network layers. A computational graph of the inside structure of each LSTM module is presented in Fig. 3.10. As it is illustrated in this gragh, the output of one LSTM node is fed as input to the other LSTM nodes. Regarding the notation used, each line symbolizes vector transfer,the pink circles correspond to point-wise operations, like vector addition and vector multiplication, the yellow rectangles are learned neural network layers, the merging lines denote concatenation and a forking line denotes copying of its content and redirection of each copy to different locations.

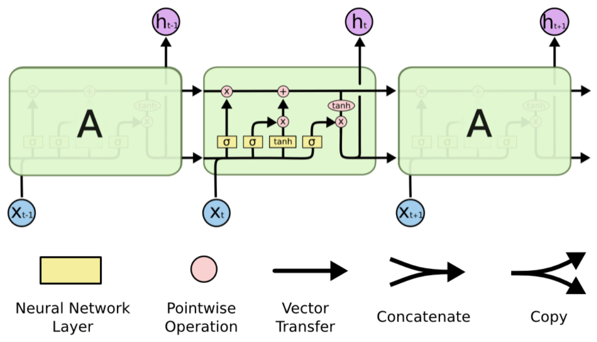


Figure 3.10: The repeating module in an LSTM, image by Christopher Olah.

The core idea behind the architecture of LSTMs is the cell state. It is the horizontal line running through the top of the diagram and crosses the entire chain of nodes in the network, Fig. 3.11. It is a long-term state which gives the network the ability to learn during the training procedure what information to add or what information to remove. This is regulated by structures called gates.

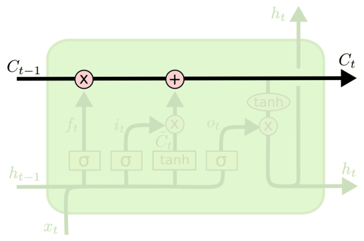
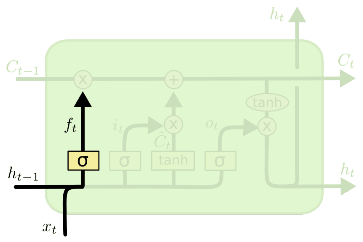


Figure 3.11: The cell state of an LSTM node, image by Christopher Olah.

Specifically, gates are structures that optionally let information through. It consists of a sigmoid neural network layer and a pointwise multiplication operation. Inside an LSTM node there are three gates which control the information flow of the cell state, the *forget gate*, the *input gate* and the *output gate*.

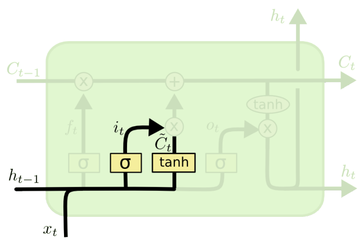
On the left side of Fig. 3.12 there is a graphical representation of the forget gate. Additionally, on the right side of Fig. 3.12 there is a formula that mathematically models this gates behaviour. The forget gate is responsible for the decision of what information are going to be thrown away from the cell state. Information from the previous hidden state h_{t-1} and from the current input x_t is passed through a sigmoid funcion. The output of the sigmoid layer f_t are numbers between 0 and 1, the closer to 1 means to keep and the closer to 0 means to forget.



$$f_t = \sigma \left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f \right)$$

Figure 3.12: The forget gate layer of an LSTM, where f_t is the forget gate output, h_{t-1} is the previous hidden state, x_t the current input and σ is a sigmoid funcion, image by Christopher Olah.

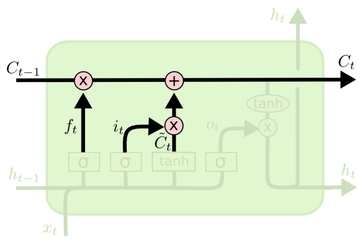
Continuing, in Fig. 3.13 there is a graphical representation of the input gate and the formula that models its function. The role of the input gate layer is to update the cell state. So as to determine which values will be updated, the previous hidden state and the current input is passed through a sigmoid function. The output values are between 0 and 1, where 0 means the value is of a less importance and does not need to be stored and 1 means the value is of a great importance and needs to be stored in the cell state for the next steps. Furthermore, the previous hidden state and the current input is passed thought a *tanh* function, taking values between -1 and 1. The output values are the new candidate cell state $\hat{c}_t$, that could be added to the cell state.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \;+\; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \;+\; b_C)$$

Figure 3.13: The input gate layer of an LSTM, where i_t is the input gate output and $\hat{c}_t$ candidate cell state, image by Christopher Olah.

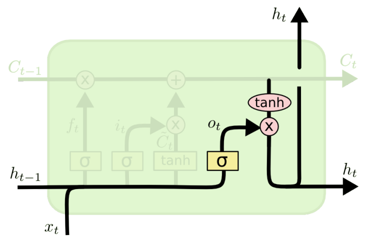
The final update of the old cell state into the new cell state c_t is illustrated in Fig. 3.13 and is mathematically described by the respective formula. Specifically, the state forgets what it has to forget as it was decided by the forgate gate, by multiplying the old cell state by the forget gate output vector. The next step for the complete update of the state is to add the multiplication of the input gate output and the candidate cell state, particularly the information that the sigmoid output decided that are important to keep from the *tanh* output.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure 3.14: The new cell state c_t of an LSTM, image by Christopher Olah.

Last but not least is the output gate. The output gate determines the next hidden state that will be fed to the next LSTM node. As mentioned above, the hidden state is of a great importance as it includes information from previous inputs and is useful to make predictions. The first step is to pass the previous hidden state and the current input into a sigmoid function. Then, the updated new cell state is passed througth a tanh function. The tanh output and the sigmoid output are multiplied, producing the new hidden state h_t.



$$o_t = \sigma\left(W_o \ [h_{t-1}, x_t] \ + \ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figure 3.15: The output gate layer of an LSTM, where o_t is the output gate output and h_t the hidden state, image by Christopher Olah.

The next LSTM node receives the new cell state and the new hidden state. Along with the new input value x_t, it repeats the same procedure so to update the cell state and produce the new hidden state.

4 Deep Neural Network based Speech Synthesis

Recently, several deep neural network models demonstrate the ability to synthesize high quality speech by generating raw audio waveforms. There are multiple applications of neural network based speech synthesis models. Among the most popular paradigms is the text-to-speech synthesis, speech-to-speech synthesis, speech compression, voice conversion and speech enhancement.

In natural speech generation, deep neural network models with sequential structure achieve state-of-the-art performance. In the paragraphs below we will have a closer look at such a model, named *WaveRNN* [5]. First, we will describe the architecture of a WaveRNN model and secondly we will present a WaveRNN variant on a speech-to-speech synthesis paradigm and demonstrate the generated results.

4.1 Definition of a Single Output WaveRNN

In natural speech generation, sequential generative models are widely used, because of their high performance in that field. Specifically, sequential models factorize the data distribution into a product of conditional probabilities over each sample. In that way, they learn the joint probability of the data and thus the distribution of the speech signal waveform.

Recurrent neural networks are especially suitable for this scope. As clarified in the previous chapter, the main characteristic of a RNN is that a non-linear transformation of the information a state carries, is delivered to the next state, through a recurrent layer. In that way, the model is able to detect and learn complex relation among the data spontaneously, because of its structure itself.

Below we will examine the structure and the performance of the *WaveRNN* model. The *WaveRNN* was proposed in [5] and is a single-layer RNN composed of two feed forward layers followed by a dual softmax layer. Additionally, the WaveRNN is a model of great interest as it matches the quality of the state-of-the-art WaveNet [4] model, a model which uses Convolutional Neural Networks (CNNs) instead of RNNs. The WaveRNN was initially proposed as a 16-bit model and was split as 8 coarse bits and 8 fine bits. It was taking as input the coarse and fine parts of an audio sample and generating as output the probability distributions for the coarse and fine parts of the next sample. In this summary, the coarse/fine split will be neglected. Our aim is to examine and clarify a simple version of the model with *one single output*.

In Fig. 4.1 there is a graphical representation of a single output WaveRNN. The previous audio sample s_{t-1}, along with conditioning parameters that correspond to spectral information S, is fed as input to the model and as output a discrete probability distribution for the next audio sample $P(s_t)$ is generated. It is mainly composed of a gated recurrent unit (GRU), two fully-connected layers and a softmax activation function. A GRU unit is a recurrent neural network unit, similar to the LSTM described in the previous chapter, but with simpler structure. The W and U matrices shown in Fig. 4.1, are the GRU weights.

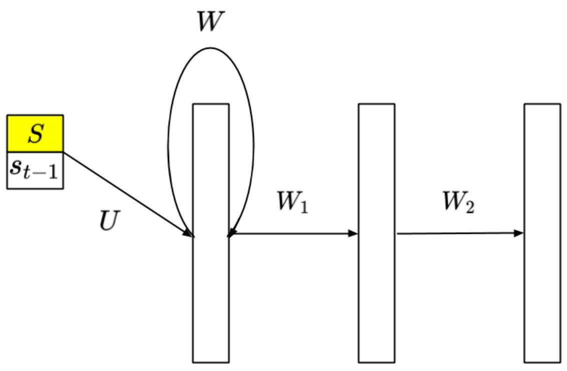


Figure 4.1: WaveRNN with a single output, s_{t-1} is the previous audio sample, S the previous audio sample conditioning parameters, W and U the GRU weight matrices, [5].

The following set of equations models the overall computation in a single output WaveRNN unit:

$$x_t = [s_{t-1}; S] \tag{4.1}$$

$$u_t = \sigma\left(W^{(u)} \cdot h_{t-1} + U^{(u)} \cdot x_t\right) \tag{4.2}$$

$$r_t = \sigma\left(W^{(r)} \cdot h_{t-1} + U^{(r)} \cdot x_t\right) \tag{4.3}$$

$$\tilde{h_t} = tanh\left(r_t \circ W^{(h)} \cdot h_{t-1} + U^{(h)} \cdot x_t\right) \tag{4.4}$$

$$h_t = u_t \circ h_{t-1} + (1 - u_t) \circ \tilde{h_t} \tag{4.5}$$

$$P(s_t) = softmax(W_2 ReLu(W_1 \cdot h_t)), \tag{4.6}$$

where the biases are discarded for clarity, the sigmoid function is equal to $\sigma(x) = \frac{1}{1+e^{-x}}$ and the $\circ$ indicates an element-wise vector multiplication. The $W^{(u)}, W^{(r)}$ and $W^{(h)}$ matrices, which form the W matrix, are computed as a single matrix-vector product. Each one of these matrices corresponds to the contribution to the two gates u_t, r_t and the candidate hidden state $\tilde{h_t}$ respectively, as similarly implemented inside a GRU cell. Sampling from the probability distribution $P(s_t)$ outputs the desired synthesized audio sample s_t.

Additionally, the sampling of an audio sample s_t depends on the previous samples values. To that end, the sampling procedure is strictly serial. In other words, a particular sample can be generated as long as the previous samples it depends on have been generated, too.

4.2 Implementation of a WaveRNN Variant

Figure Fig. 4.2 presents the computational graph of a variant of the WaveRNN model [17]. The main core of its architecture is like a single output WaveRNN model, constructed mainly by a GRU unit, two fully connected layers and a softmax layer, as described in the previous paragraph. In addition, there are more layers before the GRUs, building a more complex model. Below we will take a closer look at the function of each layer and the overall computational flow.

The model is constructed out of a set of FC and GRU layers, following the structure demonstrated at Fig. 4.2. An FC layer is a *Fully Connected Layer*, specifically a layer

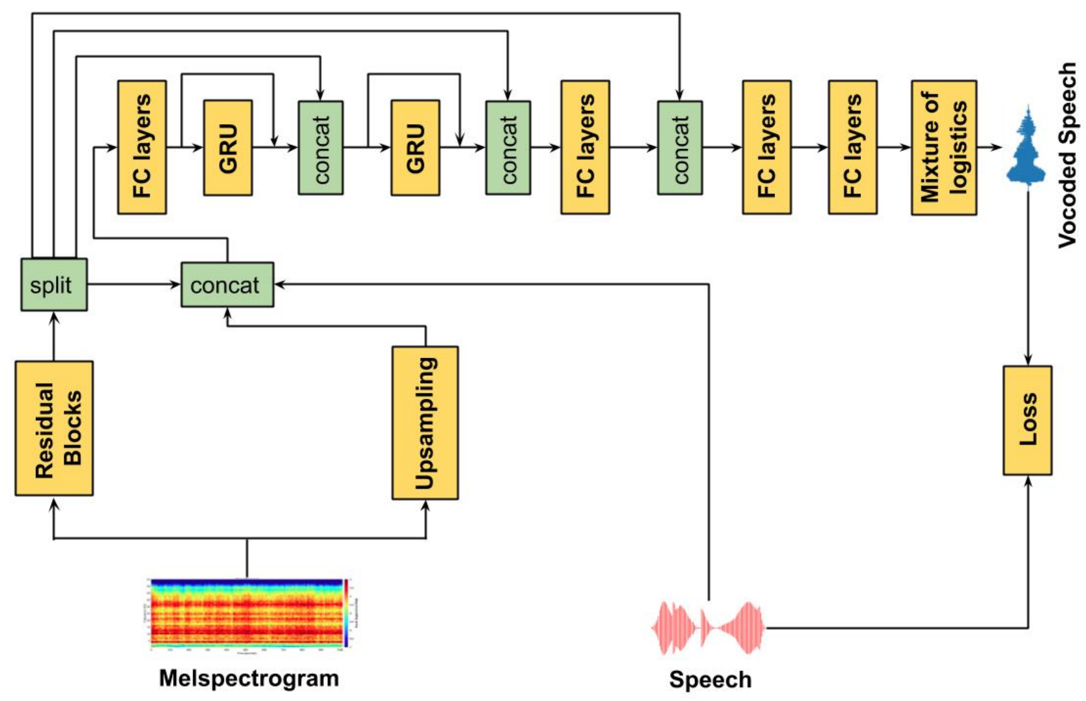


Figure 4.2: Variant of WaveRNN Diagram, [17] .

which connects all the inputs it receives to every activation unit of the next layer. After the input information passes through all the structure of layers, is fed into a mixture of logistics. For instance, in the general case of a single output WaveRNN, discussed in the previous paragraph, the mixture of logistic was a softmax and a ReLu layer. The output of the mixture of logistic is the generated vocoded speech.

To begin with, as illustrated in Fig. 4.2, the Mel-scaled spectrogram of a speech signal gets though an *upsampling* process and through a layer of *residual blocks*. Particularly, the melspectrogram of a speech signal is a colormap which describes the frequency amplitude modulation over time and is generated by applying *Short Time Fourier Transform* (STFT) to the signal. Also, this colormap is rescaled into a logarithmic scale, called the Mel scale, examined in the Chapter 2 of this study. The residual blocks layer, allows the activation from a previous layer to be fast-forwarded to deeper layers in the neural network. Finally, the upsampling indicates a procedure where the melspectrogram is upsampled, so as to generate a corresponding frequency band for each timestep of the speech signal.

Initially the conditioning vector which corresponds to information derived from the melspectrogram, is divided into parts. Those parts are used to transfer information of the melspectrogram into other layers deeper into the network. Furthermore, the upsampled melspectrogram along with information from the initial speech signal and the initial spectrogram are concatenated and fed into the model.

During the training procedure, the model learns how to synthesize speech by computing the *loss* between the vocoded output speech signal and the initial input signal. Since the aim is to predict the probability distribution of the next sample, it is a classification problem. To that end the loss function used is the *Cross-entropy*, or log loss, which measures the performance of a classification model whose output is a probability value between 0 and 1. The code for the implementation of this model is open source and available online on github [17].

4.3 WaveRNN Variant Results

In this paragraph we will present the state-of-the-art performance of the WaveRNN model described in the previous section. The code for the implementation of this model is open source and available online on github [17].

Particularly, we used single-speaker audio clips with sampling rate f_s equal to $22050 samples/second$ collected from a public domain speech dataset, *The LJ Speech Dataset* [18]. Also, the neural model was trained for 5000 epochs, with batch size equal to 64 and the training procedure duration was about 3,5 days. Additionally, as mentioned above the sampling procedure is serial. Particularly, the prediction of audio sample depends on the prediction of the previous sample, with respect to the ordering.

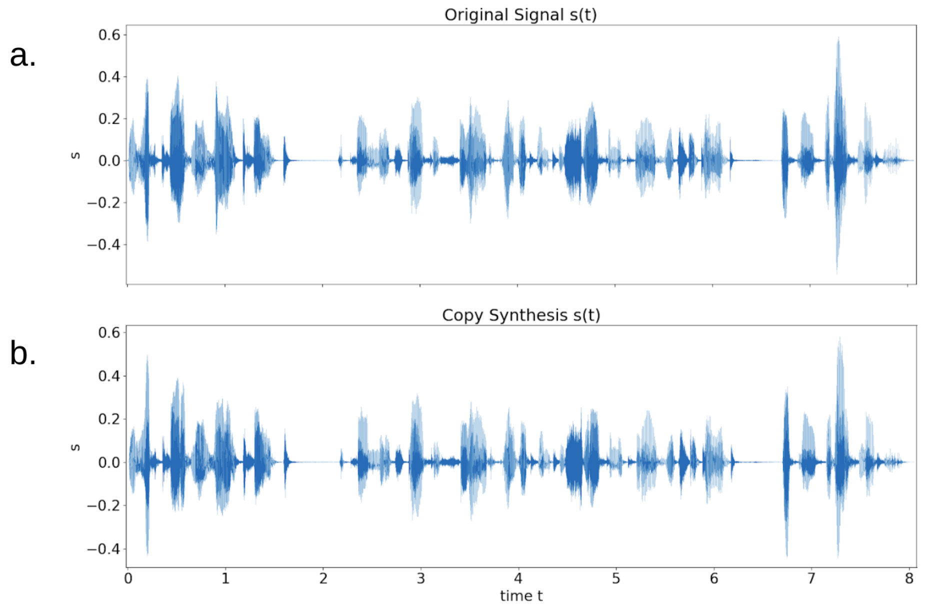


Figure 4.3: Time representation of a test speech signal, where figure Fig. 4.3.a illustrates the original signal waveform, Fig. 4.3.b the copy synthesis signal. x axis is time and y axis is the signal $s(t)$ value.

In figure Fig. 4.3 there is the model's performance in the time domain. The figure Fig. 4.3.a illustrates the original signal waveform and the figure Fig. 4.3.b the *copy synthesis* results. At the x axis there is time and at the y axis the value of the signal $s(t)$.

Moreover, if we zoom in figure Fig. 4.3, we gain a closer look of the signal waveform for a short time range. The area we focus on is presented in figure Fig. 4.4. Respectively, in figure Fig. 4.4.a and figure Fig. 4.4.b there are the original and the copy synthesis signal waveforms.

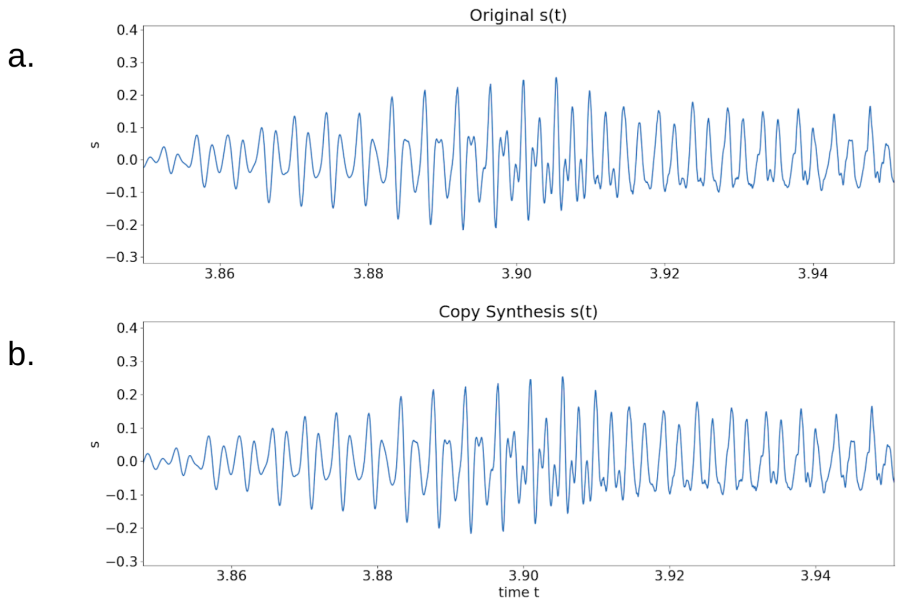


Figure 4.4: A short time range focus of figure Fig. 4.3, figure Fig. 4.3.a presents the original and figure Fig. 4.3.b the copy synthesis signal waveform.

The model's performance in the frequency domain is demonstrated in figure Fig. 4.5. Figure Fig. 4.5.a illustrates the Mel-scaled spectrogram of the original signal and figure Fig. 4.4.b the Mel-scaled spectrogram of the copy synthesized signal. The x axis represents the time evolution is seconds, the y axis the frequency range in Hertz and the colorbar the frequency amplitude of the signal at each point at the frequency-time lattice in Decibel. Additionally, the y-axis is in logarithmic scale.

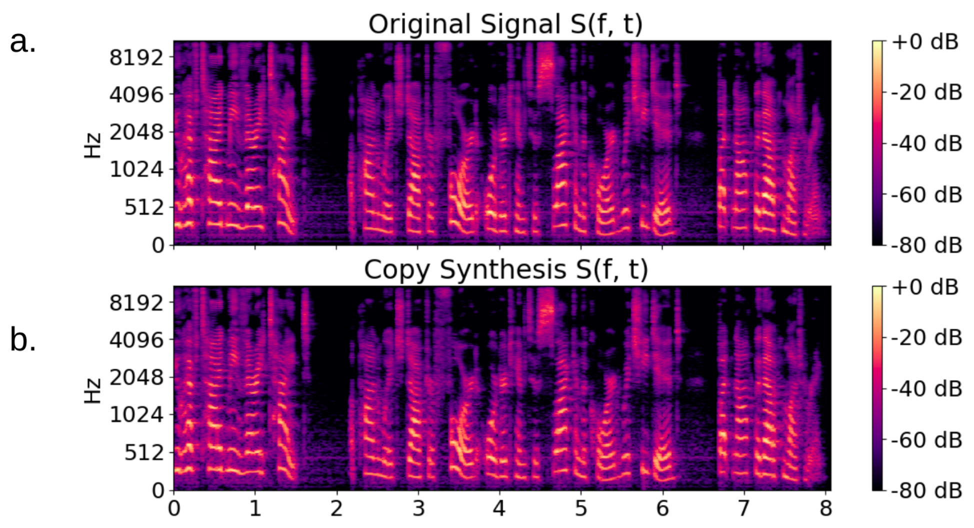


Figure 4.5: Frequency representation - Male Scaled Spectrogram of a speech signal, where x axis is time t in seconds, y axis frequency in Hertz in logarithmic scale and the colorbar illustrates the frequency amplitude at each point at the frequency-time lattice in Decibel. Figure Fig. 4.5.a corresponds to the original signal and figure Fig. 4.5.b to the copy synthesized signal.

5 RNN based Modelling of Speech Signals

5.1 Model Definition

First generation neural vocoders, like WaveRNN, are designed to output the probability distribution of the next sample, by quantizing the waveform of the speech signal. However, more recently there have developed models which reconstruct the speech signal waveform directly and predict the actual waveform value of the next sample, employing suitable loss functions.

In this thesis, our aim is to design and develop a deep neural network model for speech synthesis, faster than the WaveRNN model, examined in the previous chapter. We will make use of recurrent neural networks, as in the WaveRNN. However our purpose is to embrace the benefits of RNNs, while at the same time reducing the total number of the model's parameters. *Long Short Term Memory* recurrent neural networks help us achieve this scope.

5.1.1 Mathematical Background

Below we will have a closer look at the mathematical background of our proposed model.

To begin with, the sum of *amplitude-modulated* AM and *frequency modulated* FM sinusoidal waves can principally simulate speech signals. To that end a speech signal is described by

$$s(t) = \sum_{k=1}^{K} \alpha_k(t) cos\left(\varphi_k(t)\right), \tag{5.1}$$

where $\alpha_k(t)$ is the time-varying amplitude of the k^{th} signal component and $\varphi_k(t)$ is the time-varying phase. The amplitude $\alpha_k(t)$ and phase $\varphi_k(t)$ components are slow varying quantities. Additionally, observing the above formula Eq. 5.1 one can realize that a speech signal is equal to the real part of the complex function $x(t) \in \mathbb{C}$:

$$s(t) = Re\{x(t)\} = Re\left\{\sum_{k=1}^{K} \alpha_k(t)e^{i\varphi_k(t)}\right\} \tag{5.2}$$

The decomposition of the time-varying phase component $\varphi_k(t)$ is presented in the equation Eq. 5.3

$$\varphi_k(t) = 2\pi f_{k,car}(t) + \phi_k(t), \tag{5.3}$$

where $f_{k,car}$ is the carrier frequency of the k^{th} signal component , a constant quantity which corresponds to the frequency of the signal's carrier wave, and $\phi_k(t)$ the respective phase component.

Under the assumption of slowly varying components, we consider a single-component signal and apply the following approximations

$$\alpha(t+1) = \alpha(t) + \Delta\alpha(t) \tag{5.4}$$

$$\varphi(t+1) = \varphi(t) + 2\pi f_{car} + \Delta\varphi(t). \tag{5.5}$$

If we substitute the above approximations into the single component complex valued signal function

$$x(t) = \alpha(t)e^{i\varphi(t)} \tag{5.6}$$

we get

$$\begin{aligned} x(t+1) &= \alpha(t+1)e^{i\varphi(t+1)} \\ &= (\alpha(t) + \Delta\alpha(t))e^{i(\varphi(t)+2\pi f_{car}+\Delta\varphi(t))} \\ &= \left(1 + \frac{\Delta\alpha(t)}{\alpha(t)}\right) e^{i(2\pi f_{car}+\Delta\varphi(t))}x(t) \\ &= c(t)e^{i2\pi f_{car}}x(t), \end{aligned} \tag{5.7}$$

where $c(t) \in \mathbb{C}$ is a complex, time-varying factor equal to

$$c(t) = \left(1 + \frac{\Delta\alpha(t)}{\alpha(t)}\right) e^{i\Delta\varphi(t)}. \tag{5.8}$$

So, finally the temporal advancement of a sinusoidal signal is given by the formula

$$x(t+1) = c(t)e^{i2\pi f_{car}}x(t). \tag{5.9}$$

Observing the result Eq. 5.9, in our proposed model the evolution over time of a sinusoidal signal depends on the previous sample multiplied by a time-varying factor. However, it is noteworthy to mention that the fact that the temporal advancement is performed multiplicatively, is in direct contrast to autoregressive models where is performed additively. The term *autoregressive* implies models where the output signal depends linearly on its own previous values.

Inspired from the above observation, the goal of this study is to build a recurrent neural network capable of predicting the complex factors $c_k(t)$, efficiently and robustly. Specifically, our aim is to build a neural phase vocoder, as illustrated in figure Fig. 5.1. Traditionally, a phase vocoder based on the sinusoidal modeling of speech, with STFT

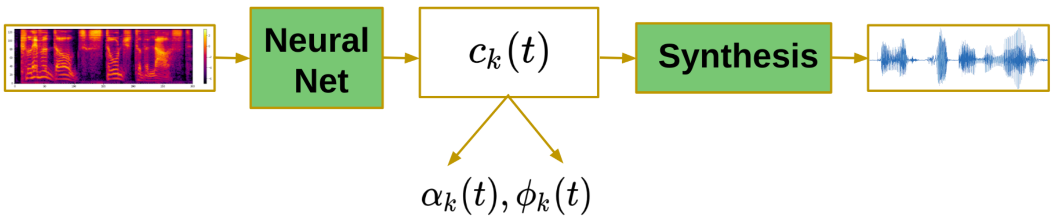


Figure 5.1: Proposed neural phase vocoder.

extracts information for the component phase and amplitude, from the signal frequency representation. Therefore, we will employ neural networks so as to extract each component time-varying amplitude and phase, from frequency domain. Subsequently, we will synthesize the complex signal waveform out of its components.

5.1.2 Model Architecture

The above formulas and calculations driven us to the idea to build a vocoder that exploits the fact that a speech signal can be decomposed into simpler sinusoidal components. To that end, the main goal of this study is to design and train a simpler, smaller and sparser neural network, comparing to WaveRNN, by decoupling the complex speech signal into simpler sinusoidal components .

The architecture and overall computation flow of the proposed model is illustrated in figure Fig. 5.2. The first step is to generate the Mel-scaled spectogram of the signal and extract the information required to be fed as input to the LSTM network. Continuing, the generated output $c_k(t)$ combined with the analytic ground true signal components $x_k(t)$, produce the prediction for the next sample component value $\tilde{x}_k(t+1)$. Finally, the sum of all components construct the prediction for analytic complex signal $\tilde{x}_k(t+1)$. We can measure the accuracy of our prediction, by comparing the prediction with the ground true signal value.

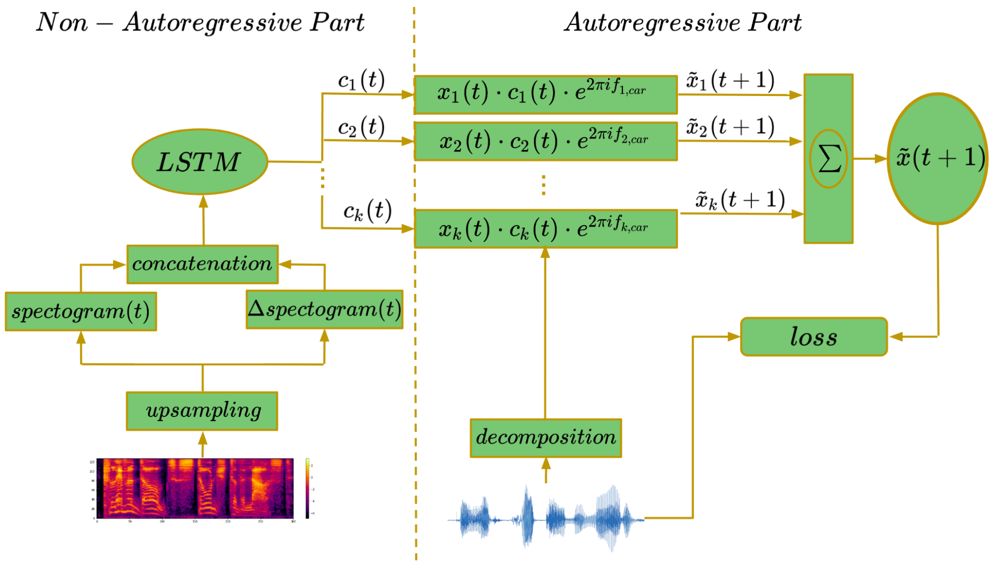


Figure 5.2: Computational Graph of the Proposed Model.

It is noteworthy to mention that the proposed model is separated into a non-autoregressive and an autoregressive part. As we can observe in the above graph, autoregressive is the second part of the model, where the next sample $\tilde{x}_k(t+1)$ is predicted, using the previous sample $x_k(t)$ ground true value. However, the first part of the model, where the factor $c_k(t)$ is generated, is not autoregressive.

5.2 Model Implementation

As discussed in the previous chapter, our target is to train a model qualified to perform waveform reconstruction of the initial speech signal, as well as of the components that comprise it.

To achieve this scope, we mainly used the programming language Python along with a Matlab script, so to preprocess and extract the information required from the data. Also, we used the deep learning framework *Pytorch*, so as to build and train our neural vocoder. Below, we will have a closer look at the overall data processing and the technical details required for the model implementation. We will clarify how to extract the information we need so as to define the input, the output and the loss function of the neural model.

5.2.1 Synthetic Speech Signals

The first step on building our neural vocoder was to produce synthetic speech signals. Particularly, we produced mono-component, as well as multi-component AM and FM complex valued signals $x(t)$

$$x(t) = \sum_{k=1}^{K} x_k(t). \tag{5.10}$$

As shown in equation Eq. 5.10, the sum over all the K components $x_k(t)$ constitute the complex valued signal $x(t)$. Each component has a different time-varying amplitude $\alpha_k(t)$ and phase $\varphi_k(t)$

$$x_k(t) = \alpha_k(t)e^{i\varphi_k(t)}. \tag{5.11}$$

While producing synthetic speech signals, *linguistic limitations* should be taken into consideration. Particularly, the maximum values of each component frequency $f_k(t)$ have to satisfy the following relation

$$f_k(t)_{max} < \frac{f_s}{2}, \tag{5.12}$$

where f_s is the sampling rate, a constant value which defines how many samples are generated per unit time. Additionally, time-varying signal component frequency $f_k(t)$ is directly associated with the phase $\varphi_k(t)$ through its temporal derivative

$$f_k(t) = \frac{1}{2\pi}\frac{d\varphi_k(t)}{dt}. \tag{5.13}$$

5.2.2 Real Audio Speech Signals

Moreover, we used real audio clips collected from a public domain speech dataset, *The LJ Speech Dataset* [18]. In this case, we extracted the signal real valued waveform $s(t)$ and the sampling rate f_s value using a Matlab script. However, so as to define the neural model input and loss function, we need to extract the components $s_k(t)$ of those speech signals as well as the analytic signal $x_k(t)$ of each component.

5.2.2.1 Filter Banks

So as to decompose the speech signals generated from the audio clips into separate components, we used *Filter Banks*.

Using the Matlab function *Uniform Filter Bank* [19], we designed a uniform filter bank with a given number of K filters. The impulse response of the filter bank on the waveform of the signal $s(t)$ returns the signal decomposition into a number of K components. Each component is covering a specific range of the signal frequencies and those ranges are equally separated, since the filter bank is uniform. The carrier frequency of each signal component is equal to the intermediate value of the frequency range it covers.

To that end, the output of each filter is assumed to be a mono-component AM and FM signal, while the sum over all the components construct the initial multi-component signal.

5.2.2.2 Hilbert Transformation

The analytic signal component $x_k(t)$ is generated by the *Hilbert Transformation* of the real valued signal $s(t)$

$$x_k(t) = F^{-1}\Big(F(s_k(t))2U\Big), \tag{5.14}$$

where F is the Fourier transform, U the unit step function [1].

5.2.3 Spectogram Linear Interpolation

In order to build the input dataset, we need to produce one frequency band $S(t)$ corresponding to each timestep of the signal $s(t)$.

So, firstly, we extract the signal's melspectrogram, by applying STFT and rescaling the frequency axis to the Mel-scale. As we have mentioned in previous chapters, the melspectrogram $S(f,t)$ illustrates the amplitude of each frequency value f at each timestep t, at a discrete time interval. While implementing STFT, the number of Mel frequency bands that are generated is smaller than the overall signal timesteps. As a result, so as to assure one corresponding frequency band for each signal timestep, we *upsample* the melspectrogram by applying a *linear interpolation algorithm.*

Particularly, assume that we want to interpolate a discrete function $f(\tau)$, with sampling step $d\tau$. A linear approximation of the value the function has at a specific time t can be estimated using the following formula

$$f(t) = 1 - \frac{t - \lfloor \frac{t}{d\tau} \rfloor d\tau}{d\tau} f(\left\lfloor \frac{t}{d\tau} \right\rfloor) + \frac{t - \lceil \frac{t}{d\tau} \rceil d\tau}{d\tau} f(\left\lceil \frac{t}{d\tau} \right\rceil), \tag{5.15}$$

where $\lfloor \frac{t}{d\tau} \rfloor$ indicates the floor of the scalar $\frac{t}{d\tau}$, particularly the largest integer i, such that $i \leq \frac{t}{d\tau}$. Subsequently, $\lceil \frac{t}{d\tau} \rceil$ indicates the ceil of the scalar $\frac{t}{d\tau}$, particularly the smallest integer i, such that $i \geq \frac{t}{d\tau}$.

5.2.4 Model input

Using the convex interpolation algorithm we generate a frequency representation corresponding to each timestep of the initial speech signal. So finally the input data that we feed our network are the frequency bands of each specific timestep t concatenated with their differences from the frequency bands of the previous timestep $t-1$

$$\Delta S(t) = S(t) - S(t-1). \tag{5.16}$$

When implementing the STFT of the initial speech signal, we choose the number of the Mel bands we want to generate, and subsequently the size of the input frequency bands. At each time point, since the input dataset $I(t)$ is produced from the concatenation described above, its size will be equal to twice the number of Mel bands we have chosen

$$I_{2*mels}(t) = [S_{mels}(t), \Delta S_{mels}(t)]. \tag{5.17}$$

5.2.5 Model Output

As described above, at each time point we feed the respective input $I(t)$ into the LSTM model and the model outputs the complex valued multiplicative factor $c_k(t)$. Technically, in Pytorch framework, where we define and train our LSTM model, we work with Tensors and complex valued Tensors are not supported. Thus, we define our LSTM model output to be the real and the imaginary part of each $c_k(t)$, separately. To that end, each output $O(t)$ has size twice the number K of the signal's components

$$O_{2*K}(t) = [Re\{c_1(t)\}, Im\{c_1(t)\}, ..., Re\{c_K(t)\}, Im\{c_K(t)\}]. \tag{5.18}$$

5.2.6 Loss Function

So as to train our neural model we feed it examples of input and output pairs. The model is taught the correct answer and learns how to generate the desired output by making mistakes. In our case, at each time point the deviation between the prediction $\tilde{x}(t+1)$ and the ground true value $x(t)$ is calculated through their *Mean Squared Error* (MSE)

$$MSE(t) = \overline{|\tilde{x}(t+1) - x(t+1)|^2}. \tag{5.19}$$

In contrast to the WaveRNN variant examined in the previous chapter, our model does not learn to solve a classification problem. We aim to directly reconstruct the signal waveform and not the probability distribution of the next sample value. To that we use a different and appropriate to our implementation loss function, the *Mean squared Error*.

More technical details of how the MSE is computed using the deep learning framework Pytorch can be found in Appendix A.

6 Results & Discussion

6.1 Modelling of Sinusoidal Signals

The results of our proposed model implementation on synthetic as well as real speech signals are presented in this paragraph. As clarified at the Chapter 5.1.2, the second part of our proposed model is *autoregressive*, in other words the model uses the previous sample value so as to generate the next. In this study, during the training as well as the inference procedure, for the prediction of next sample value $\tilde{x}(t+1)$ we use the previous sample value of the original speech signal $x(t)$. In the future steps of our study we plan to use the previous sample value that the model predicts $\tilde{x}(t)$, instead of the value generated from the original speech signal $x(t)$, so as to build a stand-alone neural vocoder faster than the WaveRNN examined in Chapter 4.

6.1.1 Modelling of Single Component Sinusoidal Signals

The first step was to test the model on synthetic mono-component sinusoidal signals. Particularly we generated an AM and FM sinusoidal signal in discrete time domain, with $n = \{1, 2, ..., 50000\}$ timesteps. We created a signal with 10 periods T, with sampling rate f_s equal to $5000 samples/sec$ and with carrier frequency f_{car} equal to 1000. Also, the LSTM model we trained had 20 hidden states, a number of 100 epochs and learning rate equal to 0.01.

We produced the analytic signal using the following formula

$$x(t) = a(t)e^{i\varphi(t)}. \tag{6.1}$$

The time-varying amplitude $a(t)$ and phase $\varphi(t)$ are

$$\begin{aligned} a(t) &= 1 + acos(2\pi t + \phi_a) \\ \varphi(t) &= 2\pi f_{car} t + a_\varphi cos(2\pi t + \phi_\varphi), \end{aligned} \tag{6.2}$$

where the constant amplitudes a, a_φ have values equal to 0.7 and 75.5, respectively. Also, the constant phases ϕ_a, ϕ_φ have values equal to 0 and 0.4, respectively. Each discrete time step t_i is equal to n_i/f_s, with $n = \{1, 2, ..., 50000\}$.

Continuing, we extracted the necessary information from the produced signal and followed the computational flow described in the Chapter 5.2. Figure Fig. 6.1 presents the model's performance in the time domain. The blue line illustrates the original signal waveform and the orange line the signal waveform generated from the proposed model, or else the *copy synthesis* results.

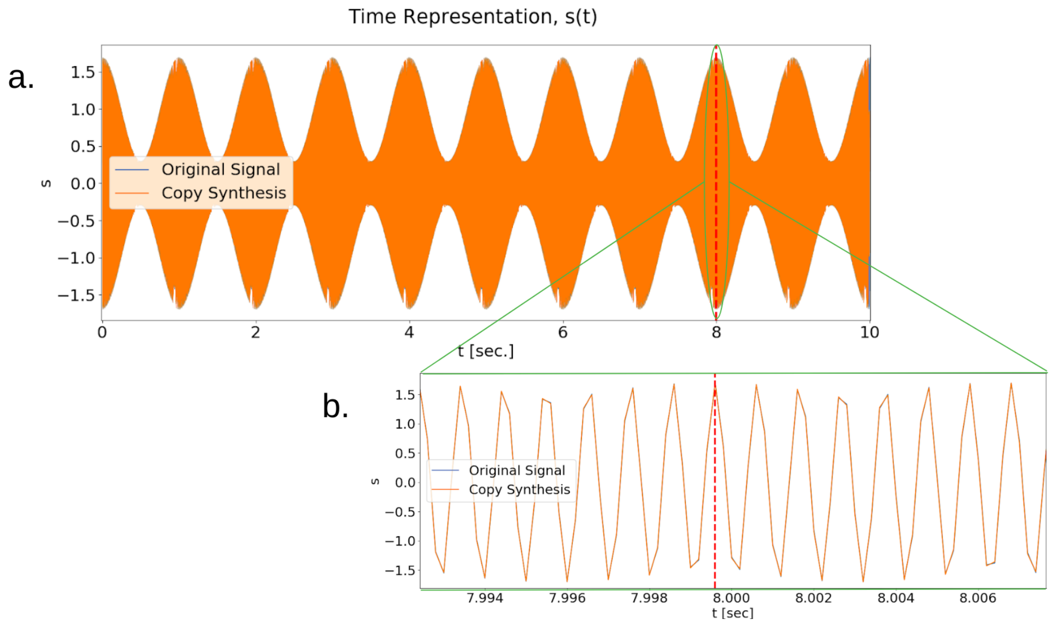


Figure 6.1: Time representation of a mono-component AM and FM signal, where the blue line illustrates the original signal waveform, the orange line the copy synthesis signal, x axis is time t and y axis is the signal $s(t)$ value. Figure Fig. 6.1.b focuses on a short time range of figure Fig. 6.1.a.

At the x axis there is time and at the y axis the value of the signal $s(t) = Re\{x(t)\}$. The red dashed line separates the timeseries into two parts, the left part is used during training and the left part during inference. We train our model using examples of input and output pairs extracted only from the left part of the waveform. The right part of the waveform is used to test our model's performance on new different examples, that has not processed before. Moreover, if we zoom in Fig. 6.1.a, we gain a closer look of the signal waveform for a short time range. The green oval circle indicates the area we focus and figure Fig. 6.1.b presents the signals waveform of this area.

The model's performance in the frequency domain is demonstrated in figure Fig. 6.2. Figure Fig. 6.2.a illustrates the Mel-scaled spectrogram of the original signal and Figure Fig. 6.2.b the Mel-scaled spectrogram of the copy synthesized signal. The x axis represents the time evolution is seconds, the y axis the frequency range in Hertz and the colorbar the frequency amplitude of the signal at each point at the frequency-time lattice in Decibel. The red dashed line separates the spectrograms into two parts, the left part is used during the training and the left part during the inference of the LSTM model.

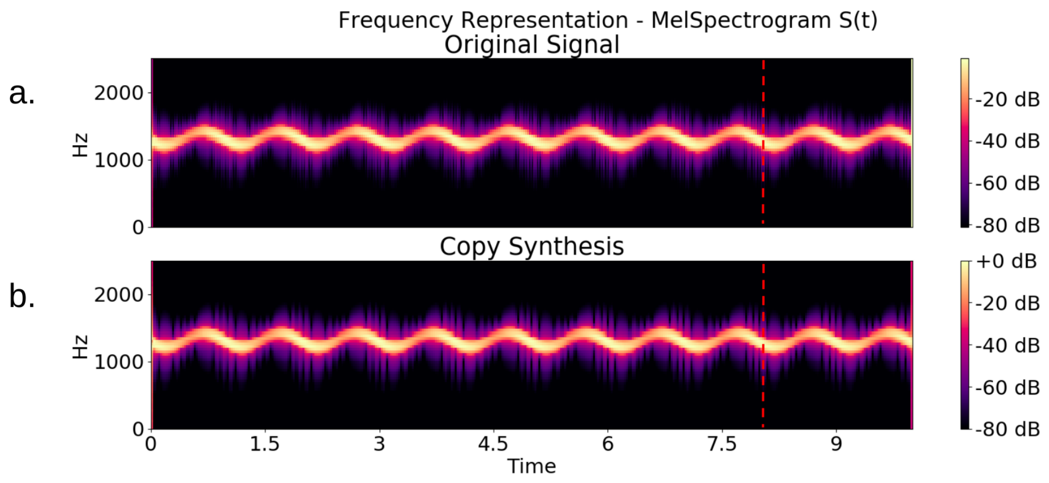


Figure 6.2: Frequency representation - Male Scaled Spectrogram of a mono-component AM and FM signal, where x axis is time t in seconds, y axis is frequency in Hertz and the colorbar illustrates the frequency amplitude at each point at the frequency-time lattice in Decibel. Figure Fig. 6.2.a corresponds to the original signal and figure Fig. 6.2.b to the copy synthesized signal.

In the above figures, it is clear that the model's prediction for the next sample come to a great agreement with the respective original signal value. So as to quantitatively measure our model's performance we will calculate the *Relative Mean Squared Error* (RelMSE) between the original $x(t)$ and the copy synthesized signal $\tilde{x}(t)$, using the following formula

$$RelMSE = \frac{\overline{|x(t) - \tilde{x}(t)|^2}}{\overline{|x(t) - \overline{x}|^2}}, \tag{6.3}$$

where $\overline{x}$ is the original signal mean value.

Taking into consideration that $x(t), \tilde{x}(t) \in \mathbb{C}$ (see Appendix A), the RelMSE of the model implementation on a mono-component, AM and FM signal for the training and the inference data set is shown in Table 6.1.

Table 6.1: Mono-Component Sinusoidal Signal

SET	RelMSE
Training	0.0038
Inference	0.0077

We observe that in the inference dataset the RelMSE is slightly greater than in the training dataset. This is a reasonable result, since the model has been trained on predicting samples from the training dataset. In conjunction, the inference data set is more challenging, since it contains samples different from those that the model has been trained on. Also, the fact that the RelMSE for both sets has the same magnitude of value is a promising indication that our model is *generalized* and is not *overfit*. Particularly, generalization is the model's ability to give sensible outputs to sets of input that has never seen before. Also, with the term overfit model we indicate a model that learns to output only to the training sets of input and fails to output to new sets of input that is unfamiliar to.

6.1.2 Modelling of Multi-Component Sinusoidal Signals

The second step of our study, was to repeat the above procedure for a synthetic multi-component sinusoidal signal. Similarly, to the previous section, we generated three different AM and FM sinusoidal signals in discrete time domain, with $n = \{1, 2, ..., 50000\}$ timesteps. Each one of the three signals is created has 10 periods T, sampling rate f_s equal to $5000 samples/sec$ and carrier frequency f_{car} , equal to $400, 1000, 1700$, respectively. The LSTM model characteristics are 20 hidden states, 100 epochs and learning rate equal to 0.01, as in the implementation described in the previous section.

In this case, we produced the analytic signal using the following formula

$$x(t) = \sum_{k=1}^{3} a_k(t) e^{i\varphi_k(t)}. \tag{6.4}$$

The time-varying amplitude $a_k(t)$ and phase $\varphi_k(t)$ are

$$\begin{aligned} a_k(t) &= 1 + a_k cos(2\pi t + \phi_{a,k}) \\ \varphi_k(t) &= 2\pi f_{car,k} t + a_{\varphi,k} cos(2\pi t + \phi_{\varphi,k}), \end{aligned} \tag{6.5}$$

where $k = \{1, 2, 3\}$,for each component the amplitudes values are $a_k = \{0.7, 0.6, 0.5\}$ and $a_{\varphi,k} = \{142.5, 66.3, 51.2\}$ and the component phases values are $\phi_{a,k} = \{2.2, 2.3, 2.9\}$, $\phi_{\varphi,k} = \{0.6, 2.9, 0.3\}$. Each discrete time step t_i is equal to n_i/f_s, with $n = \{1, 2, ..., 50000\}$.

Below, we will examine our model's performance in synthesizing multi-component AM and FM signals. We will observe the results in the time as well as in the frequency domain, equivalently to section 6.1.1.

In Figure Fig. 6.3 there is a time representation. The blue line illustrates the original signal waveform and the orange line its copy synthesis. At the x axis there is time and at the y axis the value of the signal $s(t) = Re\{x(t)\}$. The red dashed line

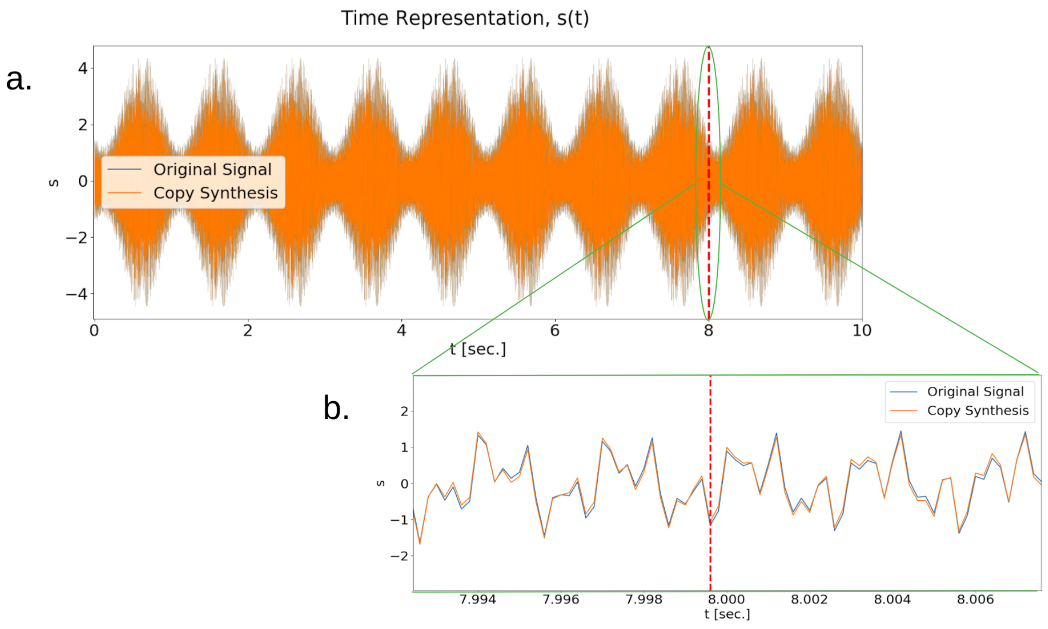


Figure 6.3: Time representation of a three-component AM and FM signal, where the blue line illustrates the original signal waveform, the orange line the copy synthesis signal, x axis is time and y axis is the signal $s(t)$ value. Figure Fig. 6.3.b focuses on a short time range of figure Fig. 6.3.a.

separates the timeseries into two parts, the left part is used during training and the left part during inference. We train our model using examples of input and output pairs extracted only from the left part of the waveform. The right part of the waveform is used to test our model's performance on new different examples, that has not processed before. Moreover, if we zoom in Fig. 6.3.a, we gain a closer look of the signal waveform for a shorter time range. The green oval circle indicates the area we focus and figure Fig. 6.3.b presents the signals waveform of this area.

The model's performance in the frequency domain is demonstrated in figure Fig. 6.4. Figure Fig. 6.4.a illustrates the Mel-scaled spectrogram of the original signal and Figure Fig. 6.2.b the Mel-scaled spectrogram of the copy synthesized signal. The x axis represents the time evolution is seconds, the y axis the frequency range in Hertz and the colorbar the frequency amplitude of the signal at each point at the frequency-time lattice in Decibel. The red dashed line separates the spectrograms into two parts, the left part is used during the training and the left part during the inference of the LSTM model.

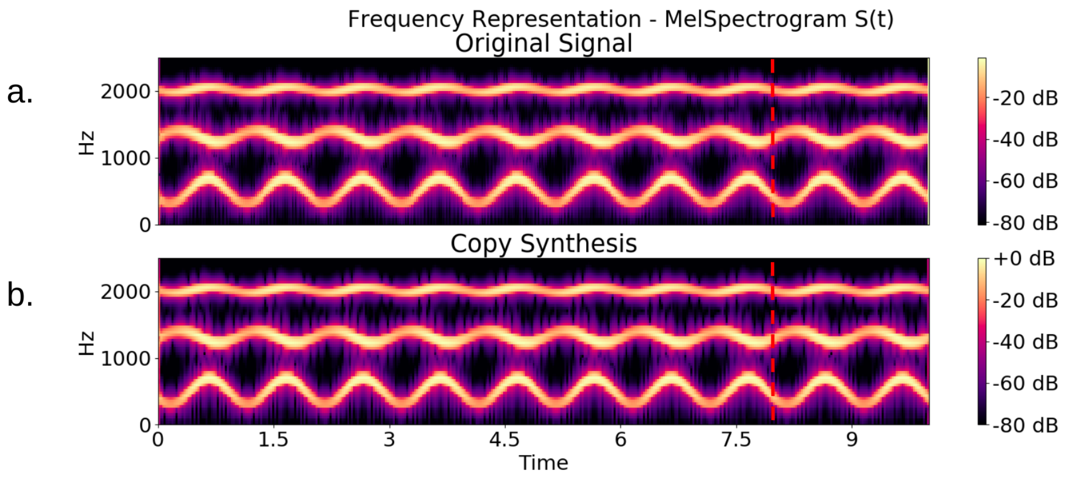


Figure 6.4: Frequency representation - Male Scaled Spectrogram of a three-component AM and FM signal, where x axis is time t in seconds, y axis is frequency in Hertz and the colorbar illustrates the frequency amplitude at each point at the frequency-time lattice in Decibel. Figure Fig. 6.4.a corresponds to the original signal and figure Fig. 6.4.b to the copy synthesized signal.

We can clearly observe that the model's prediction for the next sample comes to a great agreement with the respective original signal value, in the case of a multi-component sinusoidal signal too. Using the equation Eq. 6.3 we can calculate the *Relative Mean Squared Error* (RelMSE) between the original $x(t)$, given by the three-component signal equation Eq. 6.4 and the respective copy synthesized signal $\tilde{x}(t)$. Taking into consideration that $x(t), \tilde{x}(t) \in \mathbb{C}$ (see Appendix A), the RelMSE of the model implementation on a three-component, AM and FM signal of the training and the inference procedure is shown in Table 6.3.

Table 6.2: Three-Component Sinusoidal Signal

SET	RelMSE
Training	0.0041
Inference	0.0042

Also in this case, the values of the RelMSE are similar to the mono-component implementation of our model. We reasonable observe that in the inference dataset the RelMSE is slightly greater than in the training dataset. Also, the RelMSE for both sets has the same magnitude of value, in this case too. This is a promising indication that our model is *generalized* and is not *overfit*.

6.2 Modelling of Speech Signals

In the previous sections, we have examined the proposed model's performance on synthetic speech signals. Speech in principle is a sum of multiple AM and FM signals. To that end we firstly we tested the case of a mono-component AM and FM signal and secondly a multi-component signal, aiming to simulate real speech signals. In this section we test real audio clips collected from a public domain speech dataset, *The LJ Speech Dataset* [18].

6.2.1 Filter Bank Analysis Results

In order to test the proposed model on real audio clips, firstly we have to decompose the speech signals generated from the audio clips into separate components. By following the procedure analytically explained in Chapter 5.2.2, employing the fact that speech is a multi-component signal with the use of *Filter Banks* we separate the signal into 20 components. Each one of those 20 components is considered to be a mono-component AM and FM signal and the sum over all the mono-component signals construct the initial speech signal.

Particularly, we used one female audio clip , with sampling rate f_s equal to 16000 $samples/sec$ and duration equal to $9seconds$. We decomposed the signal using a Uniform Filter Bank of 20 filters. The Filter Bank analysis results in time domain are presented in figure Fig. 6.5. Figure Fig. 6.5.a illustrates the speech signal waveform, while the figure Fig. 6.5.b presents each one of the 20 AM/FM mono-component signals that build the initial signal.

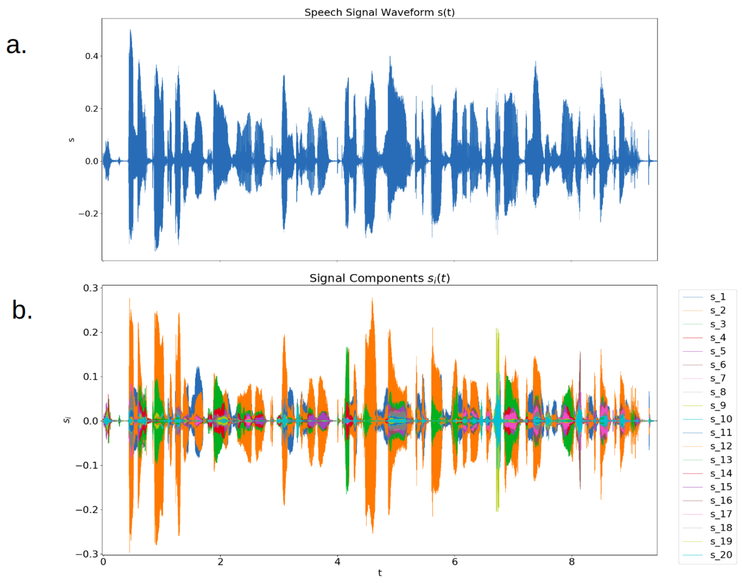


Figure 6.5: Filter Bank analysis results in time domain. Figure Fig. 6.5.a illustrates the speech signal waveform and figure Fig. 6.5.b the 20 AM/FM mono-component signals that build the initial signal.

While applying a Uniform Filter Bank of 20 sub-bands on the complex signal, the complex signal frequency range is equally divided into 20 sections. Each signal component covers a different frequency section of the signal. Additionally, we consider that each component, or else each AM/FM signal, has carrier frequency equal to the intermediate value of the frequency range it covers.

In figure Fig. 6.6 there is the spectrogram of the initial complex signal, with y-axis in linear scale. The Filter Bank analysis results in frequency domain are presented in figure Fig. 6.6, too. Particularly, each sub-figure illustrates the frequency range each signal component covers, beginning from the lowest frequency range $S_1(f,t)$ till the highest $S_{20}(f,t)$.

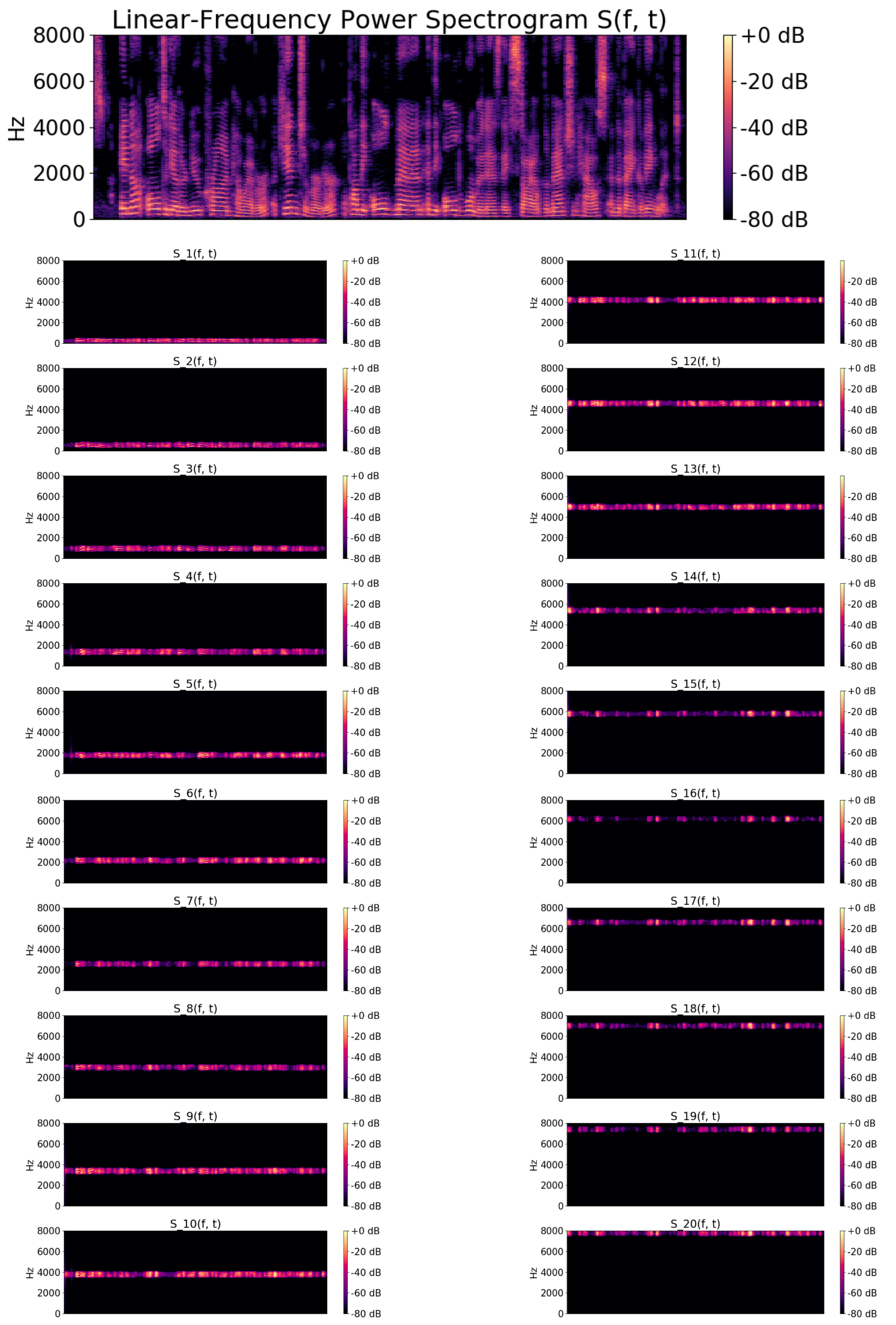


Figure 6.6: Filter Bank analysis results in frequency domain, each sub-figure illustrates the melspectrogram of each one of the 20 components, beginning from the lowest frequency range $S_1(f,t)$ till the highest $S_{20}(f,t)$.

6.2.2 Modelling of Speech Signals

In this paragraph there are presented the results generated from real audio clips. Particularly, we used one female audio clip, with sampling rate f_s equal to $16000 samples/second$ and duration equal to 9 seconds. We separated the signal into 20 components, each one characterized by a different carrier frequency, as described in the previous section. Also, the LSTM model we trained had 20 hidden states, a number of 1000 epochs and learning rate equal to 0.0001. Additionally, before feeding the training data set into the LSTM model, we separated it into batches, with batch size equal to 64.

Similarly to the previous sections, in figure Fig. 6.7 there is the model's performance in the time domain. The blue line illustrates the original signal waveform and the orange line the *copy synthesis* results. At the x axis there is time and at the y axis the value of the signal $s(t) = Re\{x(t)\}$. Moreover, if we zoom in Fig. 6.7.a, we gain a closer look of the signal waveform for a short time range. The green oval circle indicates the area we focus on and figure Fig. 6.7.b presents the signal waveform in this area.

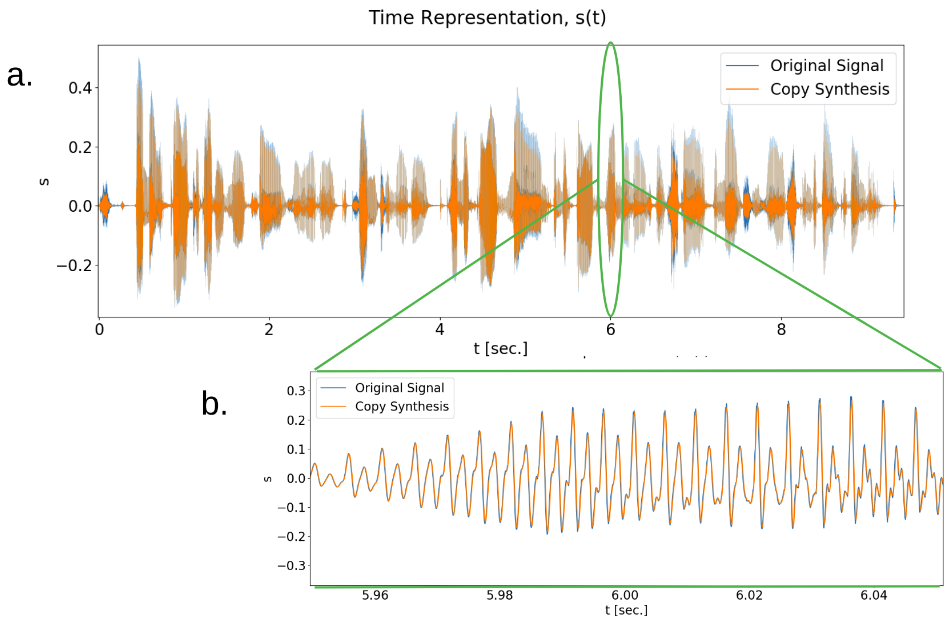


Figure 6.7: Time representation of a speech signal, where the blue line illustrates the original signal waveform, the orange line the copy synthesis signal, x axis is time and y axis is the signal $s(t)$ value. Figure Fig. 6.7.b focuses on a short time range of figure Fig. 6.7.a.

Figure Fig. 6.7 presents a timeserie that is not used during the training procedure. It corresponds to an audio clip new for the neural model, but generated from the same female voice used for training. Specifically, the training as well as the inference of the proposed model is performed on single-speaker audio clips. Multiple speaker speech synthesis is a different category, known as *universal synthesis*.

The model's performance in the frequency domain is demonstrated in figure Fig. 6.8. Figure Fig. 6.8.a illustrates the Mel-scaled spectrogram of the original signal and figure Fig. 6.2.b the Mel-scaled spectrogram of the copy synthesized signal. The x axis represents the time evolution is seconds, the y axis the frequency range in Hertz and the colorbar the frequency amplitude of the signal at each point at the frequency-time lattice in Decibel. Additionally, the y-axis is in logarithmic scale.

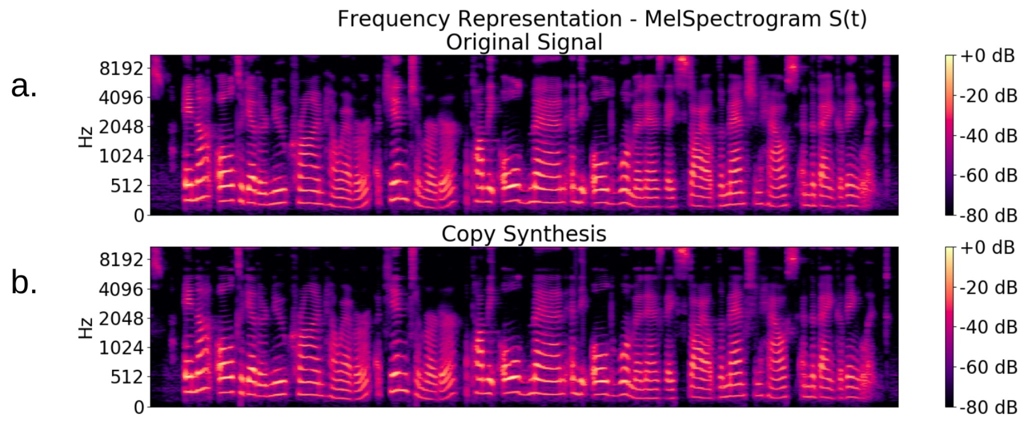


Figure 6.8: Frequency representation - Male Scaled Spectrogram of a speech signal, where x axis is time t in seconds, y axis frequency in Hertz in logarithmic scale and the colorbar illustrates the frequency amplitude at each point at the frequency-time lattice in Decibel. Figure Fig. 6.8.a corresponds to the original signal and figure Fig. 6.8.b to the copy synthesized signal.

In the above figures, it is clear that the model's prediction for the next sample come to a great agreement with the respective original signal value. Using the equation

Eq. 6.3 we can calculate the *Relative Mean Squared Error* (RelMSE) between the original $x(t)$ generated from the audio clip and the respective copy synthesized signal $\tilde{x}(t)$. Taking into consideration that $x(t), \tilde{x}(t) \in \mathbb{C}$ (see Appendix A), the RelMSE of the model implementation on a speech signal for the inference data set is shown in Table 6.3

Table 6.3: Speech Signal

SET	RelMSE
Inference	0.0890

From the above results we observe that the proposed model perform satisfyingly in the case of real speech signals, too. However so as to able to compare our models accuracy with state-of-the-art models, like the WaveRNN model examined in Chapter 4, there are some additional future steps that we have to implement further.

7 Conclusions and Future Steps

In this thesis, we target to develop a light neural vocoder, faster than the state-of-the-art WaveRNN model [5]. For this scope we combine deep learning(RNNs) with prior knowledge of the inherent sinusoidal nature of speech. Particularly, we build a neural vocoder taking into consideration the sinusoidal nature of speech signals. Moreover, we use Recurrent Neural Networks (RNNs) and particularly Long Short Term Recurrent neural networks.

We demonstrate the state-of-the-art performance of a WaveRNN variant model and present the generated results both in time and frequency domain. Firstly, we examine the proposed model performance using mono-component as well as multi component sinusoidal signals. We demonstrate the results both in time and frequency domain, also we calculate the Relative Mean Speared Error between the target and the generated sample. Observing the results we conclude that the model's prediction for the next sample comes to a great agreement with the respective original signal value.

Furthermore, we use real speech signals generated from audio clips. After the speech signal decomposition, we test the proposed model under the assumption that each signal component is a mono-component sinusoidal signal. Also, we use both time and frequency representation of the results and we calculate the Relative Mean Speared Error between the target and the generated sample. From the results we observe that the proposed model perform satisfyingly in the case of real speech signals, too.

However so as to able to compare our models accuracy with state-of-the-art models we have to implement further some additional steps. In this study, the prediction of next sample value is executed using the previous sample value of the original signal, during the training as well as the inference procedure. In the future steps of our study we plan to use the previous sample value that the model predicts, instead of the original previous sample value, so as to build a stand-alone neural vocoder.

Moreover, there has to be more research on the neural model architecture. An alternative of the LSTM unit could be a GRU unit. Also, we could use different loss function, too. Currently, we are using the MSE of the real speech signal values, in time domain. However, we could test the performance of a different loss function too, taking into consideration the nature our problem. Particularly, the fact that speech has both real and imaginary part and that speech is represented in frequency domain too.

I would be glad to discuss any comments or feedback (apapadaki@physics.uoc.gr).

A
Pytorch, Tensors & Complex Numbers

As mentioned in Chapter 5, our neural vocoder is built and trained in Pytorch framework. Pytorch framework uses Tensors so as to perform the training. However, complex valued Tensors are not supported. So, below we will describe in detail how we implement technically the training procedure.

As described in Chapter 5, at each time step we feed our model examples of input and output pairs. For each $I(t)$ there is a respective $O(t)$. Final aim of the neural vocoder is to predict the next complex valued audio sample $\tilde{x}(t+1)$, which is equal to the summation Eq. A.1 over all signal component following

$$\tilde{x}(t+1) = \sum_{k=1}^{K} c_k(t) x_k(t) e^{i2\pi f_{k,car}}, \tag{A.1}$$

where $c_k(t), x_k(t) \in \mathbb{C}$. The $f_{k,car}$ corresponds to the carrier frequency of each component.

According to Eq. A.1, we can build the next sample component $x_k(t+1)$ by multiplying the LSTM model output $c_k(t)$ with the previous sample component $x_k(t)$ and with the exponential $e^{i2\pi f_{k,car}}$. However, since we can not manipulate complex values, we calculate separately the real and the imaginary part of the next sample $\tilde{x}(t+1)$. Particularly, consider three complex numbers $z_1, z_2, z_3 \in \mathbb{C}$, Eq. A.2

$$\begin{aligned} z_1 &= a + ib \\ z_2 &= c + id \\ z_3 &= e + if, \end{aligned} \tag{A.2}$$

where $a, b, c, d, e, f \in \mathbb{R}$. The product of those three complex numbers is given by the following formula Eq. A.3

$$\begin{aligned} z_1 \cdot z_2 \cdot z_3 = & [(a \cdot c - b \cdot d) \cdot e - (a \cdot d + b \cdot c) \cdot f] \\ & + i[(a \cdot c - b \cdot d) \cdot f + (a \cdot d + b \cdot c) \cdot e]. \end{aligned} \tag{A.3}$$

Thus, since equation Eq. A.1 is the product of three complex values, the real and the imaginary parts of the next sample prediction are calculated according to the following equations

$$\begin{aligned} Re\{\tilde{x}(t+1)\} = & \sum_{k=1}^{K} \Big(Re\{c_k(t)\}Re\{x_k(t)\} - Im\{c_k(t)\}Im\{x_k(t)\} \Big) cos(2\pi f_{k,car}) \\ & - \Big(Re\{c_k(t)\}Re\{x_k(t)\} + Im\{c_k(t)\}Im\{x_k(t)\} \Big) sin(2\pi f_{k,car}) \\ Im\{\tilde{x}(t+1)\} = & \sum_{k=1}^{K} \Big(Re\{c_k(t)\}Re\{x_k(t)\} - Im\{c_k(t)\}Im\{x_k(t)\} \Big) sin(2\pi f_{k,car}) \\ & + \Big(Re\{c_k(t)\}Re\{x_k(t)\} + Im\{c_k(t)\}Im\{x_k(t)\} \Big) cos(2\pi f_{k,car}). \end{aligned} \tag{A.4}$$

The loss function of our model is the MSE between the model prediction of the next sample value $\tilde{x}(t+1)$ and the ground true sample value $x(t)$, as shown in equation Eq. 5.19. However, we need a formula that calculates the MSE using the real and the imaginary part of $\tilde{x}(t+1)$ and $x(t)$. Below we will mathematically describe the procedure of extracting such a formula.

To begin with, if we substitute the real and the imaginary parts of $\tilde{x}(t+1)$ and $x(t)$, the equation Eq. 5.19 takes the following form

$$\begin{aligned} \overline{|\tilde{x}(t+1) - x(t)|^2} = & \overline{\left| \Big(Re\{\tilde{x}(t+1)\} - Re\{x(t)\} \Big) + i \Big(Im\{\tilde{x}(t+1)\} - Im\{x(t)\} \Big) \right|^2} \\ = & \overline{\Big(Re\{\tilde{x}(t+1)\} - Re\{x(t)\} \Big)^2 + \Big(Im\{\tilde{x}(t+1)\} - Im\{x(t)\} \Big)^2}, \end{aligned} \tag{A.5}$$

since for $z \in \mathbb{C}$ with $z = a + ib$, the square root of the absolute value is equal to $|z|^2 = a^2 + b^2$.

Continuing, consider two set of numbers A, B, where $A = a_1, a_2, ..., a_N$ and $B = b_1, b_2, ..., b_N$. The mean value of their sum is equal to the sum of the mean value of each set separately, as proven below

$$\begin{aligned}\overline{A+B} &= \frac{1}{N}\sum_{i=1}^{N}(a_i + b_i) \\ &= \frac{1}{N}\sum_{i=1}^{N} a_i + \frac{1}{N}\sum_{i=1}^{N} b_i \\ &= \overline{A} + \overline{B}.\end{aligned} \tag{A.6}$$

Observing the above result, we can extract a formula for the MSE which contains the real and the imaginary parts of $\tilde{x}(t+1)$ and $x(t)$ and can be practically used in the programming implementation of our model. To that end, applying the result Eq. A.6 on the equation Eq. A.5 gives the following formula

$$MSE(t) = \overline{|Re\{\tilde{x}(t+1)\} - Re\{x(t)\}|^2} + \overline{|Im\{\tilde{x}(t+1)\} - Im\{x(t)\}|^2}. \tag{A.7}$$

B Python Code

B.1 Training

```python

import librosa
import librosa.display
import time
import matplotlib
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import torch
import torch.nn as nn
from torch.autograd import Variable
from sklearn.preprocessing import MinMaxScaler
import scipy



# Reproducibility
np.random.seed(876543210)
torch.manual_seed(876543210)





# Define Model
class LSTM(nn.Module):     # nn.Module : convinient way of
                           # encapsulating parameters

```

```python
    def __init__(self, num_classes, input_size, hidden_size, num_layers):
        super(LSTM, self).__init__()

        self.num_classes = num_classes    # output_size
        self.num_layers = num_layers
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.seq_length = seq_length

        # Define the LSTM layer
        self.lstm = nn.LSTM(input_size = input_size, hidden_size = hidden_size,
                             num_layers = num_layers, batch_first = True)
            # when batch_first = True
            # it will use the second dimension as the sequence dimension :
            # (batch_size, seq_len, n_features)
        # Define the output layer
        self.fc = nn.Linear(hidden_size , num_classes)
            # Applies a linear transformation to the incoming data:
            # y = x W^T + b



    def forward(self, x):
        # initialise our hidden and cell state
        # Don't do this if you want your LSTM to be stateful
        h_0 = Variable(torch.zeros(
            self.num_layers, x.size(0), self.hidden_size))
#         h_0 = Variable(torch.normal( mean = 0, std = 1.5, size = (self.num_layers, x.size(0), self.hidden_size) ))
#         print("\n x.size(0) = ", x.size(0))
#         print("\n h_0.shape = ", h_0.shape)

        c_0 = Variable(torch.zeros(
            self.num_layers, x.size(0), self.hidden_size))
#         c_0 = Variable(torch.normal( mean = 0, std = 1, size = (self.num_layers, x.size(0), self.hidden_size) ))
#         print("\n c_0.shape = ", c_0.shape)

        # Propagate input through LSTM
        ula, (h_out, _) = self.lstm(x, (h_0, c_0))    # returns : lstm_out, ( hidden_state, cell_state )
```

```python
#         print("\n ula.shape = ", ula.shape)
#         print("\n A h_out.shape = ", h_out.shape)

        h_out = h_out.view(-1, self.hidden_size)    #
#         print("\n B h_out.shape = ", h_out.shape)
        out = self.fc(h_out.squeeze(0))    # works like Dence
#         print("\n out.shape = ",out.shape)
        return out



def validate(epoch, testX, testY, car_fs):
    with torch.no_grad():
        output = lstm(testX)



        # Obtain the loss function
        test_prediction = Variable(torch.zeros( (testY.shape[0], 2)) )    # [real, imag]
        test_prediction[:, 0] = (testY[:, 0, 0] * output[:, 0] - testY[:, 0, 0 + 1] * output[:, 0 + 1]) * np.cos(2 * np.pi * car_fs[0]) \
                                - (testY[:, 0, 0] * output[:, 0 + 1] + testY[:, 0, 0 + 1] * output[:, 0]) * np.sin(2 * np.pi * car_fs[0])
        test_prediction[:, 1] = (testY[:, 0, 0] * output[:, 0] - testY[:, 0, 0 + 1] * output[:, 0 + 1]) * np.sin(2 * np.pi * car_fs[0]) \
                                + (testY[:, 0, 0] * output[:, 0 + 1] + testY[:, 0, 0 + 1] * output[:, 0]) * np.cos(2 * np.pi * car_fs[0])
        for i in range (1, k):
            m = 2 * i
            test_prediction[:, 0] = test_prediction[:, 0] + (testY[:, 0, m] * output[:, m] - testY[:, 0, m + 1] * output[:, m + 1]) * np.cos(2 * np.pi * car_fs[i]) \
                                    - (testY[:, 0, m] * output[:, m + 1] + testY[:, 0, m + 1] * output[:, m]) * np.sin(2 * np.pi * car_fs[i])
            test_prediction[:, 1] = test_prediction[:, 1] + (testY[:, 0, m] * output[:, m] - testY[:, 0, m + 1] * output[:, m + 1]) * np.sin(2 * np.pi * car_fs[i]) \
                                    + (testY[:, 0, m] * output[:, m + 1] + testY[:, 0, m + 1] * output[:, m]) * np.cos(2 * np.pi * car_fs[i])





        test_groundtrue = Variable(torch.zeros( (testY.shape[0], 2)) )    # [real, imag]
        test_groundtrue[:, 0], test_groundtrue[:, 1] = testY[:, 1, 0], testY[:, 1, 0 + 1]
        for i in range (1, k):
            m = 2 * i
            test_groundtrue[:, 0] = test_groundtrue[:, 0] + testY[:, 1, m]    # real part
            test_groundtrue[:, 1] = test_groundtrue[:, 1] + testY[:, 1, m + 1]    # imaginay part





        loss_real = criterion( test_prediction[:, 0], test_groundtrue[:, 0] )
        loss_imaginary = criterion( test_prediction[:, 1], test_groundtrue[:, 1] )
        loss = loss_real + loss_imaginary


    return loss






x = np.load("/mnt/hdd2/apapadaki/Dataset/70xs.npy")
y = np.load("/mnt/hdd2/apapadaki/Dataset/70ys.npy")

print("\n x.shape = ", x.shape, " y.shape = ", y.shape)






k = 20
print("\n *** SUNM OF {} SIGNALS***".format(k))
fs= 16000
```

```
    car_f = (fs / 2) / k / 2
    car_fs = np.array( [(car_f +  (fs / 2) / k * i) for i in range (k)])


    seq_length = 1 # window/ timesptep
    n_features = 1024
    input_size = n_features
    num_classes = 2 * k  # the output has 2 features

    train_percent =  0.98







    train_size = int(len(y) * train_percent)
    test_size = len(y) - train_size

    dataX = Variable(torch.Tensor(np.array(x)))
    dataY = Variable(torch.Tensor(np.array(y)))
    print("dataX.shape = ", dataX.shape, ", dataY.shape = ", dataY.shape)

    trainX = Variable(torch.Tensor(np.array(x[0 : train_size])))
    trainY = Variable(torch.Tensor(np.array(y[0 : train_size])))
    print("trainX.shape = ", trainX.shape, ", trainY.shape = ", trainY.shape)

    testX = Variable(torch.Tensor(np.array(x[train_size : len(x)])))
    testY = Variable(torch.Tensor(np.array(y[train_size : len(y)])))
    print("testX.shape = ", testX.shape, ", testY.shape = ", testY.shape)





    # Random Train Samples
    rand_indx = np.arange(trainX.shape[0])
    np.random.shuffle(rand_indx)

    trainX = trainX[rand_indx, :, :]
    trainY = trainY[rand_indx, :, :]





    # Save Test arrays
    torch.save(testX, 'testX.pt')
    torch.save(testY, 'testY.pt')






    # Define Network Hyperparameters
    batch_size = 64
    num_epochs = 10000
    learning_rate = 0.0001
    hidden_size = 20
    num_layers = 1




    # Define Model
    lstm = LSTM(num_classes, input_size, hidden_size, num_layers)
    criterion = torch.nn.MSELoss()
    optimizer = torch.optim.Adam(lstm.parameters(), lr = learning_rate)



    # Train the model
    hist, val_hist = np.zeros(num_epochs), np.zeros(num_epochs)
    time_0 = time.time()



    for epoch in range(num_epochs):
        for b in range(0,len(trainX),batch_size):

            inpt = trainX[b : b + batch_size, : , :]
            target = trainY[b : b + batch_size, :, :]

            # Clear stored gradient
        lstm.zero_grad()

        # Initialise hidden state and do Forward/Inference pass
        outputs = lstm(inpt)
    #    print("outputs.shape = ", outputs.shape )





        # Obtain the loss function
        prediction = Variable(torch.zeros( (target.shape[0], 2)) )    # [real, imag]
        prediction[:, 0] = (target[:, 0, 0] * outputs[:, 0] - target[:, 0, 0 + 1] * outputs[:, 0 + 1]) * np.cos(2 * np.pi * car_fs[0]) \
                            - (target[:, 0, 0] * outputs[:, 0 + 1] + target[:, 0, 0 + 1] * outputs[:, 0]) * np.sin(2 * np.pi * car_fs[0])
        prediction[:, 1] = (target[:, 0, 0] * outputs[:, 0] - target[:, 0, 0 + 1] * outputs[:, 0 + 1]) * np.sin(2 * np.pi * car_fs[0]) \
                            + (target[:, 0, 0] * outputs[:, 0 + 1] + target[:, 0, 0 + 1] * outputs[:, 0]) * np.cos(2 * np.pi * car_fs[0])
        for i in range (1, k):
            m = 2 * i
            prediction[:, 0] = prediction[:, 0] + (target[:, 0, m] * outputs[:, m] - target[:, 0, m + 1] * outputs[:, m + 1]) * np.cos(2 * np.pi * car_fs[i]) \
                                - (target[:, 0, m] * outputs[:, m + 1] + target[:, 0, m + 1] * outputs[:, m]) * np.sin(2 * np.pi * car_fs[i])
            prediction[:, 1] = prediction[:, 1] + (target[:, 0, m] * outputs[:, m] - target[:, 0, m + 1] * outputs[:, m + 1]) * np.sin(2 * np.pi * car_fs[i]) \
                                + (target[:, 0, m] * outputs[:, m + 1] + target[:, 0, m + 1] * outputs[:, m]) * np.cos(2 * np.pi * car_fs[i])
    #    print("\n prediction.shape = ", prediction.shape)
    #    print(prediction[0])





        groundtrue = Variable(torch.zeros( (target.shape[0], 2)) )    # [real, imag]
        groundtrue[:, 0], groundtrue[:, 1] = target[:, 1, 0], target
```

```
[:, 1, 0 + 1]
        for i in range (1, k):
            m = 2 * i
            groundtrue[:, 0] = groundtrue[:, 0] + target[:, 1, m]
 # real part
            groundtrue[:, 1] = groundtrue[:, 1] + target[:, 1, m +
1]    # imaginay part
    #    print("\n groundtrue.shape = ", groundtrue.shape)
    #    print(groundtrue[0])





        loss_real = criterion( prediction[:, 0], groundtrue[:, 0] )
        loss_imaginary = criterion( prediction[:, 1], groundtrue[:,
1] )
        loss = loss_real + loss_imaginary

        hist[epoch] = loss.item()
        if epoch % 100 == 0:
          print( "\n Epoch : %d \n    loss : %1.5f" % (epoch, hist[
epoch]) )

        # Zero out gradient, else they will accumulate between
epochs
        optimizer.zero_grad()    # ??

        # Computes gradients and does BackPropagation
        loss.backward()

        # Update parameters
        optimizer.step()

        # Validation
        val_loss = validate(epoch, testX, testY, car_fs)    # !!!
        val_hist[epoch]  = val_loss.item()
        if epoch % 100 == 0:
          print( "    val_loss : %1.5f" % ( val_hist[epoch]) )
time_1 = time.time()
print("\n TRAINING TIME = ", (time_1 - time_0) / 60, " min \n ")




```

```



    np.savetxt("val_hist_70wavs_10Kepochs_64batchsize_20hidden.txt",
   val_hist)
    np.savetxt("hist_70wavs_10Kepochs_64batchsize_20hidden.txt", hist)


    PATH = '/mnt/hdd2/apapadaki/Dataset/generate_models/70
   wavs_10Kepochs_64batchsize_20hidden.pth'
    torch.save(lstm.state_dict(), PATH)    # save the model


```

B.2 Inference

```


    import librosa
    import librosa.display
    import time
    import matplotlib
    import numpy as np
    import matplotlib.pyplot as plt
    import pandas as pd
    import torch
    import torch.nn as nn
    from torch.autograd import Variable
    from sklearn.preprocessing import MinMaxScaler
    import scipy



    # Reproducibility
    np.random.seed(876543210)
    torch.manual_seed(876543210)



    # Define Model
    class LSTM(nn.Module):     # nn.Module : convinient way of
                               # encapsulating parameters

```

```python
    def __init__(self, num_classes, input_size, hidden_size, num_layers):
        super(LSTM, self).__init__()

        self.num_classes = num_classes    # output_size
        self.num_layers = num_layers
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.seq_length = seq_length

        # Define the LSTM layer
        self.lstm = nn.LSTM(input_size = input_size, hidden_size = hidden_size,
                            num_layers = num_layers, batch_first = True)
            # when batch_first = True
            # it will use the second dimension as the sequence dimension :
            # (batch_size, seq_len, n_features)
        # Define the output layer
        self.fc = nn.Linear(hidden_size , num_classes)
            # Applies a linear transformation to the incoming data:
            # y = x W^T + b



    def forward(self, x):
        # initialise our hidden and cell state
        # Don't do this if you want your LSTM to be stateful
        h_0 = Variable(torch.zeros(
            self.num_layers, x.size(0), self.hidden_size))
#         h_0 = Variable(torch.normal( mean = 0, std = 1.5, size = (self.num_layers, x.size(0), self.hidden_size) ))
#         print("\n x.size(0) = ", x.size(0))
#         print("\n h_0.shape = ", h_0.shape)

        c_0 = Variable(torch.zeros(
            self.num_layers, x.size(0), self.hidden_size))
#         c_0 = Variable(torch.normal( mean = 0, std = 1, size = (self.num_layers, x.size(0), self.hidden_size) ))
#         print("\n c_0.shape = ", c_0.shape)

        # Propagate input through LSTM
        ula, (h_out, _) = self.lstm(x, (h_0, c_0))    # returns : lstm_out, ( hidden_state, cell_state )
```

```
#        print("\n ula.shape = ", ula.shape)
#        print("\n A h_out.shape = ", h_out.shape)

        h_out = h_out.view(-1, self.hidden_size)
#        print("\n B h_out.shape = ", h_out.shape)
        out = self.fc(h_out.squeeze(0))    # works like Dence
#        print("\n out.shape = ",out.shape)
        return out




#xs = np.load("/mnt/hdd2/apapadaki/Dataset/1xs.npy")
#ys = np.load("/mnt/hdd2/apapadaki/Dataset/1ys.npy")
#
#testX = Variable(torch.Tensor(np.array(xs)))
#testY = Variable(torch.Tensor(np.array(ys)))





dataX = torch.load('TestX.pt')
dataY = torch.load('TestY.pt')
print("dataX.shape = ", dataX.shape, ", dataY.shape = ", dataY.shape)






k = 20
print("\n *** SUNM OF {} SIGNALS***".format(k))
fs= 16000
car_f = (fs / 2) / k / 2
car_fs = np.array( [(car_f +  (fs / 2) / k * i) for i in range (k)])

# Define Network Hyperparameters
seq_length = 1
n_features = dataX.shape[2]
input_size = n_features
num_classes = 2 * k  # the output has 2 features
hidden_size = 20
```

```python
    num_layers = 1







    # Define Model
    lstm = LSTM(num_classes, input_size, hidden_size, num_layers)

    # Load saved model
    PATH = '/home/katerina/Desktop/fromPegasus/1
   wavs_1Kepochs_64batchsize_20hidden.pth'
    lstm.load_state_dict(torch.load(PATH))    # load the saved model







    #    #    #    now we will go from TENSORS to NUMPY    #    #    #






    ##############################    Predictions
   ##############################
    lstm.eval()
    dataY_plot = dataY.data.numpy()
    print("\n dataY_plot.shape = ", dataY_plot.shape)

    train_predict = lstm(dataX)
    data_predict = train_predict.data.numpy()
    print("\n data_predict.shape = ", data_predict.shape)






    comp_data_predict = (dataY_plot[:, 0, 0] + 1j * dataY_plot[:, 0, 0 +
    1]) * (data_predict[:, 0] + 1j * data_predict[:, 0 + 1]) * np.exp( 1
   j * 2 * np.pi * car_fs[0])
    for i in range (1, k):
        m = 2 * i
        comp_data_predict += (dataY_plot[:, 0, m] + 1j * dataY_plot[:,
   0, m + 1]) * (data_predict[:, m] + 1j * data_predict[:, m + 1]) * np.
   exp( 1j * 2 * np.pi * car_fs[i])
    data_predict = comp_data_predict
    print("\n data_predict.shape = ", data_predict.shape)
    print("\n data_predict[0,:] = ", data_predict[0])





    comp_dataY = dataY_plot[:, 1, 0] + 1j * dataY_plot[:, 1, 0 + 1]
    np.savetxt("S_r_i_TEST.txt", dataY_plot[:, 1, :] )

    for i in range (1, k):
        m = 2 * i
        comp_dataY += dataY_plot[:, 1, m] + 1j * dataY_plot[:, 1, m + 1]
    dataY_plot = comp_dataY
    print("\n dataY_plot.shape = ", dataY_plot.shape)
    print("\n dataY_plot[t=0] ", dataY_plot[0] )
    np.savetxt("S_real_sum_TEST.txt", np.real(dataY_plot))








    matplotlib.rcParams.update({'font.size': 20})
    T = dataY_plot.shape[0] / fs
    n = np.linspace(1, T * fs, T * fs, endpoint = True)






    # Hear Predictions
    scipy.io.wavfile.write("1wavs_1Kepochs_64batchsize_100hidden_gtrue.
   wav", fs, np.real(dataY_plot).flatten())
    scipy.io.wavfile.write("1wavs_1Kepochs_64batchsize_100hidden_pred.
   wav", fs, np.real(data_predict).flatten())


```

```

    realmse = (np.square(np.real(dataY_plot) - np.real(data_predict))).mean(axis=0)
    immse = (np.square(np.imag(dataY_plot) - np.imag(data_predict))).mean(axis=0)
    realstd = (np.square(np.real(dataY_plot) - np.real( dataY_plot.mean(axis=0) ))).mean(axis=0)
    imstd = (np.square(np.imag(dataY_plot) - np.imag( dataY_plot).mean(axis=0) )).mean(axis=0)
    print("TRAIN REL.MSE = ", (realmse + immse) / (realstd + imstd) )
    print("TRAIN REL.MSE = ", (realmse ) / (realstd ) )


```

Bibliography

[1] Alan V. Oppenheim, Ronald W. Schafer. "Discrete-Time Signal Processing", Third Edition, 2009. 8, 37

[2] Yuxuan Wang, RJ Skerry-Ryan, Daisy Stanton, Yonghui Wu, Ron J. Weiss, et al. "Tacotron: Towards End-to-End Speech Synthesis", arXiv:1703.10135, 2017. 4

[3] J. Shen, R. Pang, R. J. Weiss, M. Schuster, N. Jaitly, Z. Yang, Z. Chen, et al. "Natural TTS Synthesis by Conditioning WaveNet on Mel Spectrogram Predictions", arXiv:1712.05884, 2018. 4

[4] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalch-brenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio", arxiv:1609.03499, 2016. 5, 24

[5] N. Kalchbrenner, E. Elsen, K. Simonyan, S. Noury, N. Casagrande, E. Lockhart, F. Stimberg, A. van den Oord, S. Dieleman, and K. Kavukcuoglu. "Efficient neural audio synthesis", arXiv:1802.08435, 2018. 1, 5, 23, 24, 53

[6] Jean-Marc Valin and Jan Skoglund. "Lpcnet: Improving neural speech synthesis through linear prediction", 2018. 5

[7] Ryan Prenger, Rafael Valle, and Bryan Catanzaro. "Waveglow: A flow-based generative network for speech synthesis", 2018. 5

[8] Kundan Kumar, Rithesh Kumar, Thibault de Boissiere, Lucas Gestin, Wei Zhen Teoh, Jose Sotelo, Alexandre de Brebisson, Yoshua Bengio, and Aaron Courville. "Melgan: Generative adversarial networks for conditional waveform synthesis", 2019. 5

[9] D. Paul, Y. Pantazis, Y. Stylianou. "Speaker Conditional WaveRNN: Towards Universal Neural Vocoder for Unseen Speaker and Recording Conditions", arXiv:2008.05289, 2020. 5

[10] D. Paul, M. PV Shifas, Y. Pantazis, Y. Stylianou. "Enhancing Speech Intelligibility in Text-To-Speech Synthesis using Speaking Style Conversion", arXiv:2008.05809, 2020. 5

[11] X. Wang, S. Takaki, J. Yamagishi."Neural source-filter-based waveform model for statistical parametric speech synthesis", 2018. 5

[12] G. P Kafentzis, G. Degottex, O. Rosec, Y. Stylianou. "Time-scale modifications based on an adaptive harmonic model". In IEEE International Conference on Acoustics Speech and Signal Processing (ICASSP), 2013. 5

[13] T. Quatieri. "Discrete-time Speech Signal Processing: Principles and Practice". Prentice Hall Press, Upper Saddle River, NJ, USA, first edition, 2001. 1

[14] R. J. McAulay and T. F. Quatieri. "Speech Analysis/Synthesis based on a Sinusoidal Representation", IEEE Trans. on Acoustics, Speech and Signal Processing, 34:744â754, 1986. 5

[15] S. M. Kay. "Modern Spectral Estimation: Theory and Applications", Prentise-Hall, Englewood Cliffs, NJ, 1988.

[16] U. Greeshma, S. Annalakshmi. "Artificial Neural Network", International Journal of Scientific Engineering Research, Volume 6, Issue 4, April-2015. 11

[17] "Text-to-Speech", (https://github.com/dipjyoti92/Text-to-Speech), 2021. 25, 26, 27, 28

[18] K. Ito, L. Johnson. "The LJ Speech Dataset", (https://keithito.com/LJ-Speech-Dataset), 2017. 28, 36, 46

[19] Iman. "Uniform Filter Bank", (https://www.mathworks.com/matlabcentral/fileexchange/35053-uniform-filter-bank), MATLAB Central File Exchange, Retrieved September 30, 2020. 36