MOSCOW INSTITUTE OF PHYSICS AND TECHNOLOGY
(National Research University)

PHYSTECH SCHOOL OF APPLIED MATHEMATICS AND INFORMATICS
Chair of Discrete Mathematics

BACHELOR THESIS
«Applied Mathematics and Informatics»
«Computer Science»

DEEP LEARNING BASED END-TO-END TEXT-INDEPENDENT SPEAKER VERIFICATION

Student:
Jimenez Velado Gabriel Antonio

Scientific Supervisor:
Sodikov Makhmudzhon Mamurzhon ogly

Moscow, 2023

Acknowledgments

I would like to express my sincere gratitude to Miguel Lacayo, Vivian de Lacayo, Lilian Zelaya, Vicky Arias and my aunt, Carmen Elena de Velado, for their trust and invaluable support in helping me to complete my studies.

I would like to honor and dedicate this work to my father, Carlos Jimenez, and my uncle, Jorge Velado, whom I also hold in high regard and love as a father. Without their life guidance, loving support, dedication, selfless sacrifices and example, I wouldn't have achieved academic success. This accomplishment is as much a testament to their enduring influence as it is a reflection of my efforts.

Abstract

Speech processing is one of the most active areas of research and development in Deep Learning. However, compared to other more common domains, the progress in speech processing has been relatively less pronounced. This is primarily due to the inherent challenge that imposes the extraction of meaningful statistical and discriminative features from speech signals. Previous advancements in speech processing often relied on manual feature modeling, requiring significant effort from developers and researchers.

Recent experimentation with neural networks architectures has shown promising results in obtaining low-dimensional vector representations of speech signals, while requiring minimal preprocessing and simplifying the feature extraction process. Embeddings obtained from these experiments have reported to be comparable, and sometimes even superior, to traditional feature modeling methods. This has impulsed the development of models that effectively address challenging speech processing tasks, such as speaker verification.

Speaker verification is the task of accepting or rejecting a speakers identity claim through the analysis of their speech signals. In this work we focus on text-independent speaker verification, which is the most challenging subset of speaker verification tasks. For this task the verification process is not dependent from a specific phrase, meaning that the user can claim their identity by speaking any phrase, and the model must verify this claim based on the characteristics extracted from this arbitrary speech signal.

In traditional speaker verification systems the pipeline consists of three distinct steps: development, enrollment and verification. For which the development and enrollment steps are trained separately from the verification model. In this work, we present an end-to-end approach based on the Generalized End-to-End (GE2E) method proposed by Google researchers. Our approach employs a single network architecture that directly maps utterances' log-Mel filterbank energy vectors to discriminative speaker embeddings. The architecture emulates the three steps of the traditional speaker verification pipeline while optimizing a unified loss function that encourages close proximity of utterance embeddings from the same speaker while increasing the distance from embeddings coming from different speakers in the embedding space. Our implementation is designed to be trained on small scale datasets, including fewer than $1,500$ speakers. Despite this limitation, it has yielded favorable results, demonstrating the effectiveness of our architecture for speaker modeling and verification, which further emphasizes the potential of deep learning in advancing the field of speech processing.

Contents

1 Digital Signal Processing

1.1 Sound Waves

In mathematical terms, a wave can be described as a disturbance that propagates through a medium, carrying energy from one point to another. *Sound* is a type of signal known as an *acoustic wave*, which propagates through a transmission medium such as a gas, liquid or solid. In this work we are interested in sound signals that traverse trough air. These sound signals manifest as pressure variations travelling through the air, they are often referred to as *sound waves*. Waves exhibit certain properties, including amplitude and frequency:

The *amplitude* of a wave refers to the maximum displacement or distance from the equilibrium position. In the context of sound waves, the amplitude represents the maximum variation in air pressure caused by the wave.

The *frequency* of a wave refers to the number of complete oscillations or cycles that the wave undergoes in a given time period. In the case of sound waves, the frequency corresponds to the rate at which the air pressure oscillates back and forth.

To exemplify these concepts, we can consider a simple sinus function. The sinus function serves as a fundamental representation of a wave, exhibiting periodicity denoted by T, i.e. $\exists T \in \mathbb{R}^+ : x(t+T) = x(t) \forall t$. Is also characterized by its continuity, constant amplitude, frequency and phase over time. This results in a smooth and uninterrupted oscillation, making it an *ideal* model for demonstrating wave behaviour:

$$x(t) = A\sin(2\pi f t + \phi) = A\sin(\omega t + \phi)$$

Here A denotes the amplitude, f represents the frequency, $\omega = 2\pi f$ is the *angular frequency* denoting the rate of change of the function's argument (measured) in *radians per second*. The initial shift or displacement of the wave relative to a reference point is represented by the *phase*, denoted as ϕ.

The term *sinusoid* describes any wave with characteristics of a sine wave. Consequently, a cosine is also considered sinusoidal. Sinusoidal waves are of mathematical importance due to their ability to preserve their wave shape when added to another sine (in the case of a sine wave) or cosine (in the case of a cosine wave) of the same frequency and arbitrary phase and magnitude. It is the only periodic waveform that has this property.

The sinusoidal wave is the simplest form of a wave, it serves as a fundamental building block for understanding more complex waveforms and their behaviours across various phenomena.

Sound waves extend beyond the simplicity of a single sinusoidal wave, they are typically composed of a complex combination of multiple sinusoidal waves, each with its own frequency, amplitude, and phase. Understanding the complexity of sound waves requires considering the interplay of various frequencies, amplitudes, and temporal characteristics. This complex interplay of multiple frequencies and amplitudes gives rise to the diverse range of sounds. In subsequent sections of this work we will discuss about the complexity of sound waves and how the results from Fourier analysis enables us to decompose complex sound waves into a

summation of sinusoidal waves.

Airflow itself is not audible as a sound, to obtain sound we need to obstruct airflow to obtain an oscillation or turbulence. The physical process of producing sound begins by contracting the lungs, pushing out air from the lungs through the throat, oral and nasal cavities. Oscillations are primarily produced when the vocal cords are tensioned appropriately, this produces *Voiced Sounds*. Oscillations produced by other parts of the speech production organs, such as letting the tongue oscillate against the teeth should not be confused with voiced sounds, which are always generated by oscillations in the vocal cords. Sounds without oscillations in the vocal cords are known as *unvoiced sounds*.

We must emphasize on the fact that not all voiced sounds can be considered as speech, to be more precise, in the context of this work, we will rely on the following definition of *Speech*: "*Sounds (signals) articulated by humans that are communicational meaningful*".

We are primarily interested in analysis and processing of such waveforms in digital systems. We will therefore always assume that the acoustic speech signals have been captured by a microphone and converted to a digital form.

For the rest of this work, we will use the terms 'sound signals', 'signals', 'sound waves' and 'speech signal' interchangeable.

In their natural form, sound signals are analog, which means they can take on any value within a continuous range. They exhibit a smooth and continuous variation in air pressure caused by the sound wave.

However, when sound signals are captured by a microphone they are *digitized* or *discretized*, i.e, transferring the continuous captured signal $x(t)$ into its *discrete* counterpart $x[n]$ (In most literature related to digital signal processing $x[n]$ is used instead of $x(n)$ to denote the discrete representation of an analog signal $x(t)$, we choose to follow this convention). This process is known as *Analog to Digital Conversion* or *"a-to-d" conversion*:

- A "real-world" sound signal is captured using a microphone which has a diaphragm that is pushed back and forth according to the compression and rarefaction of the sounding pressure waveform. The microphone transforms this displacement into a time-varying voltage - an analog (continuous-time) electrical signal.

- the analog-to-digital conversion is done using hardware called an *analog-to-digital converter (ADC)*, the ADC accomplish this task in two steps:

  Sampling: is the process of capturing "sample" values - measuring the amplitude from the continuous signal at a regularly spaced time intervals.

  The time interval (in seconds) between consecutive sample measures is called the *Sampling Period* or *Sampling Interval* T_s and is inversely related to the *Sampling Rate* f_s which is the average number of samples obtained in one second: $f_s = \frac{1}{T_s}$ with the unit *samples per second*, usually referred as *hertz* (Hz). Sampling involves transforming the continuous

time variable t into a set of discrete times that are integer multiples of the sampling period T_s, that is, $t \rightarrow nT_s$ where $n \in \mathbb{N}$ and corresponds to the index in the sequence.

The continuous signal $x(t)$ is sampled at discrete intervals $\{T_s, 2T_s, 3T_s, \cdots, nT_s\}$, resulting in a sequence of sampled values $\{x[n]\}$, effectively converting the signal from continuous-time to discrete-time representation. For example, recall the sinusoid function as a function of the continuous variable t: $x(t) = A\sin(2\pi ft+\phi)$ by sampling this function we obtain: $x[n] = x(nT_s) = A\sin(2\pi fnT_s + \phi) = A\sin(\omega n + \phi)$ where $\omega = 2\pi\frac{f}{f_s}$ is called *normalized radian frequency* and n represents normalized time.

For example, CD's are typically sampled at $44.1kHz$, which means that $\sim 41,000$ amplitude points are sampled in a second.

Quantization: While sampling captures the amplitude values of the continuous signal at specific time points, it does not address the issue of representing those values digitally. In a digital system, each sampled value needs to be stored and processed using a finite number of bits. Quantization enables the mapping of the continuous amplitude values to a limited set of discrete levels that can be accurately represented using digital binary codes. *Levels* refer to the distinct values that can be represented by the digital samples, they represent the quantized amplitude values that the analog signal can take on in the digital representation.

The quantization process introduces some level of approximation or rounding, as the continuous signal's infinite amplitude possibilities are mapped to a finite set of discrete levels, this involves the projection of the amplitudes to the closest bit values.

The number of bits used to represent each sample of the analog signal is know as *bit-depth*, the bit-depth directly corresponds to the number of discrete levels that can be represented.

A higher bit-depth provides a greater number of discrete levels, allowing for a more accurate representation of the analog signal. It enables finer distinctions between amplitude values and reduces quantization errors. As a result, higher bit-depths generally lead to improved fidelity and better signal quality.

However, it's important to note that using a higher bit-depth also increases the data size and computational requirements. It requires more memory to store the samples and additional processing power for calculations. Therefore, the choice of an appropriate bit-depth involves considering the specific requirements, such as the desired signal quality, available resources, and the trade-off between fidelity and data efficiency.

As an example, consider again the CD, which, as mentioned previously, is typically sampled at a $44.1kHz$ sampling rate and has a bit-depth of 16 bits, this means that, each audio sample can have $2^{16}(65,536)$ possible discrete amplitude levels, i.e., they are mapped within the range of $(-32768, 32767)$ since audio signals are both positive and negative.

One of the most important results in digital signal processing is the *Nyquist-Shannon Sam-*

pling Theorem which establishes a sufficient condition for a sample rate that permits a discrete sequence of samples to capture all the information from a continuous-time signal of finite bandwidth. *Bandwidth* refers to the range of frequencies present in a signal, is measured in Hz and represents the difference between the highest and lowest frequencies present in a signal $B = f_{max} - f_{min}$.

Theorem (**Nyquist-Shannon Sampling**). *If a function $x(t)$ contains no frequencies higher that B hertz, then it can be completely determined from its ordinates at a sequence of points spaced less than $\frac{1}{2B}$ seconds apart.*

The following are direct consequences of the Nyquist-Shannon Sampling theorem:

In order for a bandlimited signal, a signal whose frequency spectrum lies between f_{min} and f_{max}, to be fully reconstructed, it must be sampled at a rate of $f_s > 2B$, where $2B$ is the *Nyquist frequency*, usually $f_{min} = 0Hz$ and the Nyquist frequency is then $2f_{max}$.

Half the sampling rate, i.e., $\frac{f_s}{2}$ is the highest frequency component that can be accurately represented, this value is referred as the *Nyquist limit*. This means that, for example, if the sampling rate is $f_s = 44kHz > 2f_{max}$, then signals in the frequency range $[0Hz, 22,000Hz]$ can be uniquely described with this sampling rate.

No information is lost if a signal is sampled above the Nyquist frequency, and no additional information is gained by sampling faster than this rate.

1.2 Perception of Sound

Before we examine the perception of sound by the human auditory system, is important to discuss about some other physical characteristic of sound waves:

Sound Power is the rate at which energy (in Joules, J) is transferred, i.e., is energy per second emitted by a sound in all directions. This quantity is measured in Watts (W, $1\ W = 1\ J/s$)

Sound Intensity is the measure of energy carried by the wave per unit area. It represents the power per unit area that the sound wave delivers to a surface perpendicular to its direction of propagation. In other words, it quantifies the concentration of energy in the sound wave. This quantity is measured in Watts per meter squared (W/m^2)

The minimum sound intensity or pressure level that can be detected by the human auditory system is referred as the *threshold of hearing (TOH)*, it varies among individuals, however an average quantity has been experimentally defined as $I_{TOH} = 10^{-12}W/m^2$.

The *threshold of pain (TOP)*, on the other hand,is the highest sound level that the human ear can tolerate without experiencing physical discomfort or pain. It varies among individuals, but an average quantity has been experimentally defined as $I_{TOP} = 10W/m^2$

As we have mentioned previously, when a sound wave travels through air, it causes the air particles to oscillate back and forth, transmitting the energy of the wave. The human perception of sound arises from the intricate process of our auditory system detecting and interpreting these variations in air pressure caused by sound waves. *Due to the physiological structure of*

the auditory human system, human perception of sound is nonlinear. This work will refrain from delving into specific physiological details as they are beyond the scope of this study and unnecessary for comprehending the subsequent sections, however, what is noteworthy for us is that this nonlineairity manifests in two aspects: nonlinearity of frequency and nonlinearity of intensity.

While frequency and intensity represent physical quantities inherent to sound waves, their direct interpretation from the standpoint of the human auditory perception is not inherently intuitive, this is primarily due to the nonlinear nature of sound perception. To address this, concepts such as pitch and loudness have been introduced to bridge this gap, offering more accessible descriptors for these fundamental properties:

Pitch refers to the subjective perception of the frequency of a sound. A high-pitched sound corresponds to a high-frequency sound wave and a low-pitched sound corresponds to a low frequency sound wave. Pitch is also influenced by changes in frequency.

Loudness, on the other hand, is the subjective perception of the intensity of a sound wave.

When modeling speech signals, it is crucial to consider these nonlinearities. Next, we will discuss how these nonlinearities are modeled.

Nonlinearity of frequency - Mel Scale

Throughout the study of sound perception, various scales have been introduced to model the nonlinearity of frequency. Among these scales, the Mel scale stands out, due to its ability to better align with human perception, its successful application in speech processing, and its utility in deep learning make it the most widely used scale in the study of sound perception. As it will be explored later in this work, in our experiments we made use of the Mel scale when processing the speech signals.

The *Mel Scale* is a perceptual scale of pitches that approximates the human ear's response to different frequencies. It is designed to account for the non-linear nature of human hearing, where our perception of pitch is not directly proportional to the physical frequency of a sound wave.

The Mel scale divides the audible frequency range into a series of equally spaced perceptual units called *Mel frequencies*. These Mel frequencies are based on psychoacoustic studies that have shown how the human ear perceives differences in pitch. The units of the scale are the *Mel*, a dimensionless unit representing a perceptual pitch difference.

The Mel scale is logarithmically spaced, meaning that the intervals between Mel frequencies are not linearly proportional to the corresponding physical frequencies. Humans are much better at discerning small changes in pitch at low frequencies than they are at high frequencies, because of this at lower frequencies the Mel scale has finer resolution, while at higher frequencies it has coarser resolution.

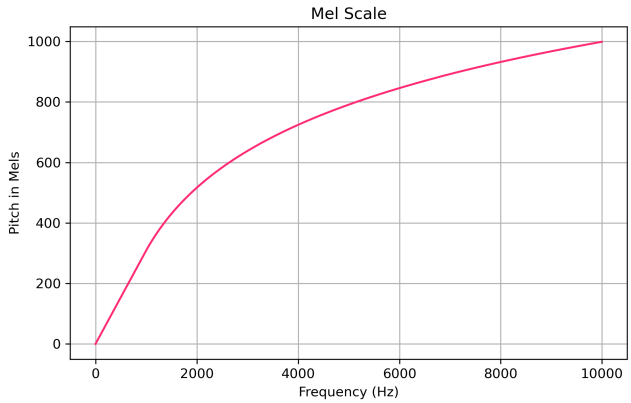

Figure 1.1: Mel Scale.

The transformation from physical frequency (in Hz) to Mel frequency is achieved using a formula known as the *Mel scale transformation*: $f_{mel} = 2595 \log_{10}(1 + f_{Hz}/700) = 1127 \ln(1 + f_{Hz}/700)$.

By using the Mel scale, audio and speech processing algorithms can better capture the perceptual characteristics of human hearing. It enables the representation of audio signals in a way that aligns with our subjective perception of pitch, making it useful for tasks such as speech recognition, music analysis, and sound synthesis.

Nonlinearity of intensity - Decibel Scale

Human hearing is more sensitive to signals with small amplitude but less sensitive to signals with large amplitude, in the *Decibel Scale*, the perceived loudness of a sound is proportional to the logarithm of its intensity. The logarithmic nature of the scale reflects the way our ears perceive changes in sound intensity. Our hearing is more sensitive to relative changes in intensity rather than absolute changes. In a linear scale, for example, a doubling of sound intensity would correspond to a doubling of the numerical value, which doesn't reflect the actual perceptual experience.

The values on the decibel scale represent the ratio between two intensity values: $I_{dB} = 10 \log_{10}(\frac{I}{I_0})$, in which one of those is an intensity of reference I_0, usually the threshold of hearing I_{TOH}, and the other one is the intensity to which we want to compare it (the one that is being measured) I. In this way $0dB$ corresponds to the intensity of the threshold of hearing ($10^{-12} W/m^2$), for example, a sound 10 times more intense ($10^{11} W/m^2$) corresponds to a sound level of $10dB$. So, a difference of 10 decibels corresponds to a tenfold change in sound intensity.

The decibel scale allows for a more accurate representation of the wide range of sound intensities that humans can perceive. It compresses the large range of possible intensity values into a more manageable and intuitive scale. By using a logarithmic scale, small changes in intensity can be represented as meaningful differences in decibel values.

1.3 Short-Time Analysis

Framing and Windowing

In the context of this work an *utterance* refers to an audio clip containing a segment of a speech signal produced by a specific speaker.

A spoken sentence comprises a sequence of phonemes, speech signals are thus time-variant in nature. To effectively extract meaningful information from these signals, a technique known as *framing* is employed. This technique involves splitting the signal into sufficiently short overlapping segments ($10ms \sim 30ms$), referred to as *frames*, such that, heuristically speaking, the signal remain relatively unchanged within each frame. By doing so, we can obtain discrete segments that facilitate the analysis and processing of speech signals.

Frames have two parameters:

Frame length/duration: Duration (in ms) covered by the frame.

Frame step/stride/hop: Temporal distance (in ms) between the starting points of two consecutive frames.

Based on the frame length there are three possible scenarios:

- $length > step$: Consecutive frames have overlapping samples. This is the most common case, as overlapping frames help capture the temporal evolution of the signal more accurately.
- $length = step$: Frames are mutually exclusive, i.e., the next frame starts right after the previous one ends, without any overlap
- $length < step$: Gaps exist between consecutive frames, because of this information is lost since some parts of the signal may not be captured by any frame.

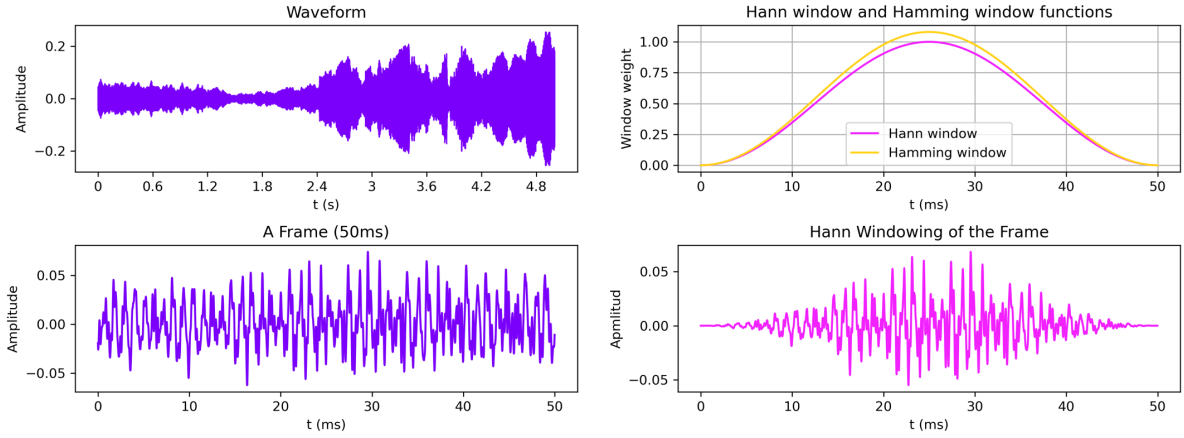

Figure 1.2: A signal, a frame from the signal, Hann and Hamming functions and the windowed frame plots.

When we divide a continuous signal into frames, we essentially isolate short segments of the signal for analysis. However, abruptly cutting the signal at the frame boundaries can

introduce artifacts and distortions due to the sudden discontinuity, which are incongruent with the real-world signal.

A technique called *Windowing* is employed to mitigate these issues by gradually tapering the signal at the beginning and end of each frame. It involves multiplying the signal within each frame by a *window function*, which smoothly transitions the signal to zero towards the frame edges. This tapering effect minimizes the abrupt changes that would otherwise occur at the frame boundaries ensuring a more seamless and continuous transition between frames. Also, the other purpose of windowing the frames before further processing them with a discrete Fourier transform (explained in a subsequent section), is to minimize spectral leakage, which happens when non-cyclic data is Fourier-transformed.

Consider a frame $\{x[n]\}$ of size L (number of samples within the frame), where $n \in \mathbb{N}$ represents the index of the sample, i.e., $0 \leq n \leq L-1$. The window function define the weights $\{w[n]\}$ in such way that, they are close to zero near the edges of the frame, and they are close to one in the neighborhood of the middle of the frame to keep the signal untouched in the middle (see Figure 1.2). The windowed signal, denoted as $\{\hat{x}[n]\}$, is obtained by the element-wise multiplication of the samples in the frame and their corresponding weights:

$$\hat{x}[n] = x[n] \cdot w[n]$$

The *Hann* and *Hamming* functions are the most commonly used window functions. They are defined as follows:

$$w_H[n] = (1-\alpha) - \alpha \cos(\frac{2\pi}{L-1}n)$$

where the Hann window uses a value of $\alpha = 0.5$ and the Hamming window uses a value of $\alpha = 0.46$

As illustrated in Figure 1.2, these window functions have a shape that gradually decreases the amplitude of the signal towards the frame edges. This specific shape helps in preserving the spectral properties of the signal within the frame while reducing spectral leakage and aliasing effects.

Time-domain and Frequency-domain

As we discussed previously, during the process of analog-to-digital conversion (ADC), an analog signal is sampled at regular discrete time intervals, followed by quantization of each sample into a specific digital value. Consequently, the digital signal is formed as a series of discrete samples that capture the amplitude variations of the original analog signal over time. Each sample corresponds to a distinct time point, collectively representing a discrete representation of the signal's temporal behaviour. As a result, the outcome is a digital representation of the signal in the *time domain*.

On the other hand, we can also analyze a digital signal in the *frequency domain*, which provides insights into its frequency characteristics rather than its temporal behavior. While a

time-domain plot illustrates how a signal changes over time, a frequency-domain plot reveals the distribution of the different frequencies present in the signal and their respective magnitudes.

The frequency domain analysis involves decomposing the signal into its constituent frequencies using mathematical techniques like the Fourier transform. By analyzing the signal in terms of various frequencies, we shift our focus from its variations at different time points to the amount of signal energy present within specific frequency ranges. This approach enables us to identify and assess the presence and relative strength of different frequencies in the signal. In essence, the frequency domain analysis provides a detailed understanding of the signal's frequency composition and allows us to explore its spectral characteristics.

In the frequency domain, the representation of a signal is commonly referred to as the *spectrum*. The spectrum depicts the relationship between frequency and magnitude of the signal. The x-axis of the spectrum represents the frequency. It shows the range of frequencies present in the signal, typically spanning from $0Hz$ to the Nyquist frequency, which, as discussed before, is half the sampling rate of the signal. The y-axis of the spectrum represents the magnitude or amplitude of each frequency component. It indicates the strength or intensity of each frequency present in the signal. The magnitude can be represented in various units, such as decibels (dB) or linear amplitude values, depending on the specific context and application.

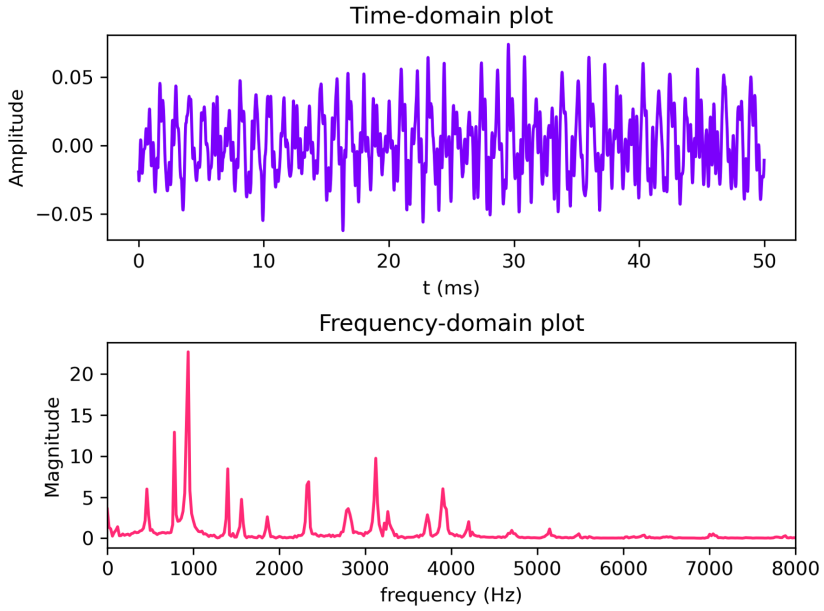

Figure 1.3: Time-domain and Frequency-domain plots of a signal.

Most of the state-of-the art network architectures for speech processing rely on frequency domain data rather than time domain data. Additionally, as we will discuss later, numerous research papers have demonstrated the efficiency of utilizing frequency domain data in the realm of deep learning. Networks exhibit remarkable proficiency in extracting speech and/or speaker-specific acoustic characteristics from spectrograms and mel-spectrograms, both of which represent the signal in the frequency domain. We will delve into these representations in more

detail in a subsequent section.

Short Time Fourier Transform

We now turn our attention to what is perhaps the most important concept in digital speech processing, the *Short-time Fourier Transform*.

Let us recall one of the fundamental results of Fourier Analysis: a periodic function can be expressed as an infinite sum of orthogonal functions. This result is known as the *Fourier Series*. Here, the term "orthogonal" refers to a system of continuous functions $\{e_k(x)\}$ defined over a closed interval $[a, b]$ such that the scalar product $(e_n, e_m) := \int_a^b e_n(x)e_m(x)dx$ is equal to 0 for $n, m \in \mathbb{N}$ and $n \neq m$.

The *Trigonometric System* is the orthogonal system of functions:

$\{\frac{1}{2}, \sin(\frac{\pi x}{l}), \cos(\frac{\pi x}{l}), \cdots, \sin(\frac{\pi kx}{l}), \cos(\frac{\pi kx}{l}), \cdots\}$ where $k \in \mathbb{N}$ and $l \in \mathbb{R}^+$ is a fixed number. The trigonometric system is orthogonal over any closed interval of length $2l$, i.e, $[-l, l]$, $[d, d + 2l]$ and $[d - l, d + l], d \in \mathbb{R}$

The trigonometric system is commonly used in Fourier series expansions due to their periodicity and orthogonal properties. When this system is employed, the resulting series expansion is referred as a Fourier Trigonometric Series.

A function f is said to be *Absolutely Integrable* on a finite or infinite (open if infinite) interval $[a, b]$ if: $\exists$ a finite set of points $\{x_j\}_{j=0}^k$ such that $a = x_0 < x_1 < \cdots < x_k = b$, $\forall[\alpha, \beta] \subset (x_{i-1}, x_i); (i = 1, \cdots, k)$ the function f is Riemann Integrable, i.e., $\exists \int_\beta^\alpha f(x)dx$. And the integral $\int_a^b |f(x)|dx$ converges.

Let f be a periodic function with period $2T \in \mathbb{R}^+$ that is absolutely integrable over $[-T, T]$, the *Trigonometric Fourier Series* of f over the interval $[-T, T]$ is given by

$$f(x) \sim \frac{a_0}{2} + \sum_{k=1}^{\infty}(a_k \cos(\frac{\pi kx}{T}) + b_k \sin(\frac{\pi kx}{T}))$$

where a_0, a_k and b_k are the *Fourier coefficients*, which determine the contribution of each cosine and sine term in the series expansion and are equal to

$$a_0 = \frac{1}{T}\int_{-T}^{T} f(x)dx; a_k = \frac{1}{T}\int_{-T}^{T} f(x)\cos(\frac{\pi kx}{T})dx; b_k = \frac{1}{T}\int_{-T}^{T} f(x)\sin(\frac{\pi kx}{T})dx$$

where $k \in \mathbb{N}$. It is worth mentioning that if the function f is defined over the interval $[a, b]$ then the previous definition can be applied with $T = \frac{b-a}{2}$.

Using " $\sim$ " instead of " = " is a common convention when representing Fourier series expansions, because they provide an approximation of the original function rather than an exact equality. Fourier series represent periodic functions as infinite series of harmonics, and in most cases, the series converges to the function but might not coincide at every point, also the Fourier series is an infinite sum, including terms with varying coefficients. Using the equality sign " = " might imply that the series representation holds exactly at every point, which is not the case due

to convergence properties.

Recall that from Euler's formula it can be obtained the following definitions for the sinusoidal functions: $\cos(z) = \frac{e^{iz}+e^{-iz}}{2}, \sin(z) = \frac{e^{iz}-e^{-iz}}{2i}$

Using these formulas, we can rewrite sine and cosine terms from the Fourier series in terms of complex exponentials:

$$f(x) \sim \frac{a_0}{2} + \sum_{k=1}^{\infty} a_k \frac{e^{i\frac{\pi kx}{T}} + e^{-i\frac{\pi kx}{T}}}{2} + \sum_{k=1}^{\infty} b_k \frac{e^{i\frac{\pi kx}{T}} - e^{-i\frac{\pi kx}{T}}}{2i}$$

Further rearranging:

$f(x) \sim \frac{a_0}{2} + \sum_{k=1}^{\infty} a_k \frac{e^{i\frac{\pi kx}{T}} + e^{-i\frac{\pi kx}{T}}}{2} - \sum_{k=1}^{\infty} b_k \frac{ie^{i\frac{\pi kx}{T}} - ie^{-i\frac{\pi kx}{T}}}{2} = \frac{a_0}{2} + \sum_{k=1}^{\infty} \frac{a_k - ib_k}{2} e^{i\frac{\pi kx}{T}} + \sum_{k=1}^{\infty} \frac{a_k + ib_k}{2} e^{-i\frac{\pi kx}{T}}$

changing the summation index k to $-k$ for the second sum we obtain:

$f(x) \sim \frac{a_0}{2} + \sum_{k=1}^{\infty} \frac{a_k - ib_k}{2} e^{i\frac{\pi kx}{T}} + \sum_{k=-\infty}^{-1} \frac{a_{-k} + ib_{-k}}{2} e^{i\frac{\pi kx}{T}}$

We can combine the exponential terms to obtain the *Complex Fourier Series* of function f for an arbitrary period $2T$ (or over an interval $[a, b] : T = \frac{b-a}{2}$):

$$f(x) \sim \sum_{k=-\infty}^{\infty} c_k e^{i\frac{\pi kx}{T}}$$

where c_k is the *Complex Fourier coefficient*: $c_k = \begin{cases} \frac{a_k - ib_k}{2}, & \text{if } k > 0, \\ \frac{a_0}{2}, & \text{if } k = 0, \\ \frac{a_{|k|} + ib_{|k|}}{2}, & \text{if } k < 0. \end{cases}$

Furthermore, using the formulas for the Fourier coefficients a_0, a_k and b_k, it can be obtained that

$$c_k = \frac{1}{2T} \int_{-T}^{T} f(x) e^{-i\frac{\pi kx}{T}} dx$$

For the rest of this work, we will use the term "Fourier Series" as a shorthand, instead of "Complex Fourier Series" since we will only make use of these type of series expansions.

We will now consider that the Fourier series of f converges to f, and we will use the continuous variable $t \in \mathbb{R}$ instead of $x \in \mathbb{R}$. This change in variable notation does not alter the previously discussed results or any subsequent ones in this section. However, it serves as more meaningful representation for the purposes of this work, as we are studying signals that vary over time.

Until this point, we have considered Fourier series of a periodic function f over some fixed closed interval $[-T, T]$ (or over an interval $[a, b] : T = \frac{b-a}{2}$). Let's now examine what happens when we take the limit of the complex Fourier coefficients as T tends to infinity. This effectively will result in the Fourier series no longer describing a fixed interval $[-T, T]$ but rather describing the whole real line.

Let us introduce the following notation: $\omega_k = \frac{\pi k}{T}, \Delta\omega_k = \frac{\pi}{T}, F(\xi) = \int_{-T}^{T} f(t) e^{-i\xi t} dt$

With this notation we can express the Fourier coefficients and series as

$$c_n = \frac{1}{2T}F(\omega_k), f(t) = \sum_{k=-\infty}^{\infty} \frac{1}{2T}F(\omega_k)e^{i\omega_k t} = \sum_{k=-\infty}^{\infty} \frac{1}{2\pi}F(\omega_k)e^{i\omega_k t}\Delta\omega_k$$

The expression $f(t) = \sum_{k=-\infty}^{\infty} \frac{1}{2\pi}F(\omega_k)e^{i\omega_k t}\Delta\omega_k$ can be interpreted as the *Riemann Sum* of $\frac{1}{2\pi}F$ over $[-T, T]$ (alternately over $[a, b] : T = \frac{b-a}{2}$) with a partition $P = \{\omega_i\}_{i=0}^{n} = (\omega_0, \cdots, \omega_n) : -T = \omega_0 < \omega_1 < \cdots < \omega_n = T$ where $\Delta\omega_k = \omega_k - \omega_{k-1}$. As $T \to \infty$, it follows that $\Delta\omega_k \to 0$, therefore, we obtain the *Riemann Integral*:

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} F(\omega)e^{i\omega t}d\omega \tag{1.1}$$

where

$$F(\omega) = \int_{-\infty}^{+\infty} f(t)e^{-i\omega t}dt \tag{1.2}$$

is the *Forward Fourier Transform*, commonly simply referred to as *Fourier Transform* and eq.(1.1) is the *Inverse Fourier Transform*. $F(\omega)$ represents the complex amplitude of the frequency component ω in the function $f(t)$. By calculating this integral, we can obtain the spectrum of frequencies present in the function $f(t)$, represented by the complex function $F(\omega)$. The result provides information about the amplitude and phase of each frequency component.

It is worth noting that, as we pointed out previously, f, and consequently F, are absolutely integrable and therefore Riemann integrable.

Since we are working with digitized signals, i.e., discrete-time signals, we are interested in the *discrete* representations of the Fourier transforms: for a time-discrete set of N points (digitized signal) $\{x[n]\}$ the *Discrete Fourier Transform (DFT)* is defined as a sampled version of the Transform shown above:

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-i\frac{2\pi}{N}kn} \tag{1.3}$$

where $X[k]$ represents the k-th frequency component, and $x[n]$ is the n-th sample of the discrete-time signal. The index k ranges from 0 to $N-1$, representing the different frequency bins.

The DFT decomposes the input sequence into a sum of complex sinusoidal functions at different frequencies. The result $X[k]$ is the complex valued frequency domain representation of $x[n]$ which provides information about the amplitude and phase of each frequency component in the input sequence.

The *Inverse Discrete Fourier Transform (IDFT)* is given by:

$$x[n] = \frac{1}{N}\sum_{k=0}^{N-1} X[k]e^{i\frac{2\pi}{N}kn} \tag{1.4}$$

where $x[n]$ is the sample of the time-discrete signal, $X[k]$ is the k-th frequency component in the

frequency domain (amount of frequency k in the signal), and N is the length of the sequence. The result of the IDFT is the reconstructed time domain sequence. The DFT transforms a discrete-time domain sequence into its frequency domain representation, and the IDFT performs the inverse transformation to recover the original sequence.

As it was pointed, the output of the DFT is a frequency domain representation $X[k]$ of a discrete time signal $x[n]$ of length N, here $X[k]$ represents the complex-valued DFT coefficients for k frequencies ranging from 0 to $N-1$, i.e., $X[k] = X_{Re}[k] + iX_{Img}[k] = |X[k]|\cos(\phi(k)) + i|X[k]|\sin(\phi(k)) = |X[k]|e^{i\phi(k)} \forall k$ (where $|X[k]|, \phi(k) = arg(X[k])$ are the magnitude and phase spectra of $X[k]$). For real-valued signals $x[n]$, such as the ones we work with, positive and negative frequency components are complex conjugates of each other, such that N unique units of information are retained. The shortcoming of this is that, since spectra are complex-valued vectors, it is difficult to visualize them as such. A solution to this is to compute and plot the *Magnitude Spectrum* $|X[k]|$ which is obtained by taking the magnitude of the complex-valued DFT coefficients. It represents the distribution of the magnitudes of the frequency components in the signal. However, more generally the *Power Spectrum* $|X[k]|^2$ is computed, which is obtained by taking the squared magnitude of the complex-valued DFT coefficients. It represents the distribution of power or energy of the signal across different frequencies. The main advantage of the power spectrum over the magnitude spectrum is that it provides a measure of the signal's energy content at each frequency. However, due to the large differences in the range of different frequencies, these representations do not easily show relevant information. The *log-Spectrum* $10\log_{10}(|X[k]|^2) = 20\log_{10}(|X[k]|)$ is the most common visualization of spectra since it gives the visualization in decibels and because of this, as discussed previously, due to its logarithmic scale, small changes in power can be represented as meaningful differences in decibel-scaled values.

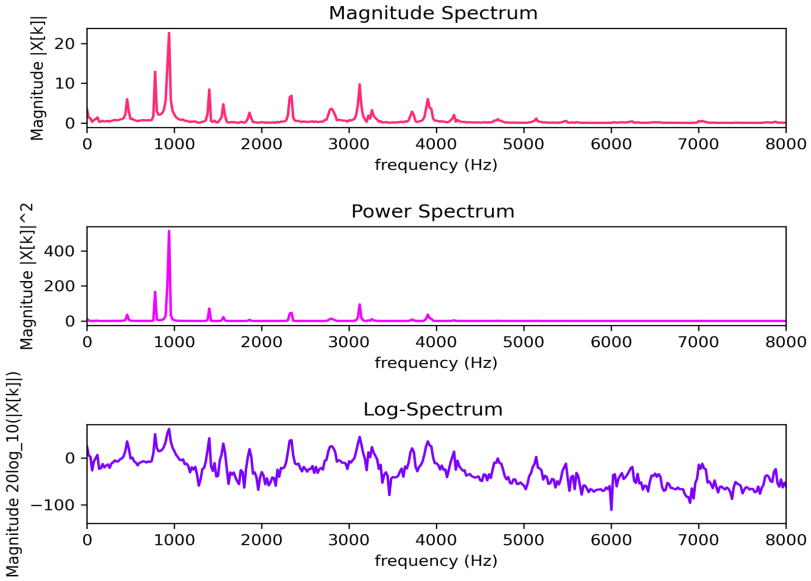

Figure 1.4: Magnitude Spectrum, Power Spectrum and Log-Spectrum plots of a sound signal.

The main limitation of these frequency spectra representations (magnitude spectrum, power

spectrum and log-spectrum) is their assumption of stationarity, for instance, the log-spectrum provides an average representation of the power or energy (decibel scaled) distribution across different frequencies for the entire duration of the signal. It assumes that the statistical properties of the signal, such as its power content, remain relatively constant over time. Similar is the case of the magnitude spectrum and power spectrum. However, as we have discussed, speech signals are time-variant. The solution to this is to break down the signal into short overlapping frames and apply a windowing function to them to focus on signal properties at a particular point in time, as we have discussed at the beginning of this section (framing and windowing). By windowing and computing the DFT of each frame, we obtain the *Short-time Fourier Transform* (STFT) of the signal. For an input discrete-time signal $\{x[n]\}$ of length L split in M frames each of size N (samples) and a window function-defined weights $\{w[n]\}$ the STFT is defined as

$$STFT\{x[n]\} = X[k,m] = \sum_{n=0}^{N-1} x[n + m \times s] \cdot w[n] e^{-i\frac{2\pi}{N}kn} \tag{1.5}$$

here $k \in \{1, \cdots, K-1\}$ represents the frequency bin index, $m \in \{1, \cdots, M-1\}$ represents the time frame index which determines the starting position of the windowed segment within the input signal $\{x[n]\}$ and s is the frame-step/hop-length/stride-length. The number of time frames can be calculated as $M = \lfloor \frac{L-N}{frame\_step} \rfloor + 1$ and the number of frequency bins K is typically equal to $N/2 + 1$ due to the Nyquist frequency limit. However, it can be adjusted based on specific requirements. By varying the value of m, the STFT allows to analyze the frequency content of a signal at different time segments by windowing each frame and computing the DFT on each windowed segment, therefore for different sections of the signal their time-varying frequency content can be analyzed at different time instants. This is in contrast to the DFT, which provides a stationary representation of the entire signal without any time-varying information. The output of the STFT $\forall k \forall m$ is a complex-valued matrix $X_{K \times M}$ of dimensions $K \times M$.

$X[k,m]$ denotes the complex-valued STFT coefficient at time frame m and frequency bin k. Since the elements of the STFT matrix $X_{K \times M}$ are complex-valued, it is difficult to visualize them as such. By computing the power spectrum of all $X[k,m]$ of the STFT matrix (i.e. $|X[k,m]|^2$) we obtain a matrix S_{pow} of size $K \times M$ known as the *Power Spectrogram* where each k-th m-th element, $S_{pow}[k,m] = |X[k,m]|^2$, is real-valued. However, to better visualize the power distribution across the frequencies the decibel-valued logarithmic scale the *log-Spectrogram* $S_{K \times M}$, or simply *Spectrogram*, is commonly used to visualize the STFT matrix, where each element of the matrix $S[k,m]$ is computed as $S[k,m] = 10\log_{10}(|X[k,m]|^2) = 20\log_{10}(|X[k,m]|)$.

These spectrograms (power and log) are visualized as heat-maps (see Figure 1.6) where the x-axis represents time, the y-axis represents frequency, and the color or intensity represents the power of each frequency component at a specific time. This provides a meaningful visualization of how the frequency content of the signal changes over time.

Mel-Spectrogram

The shortcoming of the spectrogram is that, while it provides a perceptually-relevant amplitude representation by using decibel scale, it does not provide a perceptually-relevant representation of frequency. As discussed previously, human perception of frequency (pitch) is not linear. To address this, the *Mel-Spectrogram* is introduced as a solution. The Mel-Spectrogram incorporates the concept of the Mel scale, which, as discussed previously, is nonlinear and closely aligned with human pitch perception. It utilizes a set of filters known as *Mel filter banks* (A *filter bank* is a set of filters used to extract specific frequency components or bands from a signal or signal representation. *bands* refers to distinct frequency ranges or intervals that are targeted or extracted by the filters. Each band represents a specific portion of the overall frequency spectrum). These filter banks are designed to mimic the frequency selectivity of the human auditory system. Each filter bank corresponds to a specific Mel frequency and acts as a band-pass filter that emphasizes certain frequency ranges.

The filter bank is created by building overlapping triangular Mel filters over the frequency (Hz) spectrum. The spacing between the filter banks is determined by the Mel scale, the triangle-centers are at the frequencies corresponding to equal distance steps on the Mel scale and the number of Mel filter banks is a parameter referred to as *Mel bands*. The values of the triangular filter at each frequency bin index k are computed based on the distance of that index from the left, center, and right edges of the triangular filter, these values increase linearly from 0 at the left edge to 1 at the center, and then decrease linearly to 0 at the right edge. This creates a set of overlapping triangular filters that ensures that the Mel-Spectrogram gives more importance to frequency regions that are perceptually more significant, as determined by the properties of human pitch perception.

The Mel filter banks are used to construct the corresponding *Mel Filter Banks Matrix* $H_{n\_mels \times K}$. Each row of H corresponds to a Mel filter, and each column corresponds to a frequency bin value that matches a frequency bin value from the power-spectrogram S. The elements $H[i, j]$ are the weights of each frequency bin for each Mel filter, they determine the contribution of each frequency bin to the response of the respective Mel filter. The i-th row in the matrix H corresponds to a specific Mel filter $i \in \{1, \cdots, n\_mels\}$, each element of that row $H[i, j], j \in \{1, \cdots, K\}$ contains the weight values assigned to a specific frequency bin $j \in \{1, \cdots, K\}$. Higher weight values indicate that the corresponding frequency bin will have a stronger influence on the output of the Mel filter, while lower weight values indicate a lesser influence.

The steps to compute the Mel filter banks and the Mel filter banks matrix H are:

- For the minimum and maximum frequencies $f\_min = 0Hz$, $f\_max = sr/2$ of S compute the minimum and maximum Mels mel_min, mel_max using the Mel scale transformation formula: $f_{mel} = 2595 \log_{10}(1 + \frac{f_{Hz}}{700}) = 1127 \ln(1 + \frac{f_{Hz}}{700})$
- Obtain the set of center *values* of the Mel filter banks by selecting n_mels (Mel bands) points from the segment (mel_min, mel_max) such that the resulting set of $n\_mels + 2$

points (including mel_min and mel_max) are equally spaced consecutive segments

- Obtain the *values* of the center Mels in the frequency (Hz) scale by transforming the set of center Mels points from the previous step to the Hz scale using the Mel scale reverse transformation formula: $f_{Hz} = 700(10^{\frac{f_{mel}}{2595}} - 1) = 700(e^{\frac{f_{mel}}{1127}} - 1)$. Let us denote these set of points as f_center_hz. These points represent the placement of the Mel filters in the frequency domain, however they are not scaled to the frequency bins basis.

- Compute the *indexes* of the frequency bins corresponding to the Mel filter bank, this is done by multiplying the points from f_center_hz by the total number of frequency bins K to scale the center frequencies of the Mel filters to align them with the frequency range of the frequency bins. Then, this product is divided by the sampling rate of the signal sr to normalize the frequencies with respect to it (sampling rate): $f\_index = \lfloor K \cdot f\_center\_hz/sr \rfloor$. This formula determines the placement (index) of each Mel filter in the frequency domain.

- Compute the triangular-shaped filters and construct the Mel filter banks matrix $H_{n\_mels \times K}$: The computation of the n_mels filter banks and construction of the Mel filter banks matrix H are performed simultaneously. This is because each row of the Mel filter banks matrix correspond to a Mel filter bank, and as the filter banks are computed, their weights are assigned to the corresponding rows of the matrix. For each Mel frequency point, we create a triangular-shaped filter centered at that frequency. The width of the triangular filter is determined by the neighboring Mel frequencies. The triangular-shaped filter is defined by three points: the frequencies where the filter starts, peaks, and ends. The peak frequency is the center frequency, and the start and end frequencies are determined based on the neighboring Mel frequencies. To compute the filter weights for each frequency bin, we calculate the overlap or intersection between the triangular-shaped filter and the frequency bin. This is typically done by evaluating the relative distances between the frequencies of the triangular filter and the frequency bins. The weights are calculated as the proportion of overlap or intersection between the triangular filter and each frequency bin. The next steps illustrate this process:

  For each Mel filter i from the *range* $[1, n\_mels]$:

  - From the previously computed set of indexes f_index retrieve the indexes of the left edge, center, and right edge of the triangular filter corresponding to Mel filter i: $filter\_left = f\_index[i-1]$, $filter\_center = f\_index[i]$, and $filter\_right = f\_index[i+1]$ (here $i-1$ is used to align with programming languages indexing that usually starts from 0).
  - Within the range $[filter\_left, filter\_center)$, the value of the triangular filter at each frequency bin index j is determined by the distance of j from $filter\_left$ relative to the total distance between $filter\_left$ and $filter\_center$. Therefore:

For each frequency bin index j within the range $[filter\_left, filter\_center)$: assign the value $\frac{j - filter\_left}{filter\_center - filter\_left}$ to the matrix element $H[i-1, j]$.

– Within the range $[filter\_center, filter\_right)$, the value of the triangular filter at each frequency bin index j is determined by the distance of j from $filter\_right$ relative to the total distance between $filter\_right$ and $filter\_center$. Therefore:

For each frequency bin index j within the range $[filter\_center, filter\_right)$: assign the value $\frac{filter\_right - j}{filter\_right - filter\_center}$ to the element $H[i-1, j]$.

The output of this algorithm is a Mel filter banks matrix H of dimensions $n\_mels \times K$.

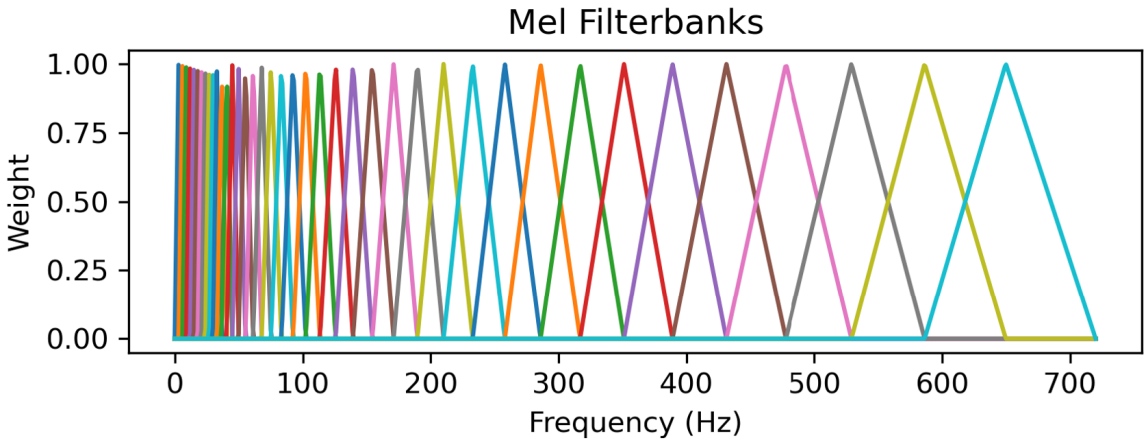

Figure 1.5: Mel Filterbanks.

To construct the Mel-Spectrogram, the signal is divided into short overlapping frames, the STFT of each windowed frame is computed to obtain the STFT matrix $X_{K \times M}$. The power-spectrogram matrix is computed $S_{K \times M} : \forall(k, m) S[k, m] = |X[k, m]|^2$. Next, following the steps described above, the corresponding Mel filter banks matrix $H_{n\_mels \times K}$ is constructed. The *Mel-Spectrogram Matrix* $M_{mel\_bands \times M}$ is computed by transposing the result from the matrix product of H with S:

$$M = (H \times S)^T \tag{1.6}$$

The *log Mel-Spectrogram Matrix* is computed by decibel scaling the "regular" Mel-spectrogram Matrix:

$$M = 10 \log_{10}((H \times S)^T) \tag{1.7}$$

By applying the Mel filter banks matrix H weights to the spectrogram S, the Mel filter banks shape the power distribution across frequency bins, emphasizing certain frequency regions and de-emphasizing others. This process reflects the non-linear characteristics of human auditory perception.

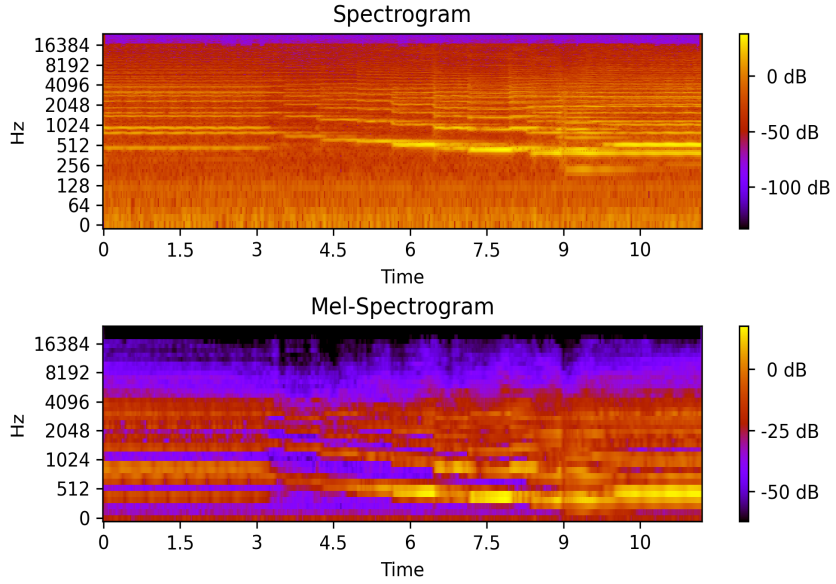

Figure 1.6: Spectrogram and Mel-Spectrogram of a signal.

With the study of the Mel-spectrogram we conclude our review of the fundamentals of digital-signal/audio-processing necessary to understand the subsequent sections of this work.

2 Fundamental Deep Learning Architectures for Sequential Data

Here we delve into the fundamental neural networks architectures specifically designed for processing sequential data. They serve as building blocks for the more complex architectures that we will discuss in subsequent sections.

2.1 RNN - Recursive Neural Networks

Recursive Neural Networks (RNN) [1] are a type of network architecture that is specifically designed to process input and output sequences of varying lengths. The key aspect of their design lies in their ability to capture and retain temporal dependencies from previous values from the input stream. Unlike conventional neural networks that are not suited for sequential data, RNNs ensure that such dependencies are not disregarded or lost in subsequent steps of the input processing. This is achieved through recurrent connections that allow exchange of information from previous steps to the current step, allowing the network to maintain memory or context of the input sequence. To implement this recursive approach, RNNs incorporate one (in the case of the "vanilla" RNN) or several hidden layers for which at each step of the input sequence, the hidden layer's output is stored and then fed back into the network along with the subsequent element of the input sequence. By adopting this recursive approach, RNNs can leverage past context when making predictions in subsequent steps. This property makes RNNs particularly useful for task such as natural language processing, time series analysis, and, the

one we are interested in at this work, speech processing.

RNNs retain information through an *internal state vector* $\mathbf{h}_t$, which is updated as the sequence is processed. At every time step t, an input vector $\mathbf{x}_t$ along with the output internal state vector from the previous hidden layer $\mathbf{h}_{t-1}$, are fed into the RNN's t-th hidden layer, the internal state at step t is computed based on this input, which will be then passed to the subsequent $t+1$-th hidden layer along with the corresponding input vector $\mathbf{x}_{t+1}$.

This process enables the RNN to capture the dependencies between consecutive elements in the input sequence. The internal state vector acts as a memory mechanism, it carries information about past inputs and is updated and passed along to subsequent time steps, allowing the network to maintain a notion of context and temporal dependencies throughout the sequence. This vector can be understood as a "representation of what the network has seen so far". Because of this the RNN produces an output that is influenced by both the current input and the historical context captured in its internal state.

The "Vanilla" RNN

Now we proceed to describe the vanilla RNN design:

Consider an input sequence of T vectors $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \cdots, \mathbf{x}_{t-1}, \mathbf{x}_t, \mathbf{x}_{t+1}, \cdots, \mathbf{x}_T) : \mathbf{x}_i \in \mathbb{R}^m$. The vanilla RNN has a single hidden layer, that is, there is only *one hidden layer that is shared across all time steps*. This means that *there is only a single set of weights that is applied at each time step*, which results in a reduction in the number of parameters. However, even though there's a single hidden layer, the internal state vector $\mathbf{h}_t$ is modified at each step t. So, while there is a single hidden layer in the RNN architecture, it effectively operates as if there were multiple hidden layers, with each hidden state $\mathbf{h}_t$ representing the "hidden layer" at time step t. At each step $t \in \{1, \cdots, T\}$, the RNN processes the input vector $\mathbf{x}_t$ along with the previous hidden state $\mathbf{h}_{t-1}$ to compute the new hidden state $\mathbf{h}_t$. This computation is accomplished through a recurrent relation employing a non-linear activation function $f_{\mathbf{h}}(\mathbf{h}_{t-1}, \mathbf{x}_t) = \phi_{\mathbf{h}}(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h)$ (e.g., sigmoid, tanh), typically and in the case of the vanilla RNN $\phi_{\mathbf{h}}() = \tanh()$ is used. In the vanilla RNN, the recurrence relation for computing the internal state vector at step t, $\mathbf{h}_t \in \mathbb{R}^n$ is given by:

$$\mathbf{h}_t = \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h) \tag{2.1}$$

where

- $\mathbf{x}_t \in \mathbb{R}^m$ is the vector from the input sequence at step/time $t \in \{1, \cdots, T\}$
- $\mathbf{h}_{t-1} \in \mathbb{R}^n$ is the internal state vector from the previous step/time-step $t-1$. At step/time-step $t = 0$ the initialization internal state vector is a zero vector $\mathbf{h}_0 = \mathbf{0} \in \mathbb{R}^n$
- $\mathbf{b}_h \in \mathbb{R}^n$ is the bias vector.
- $\mathbf{W}_{xh} \in \mathbb{R}^{n \times m}$ is the weight matrix for the input connections (from input to hidden layer)
- $\mathbf{W}_{hh} \in \mathbb{R}^{n \times n}$ is the weight matrix for the recurrent connections (hidden layer to hidden layer)

The learnable parameters are $\mathbf{W}_{hh}$, $\mathbf{W}_{xh}$ and $\mathbf{b}_h$. These parameters are shared across all time steps.

The prediction at step t is a function of the current internal hidden state $\mathbf{h}_t$, and is computed using a non-linear activation function $f_{\mathbf{o}}(\mathbf{h}_t) = \phi_{\mathbf{o}}(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y)$ over a linear matrix projection on top of the hidden state, in the vanilla RNN the activation function is the *softmax*:

$$\hat{\mathbf{y}}_t = softmax(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y) \tag{2.2}$$

where $\mathbf{W}_{hy} \in \mathbb{R}^{n \times n}$ represents the weight matrix for the projection from the hidden layer to the output, $\mathbf{b}_y \in \mathbb{R}^n$ is the bias vector and $\hat{\mathbf{y}}_t$ is the output probability distribution at step/time-step t. $\mathbf{W}_{hy}$ and $\mathbf{b}_y$ are learnable parameters, and are shared across all steps/time-steps. The softmax function ensures that the predicted outputs are normalized probabilities, allowing them to represent the probabilities of different classes or categories.

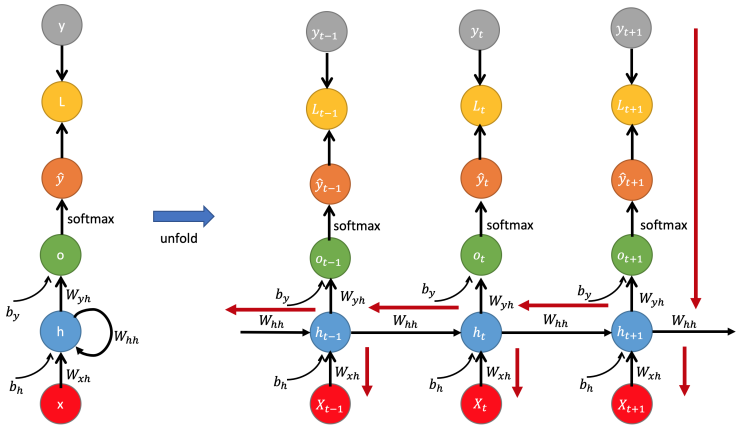

Figure 2.1: single hidden layer RNN many-to-many diagram.

On the left diagram the recurrent aspect of the network is shown through the loop in the hidden layer. The diagram on the right is the "unfolded" equivalent to the one the left, it shows how each input vector is fed to the network at the corresponding step. Is worth emphasizing on the fact that the network has a single hidden layer, on the "unfolded" version of the diagram this is apparent due to the parameter sharing between computations of the hidden state. The red arrows represent the backpropagation-through-time path.

The recurrent nature of the network becomes apparent by expanding the hidden state vector function for a setp $t = n > 1$:

$$\begin{aligned}
\mathbf{h}_n &= f_{\mathbf{h}}(\mathbf{h}_{n-1}, \mathbf{x}_n) \\
&= f_{\mathbf{h}}(f_{\mathbf{h}}(\mathbf{h}_{n-2}, \mathbf{x}_{n-1}), \mathbf{x}_n) \\
&= f_{\mathbf{h}}(f_{\mathbf{h}}(f_{\mathbf{h}}(\mathbf{h}_{n-3}, \mathbf{x}_{n-2}), \mathbf{x}_{n-1}), \mathbf{x}_n) \\
&\vdots \\
&= f_{\mathbf{h}}(f_{\mathbf{h}}(f_{\mathbf{h}}(\cdots(f_{\mathbf{h}}(f_{\mathbf{h}}(f_{\mathbf{h}}(\mathbf{h}_0, \mathbf{x}_1), \mathbf{x}_2), \mathbf{x}_3), \cdots), \mathbf{x}_{n-2}), \mathbf{x}_{n-1}), \mathbf{x}_n) \\
&= \mathbf{F}_{\mathbf{h}}(\mathbf{h}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \cdots, \mathbf{x}_{n-3}, \mathbf{x}_{n-2}, \mathbf{x}_{n-1}, \mathbf{x}_n)
\end{aligned} \tag{2.3}$$

It follows that, the internal state function calls itself recursively until reaching the first input $\mathbf{x}_1$ and the initial hidden state $\mathbf{h}_0$. The function uses the same set of parameters $\mathbf{W}_{hh}$, $\mathbf{W}_{xh}$ and

$\mathbf{b}_h$ at every step.

So, for an input sequence of T vectors, the vanilla RNN has a single hidden layer and will have T hidden internal states, one for each step/time-step. Each hidden state $\mathbf{h}_t$ is computed using the shared set of weights of the network, effectively reducing the number of learnable parameters across variable length input sequences. The vanilla RNN will produce T outputs, an output $\hat{\mathbf{y}}_t$ at every step t of the sequence. This type of RNN is known as *"many-to-many"*. There are different types of RNNs based on the length of its input T_x and output T_y:

- *One-to-one* $T_x = T_y = 1$ this corresponds to the traditional neural network
- *One-to-many* $T_x = 1, T_y > 1$
- *Many-to-one* $T_x > 1, T_y = 1$
- *Many-to-many* $T_x = T_y$ the one described above, or $T_x \neq T_y : T_x > 1 \wedge T_y > 1$

As mentioned previously, the vanilla RNN has a single hidden layer, however, more complex implementations can have several hidden layers, each having its own internal vector state and shared set of weights. The type of RNN to be implemented and the number of hidden layers depends on the purposes of the task that is set to solve.

The main advantages of RNNs are:

- RNNs process input sequences of variable length
- The model size does not increase for longer input sequences
- Weights are shared for every time-step of the input, decreasing the number of learnable parameters
- In theory, the computation for step t uses information from the previous steps

The major disadvantage of RNNs is that: In practice, it is difficult to access information from several steps back due to phenomena knows as "vanishing" and "exploding" gradients, which we will discuss in the subsequent section.

2.2 Backpropagation Through Time for RNN and the Vanishing and Exploding Gradients Problem

RNNs are design to extract features from long sequential data. However, training them can become challenging due to phenomena known as Vanishing Gradients and Exploding Gradients [2]. These may occur during the training process, particularly during the backpropagation stages. To understand this issue, we will first discuss how backpropagation works in RNNs. The backpropagation process for RNNs differs from regular neural networks and is referred to as *Backpropagation Through Time (BPTT)*. The difference lies in how recurrent connections

and the temporal dimension of the data are handled. In a regular feed-forward neural network, backpropagation involves propagating gradients from the output layer back to the preceding layers, updating the weights and biases based on the computed gradients. The computations are performed layer by layer, and there are no recurrent connections or time dependencies involved. In contrast, BPTT considers the temporal dependencies in sequential data. During BPTT, the gradients are propagated backward through the "unfolded" recurrent connections of the RNN *from the output layer to the initial step*. The gradients are accumulated and used to to update the weights and biases of the recurrent connections *at each time step*. This allows the network to capture and learn from the temporal dependencies present in the sequential data.

Consider the loss function

$$L(\hat{\mathbf{y}}, \mathbf{y}) = \sum_{t=1}^{T} L_t(\hat{\mathbf{y}}_t, \mathbf{y}_t)$$

we want to compute the gradients of the loss function L with respect to the network parameters, we are specially interested in the behavior of the gradients of L with respect to $\mathbf{W}_{hh}$ and $\mathbf{W}_{xh}$ since their computation results in a recurrent product (which we will demonstrate bellow) as the error is backpropagated through each time-step. First, let's express the more straightforward gradients of L: those are with respect to $\mathbf{W}_{hy}$ and $\mathbf{b}_y$. Recall that (due to the design of RNNs) the weights matrix $\mathbf{W}_{hy}$ is shared across all the time sequence. Therefore, at each time step we can partially differentiate L with respect to $\mathbf{W}_{hy}$ and sum it all together:

$$\frac{\partial L}{\partial \mathbf{W}_{hy}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial \mathbf{W}_{hy}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{W}_{hy}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{o}_t} \frac{\partial \mathbf{o}_t}{\partial \mathbf{W}_{hy}} \tag{2.4}$$

where

$$\mathbf{o}_t = \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y \tag{2.5}$$

Similarly, for $\frac{\partial L}{\partial \mathbf{b}_y}$:

$$\frac{\partial L}{\partial \mathbf{b}_y} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{o}_t} \frac{\partial \mathbf{o}_t}{\partial \mathbf{b}_y} \tag{2.6}$$

To compute the gradients of L with respect to $\mathbf{W}_{hh}$ and $\mathbf{W}_{xh}$ it must be considered that each backward step use the same parameters $\mathbf{W}_{hh}$, $\mathbf{W}_{xh}$ and due to the recurrent nature of the hidden state $\mathbf{h}_t$, that is, the fact that at step t the hidden state $\mathbf{h}_t$ depend from the previous hidden state step $\mathbf{h}_{t-1}$ which in turn depends on the previous one $\mathbf{h}_{t-2}$ and so on, as shown in eq.(2.3). Because of this, on the gradients of $\frac{\partial L_t}{\partial W_{hh}}$ and $\frac{\partial L_t}{\partial W_{xh}}$ a recurrent product appears, as further shown bellow. Consider at time-step t:

$$\frac{\partial L_t}{\partial \mathbf{W}_{hh}} = \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{W}_{hh}} = \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{o}_t} \frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}} \tag{2.7}$$

eq.(2.7) is only considering a single step t. However, the hidden state $\mathbf{h}_t$ also partially depends on the previous step $t-1$ hidden state $\mathbf{h}_{t-1}$ according to the recurrent formulation from eq.(2.1),

which in turn also depends on $\mathbf{W}_{hh}$. Thus: $\frac{\partial L_t}{\partial \mathbf{W}_{hh}} = \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{o}_t}\frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}} + \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{o}_t}\frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}\frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{W}_{hh}}$, but, as it was shown in eq.(2.3), the hidden state at step t partially depends not only from the hidden state from step $t-1$ but also from the one from step $t-2$ who also depends on $\mathbf{W}_{hh}$, thus considering this and the chain rule of partial differentiation for composed functions: $\frac{\partial L_t}{\partial \mathbf{W}_{hh}} = \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{o}_t}\frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}} + \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{o}_t}\frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}\frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{W}_{hh}} + \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{o}_t}\frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}\frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{h}_{t-2}}\frac{\partial \mathbf{h}_{t-2}}{\partial \mathbf{W}_{hh}}$. Consequently, the hidden state from step $t-2$ partially depends from the one from step $t-3$ who also depends on $\mathbf{W}_{hh}$, considering this: $\frac{\partial L_t}{\partial \mathbf{W}_{hh}} = \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{o}_t}\frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}} + \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{o}_t}\frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}\frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{W}_{hh}} + \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{o}_t}\frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}\frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{h}_{t-2}}\frac{\partial \mathbf{h}_{t-2}}{\partial \mathbf{W}_{hh}} + \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{o}_t}\frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}\frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{h}_{t-2}}\frac{\partial \mathbf{h}_{t-2}}{\partial \mathbf{h}_{t-3}}\frac{\partial \mathbf{h}_{t-3}}{\partial \mathbf{W}_{hh}}$. And similarly for all the $t-1$ preceding steps. As a result, at time-step t we can compactly re-write eq.(2.7) so that it properly backpropagates the error to *all* previous steps as:

$$\frac{\partial L_t}{\partial \mathbf{W}_{hh}} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{o}_t}\frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}\frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_{hh}} \tag{2.8}$$

where

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} = \prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}} = \frac{\partial \mathbf{h}_{k+1}}{\partial \mathbf{h}_k}\frac{\partial \mathbf{h}_{k+2}}{\partial \mathbf{h}_{k+1}}\frac{\partial \mathbf{h}_{k+3}}{\partial \mathbf{h}_{k+2}} \cdots \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \tag{2.9}$$

refers to the partial derivative of $\mathbf{h}_t$ with respect to *all* previous k time-steps, that is, the chain differentiation of over all internal states within the $[k, t]$ time interval.

In a similar way, at time-step t the partial derivative of L with respect to $\mathbf{W}_{xh}$ is computed as:

$$\frac{\partial L_t}{\partial \mathbf{W}_{xh}} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{o}_t}\frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}\frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_{xh}} \tag{2.10}$$

where $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}$ is eq.(2.9)

Now, aggregating the gradients from eqs. (2.8) and (2.10) over the whole T time-steps from the input sequence $\mathbf{X} = (\mathbf{x}_1, \cdots, \mathbf{x}_{t-1}, \mathbf{x}_t, \mathbf{x}_{t+1}, \cdots, \mathbf{x}_T)$ yields the gradients:

$$\frac{\partial L}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T}\sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{o}_t}\frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}\frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_{hh}} \tag{2.11}$$

$$\frac{\partial L}{\partial \mathbf{W}_{xh}} = \sum_{t=1}^{T}\sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{o}_t}\frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}\frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_{xh}} \tag{2.12}$$

where $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}$ is eq.(2.9).

From the gradients in eqs.(2.11), (2.12) follows that, there are two factors that affect the magnitude of the gradients - the weights and the hidden vectors functions (or more precisely, their derivatives) that the gradients passes through.

Let us examine how the recurrent partial derivative $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}$ (from eq.(2.9)) is computed for a single step $t-1 \rightarrow t$:

$$\mathbf{h}_t = \begin{bmatrix} h_{t_1} \\ h_{t_2} \\ h_{t_3} \\ \vdots \\ h_{t_n} \end{bmatrix} := \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h) = \tanh(\gamma_t) = \begin{bmatrix} \tanh(\gamma_{t_1}) \\ \tanh(\gamma_{t_2}) \\ \tanh(\gamma_{t_3}) \\ \vdots \\ \tanh(\gamma_{t_n)} \end{bmatrix}$$

Here we used the notation $\gamma_t \in \mathbb{R}^n$ for the vector argument of $\mathbf{h}_t$, i.e., $\gamma_t = \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h$. Then, the scalar elements from the hidden vector at step t are computed as

$$h_{t_i} = \tanh(\gamma_{t_i}) = \tanh(\sum_{j=1}^{n} w_{hh_{ij}} h_{t-1_j} + \sum_{j=1}^{n} w_{xh_{ij}} x_{t_j} + b_{h_i}) \forall h_{t_i} \in \mathbf{h}_t$$

from there, we can express the element-wise gradient computation as:

$$\frac{\partial h_{t_i}}{\partial h_{t-1_j}} = \frac{\partial h_{t_i}}{\partial \gamma_{t_i}} \frac{\partial \gamma_{t_i}}{\partial h_{t-1_j}} = \tanh'(\gamma_{t_i}) w_{hh_{ij}} = w_{hh_{ij}} \tanh'(\gamma_{t_i}) \forall i, j \in \{1, \cdots, n\}$$

then, the Jacobian matrix $\partial \mathbf{h}_t / \partial \mathbf{h}_{t-1} \in \mathbb{R}^{n \times n}$ is:

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} = \begin{bmatrix} \frac{\partial h_{t_1}}{\partial h_{t-1_1}} & \cdots & \frac{\partial h_{t_1}}{\partial h_{t-1_n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_{t_n}}{\partial h_{t-1_1}} & \cdots & \frac{\partial h_{t_n}}{\partial h_{t-1_n}} \end{bmatrix} = \begin{bmatrix} w_{hh_{11}} \tanh'(\gamma_{t_1}) & \cdots & w_{hh_{1n}} \tanh'(\gamma_{t_1}) \\ \vdots & \ddots & \vdots \\ w_{hh_{n1}} \tanh'(\gamma_{t_n}) & \cdots & w_{hh_{nn}} \tanh'(\gamma_{t_n}) \end{bmatrix}$$

$$= \begin{bmatrix} \tanh'(\gamma_{t_1}) & \cdots & 0 \\ \mathbf{0} & \tanh'(\gamma_{t_i}) & \mathbf{0} \\ 0 & \cdots & \tanh'(\gamma_{t_n}) \end{bmatrix} \times \begin{bmatrix} w_{hh_{11}} & \cdots & w_{hh_{1n}} \\ \vdots & w_{hh_{ii}} & \vdots \\ w_{hh_{n1}} & \cdots & w_{hh_{nn}} \end{bmatrix} = diag[\tanh'(\gamma_t)] \times \mathbf{W}_{hh}$$

$$= \mathbf{W}_{hh}^T \times diag[\tanh'(\gamma_t)] = \mathbf{W}_{hh}^T \times diag[\tanh'(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h)] \tag{2.13}$$

Thus, if we propagate through $[k, t]$ steps, the gradient from eq.(2.9) will be:

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} = \prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}} = \prod_{j=k+1}^{t} \left(\mathbf{W}_{hh}^T \times diag[\tanh'(\gamma_j)] \right) \tag{2.14}$$

The intuition of the *Vanishing Gradients Problem* [2] is as follows: if the magnitudes of the recurrent products from eq.(2.9) are less than 1, then the gradients will "vanish" as $t \to \infty$:

$$\left\| \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}} \right\| < 1 \Rightarrow \prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}} \to 0 \text{ as } t \to \infty \Longrightarrow \frac{\partial L}{\partial \mathbf{W}_{hh}} \to 0 \wedge \frac{\partial L}{\partial \mathbf{W}_{xh}} \to 0$$

here $||\cdot||$ is the matrix 2-norm/spectral norm.

When the gradients $\partial L/\partial \mathbf{W}_{hh}$, $\partial L/\partial \mathbf{W}_{xh}$ tend to zero, it can go undetected while reducing the learning quality of the model for far-away steps of the input sequence. Due to this, is not known whether there is no dependency between steps t and $t+T$ in the input sequence, or whether is only that the dependency cannot be captured because of this issue.

The intuition of the *Exploding Gradients Problem* [2] is as follows: if the magnitudes of the recurrent products from eq.(2.9) are greater than 1, then the gradients will "explode" as $t \to \infty$:

$$\left\|\frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| > 1 \Rightarrow \prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}} \to \infty \text{ as } t \to \infty \Longrightarrow \frac{\partial L}{\partial \mathbf{W}_{hh}} \to \infty \wedge \frac{\partial L}{\partial \mathbf{W}_{xh}} \to \infty$$

here $||\cdot||$ is the matrix *2*-norm/spectral norm.

When the gradients $\partial L/\partial \mathbf{W}_{hh}$, $\partial L/\partial \mathbf{W}_{xh}$ grows extremely large, it causes an overflow, this is more easily detectable during training than with the vanishing gradients.

Let us formalize the conditions for vanishing and exploding gradients:

Let β_h be the upper bound of the matrix norm of the derivative of the non-linearity hidden state function ϕ_h. It is worth pointing out that, indeed, the upper bound β_h exists, since, if $\phi_h() = \tanh()$ then $-1 \leq \tanh(x) \leq 1$ and $0 \leq \tanh'(x) = 1 - \tanh^2(x) \leq 1$. Or, if $\phi_h() = \sigma()$ then $0 \leq \sigma(x) \leq 1$ and $0 \leq \sigma'(x) = \sigma(x)(1-\sigma(x)) \leq 1/4$:

$$\exists \beta_h \in \mathbb{R} : \left\|diag[\phi'_h(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h)]\right\| = \left\|diag[\phi'_h(\gamma_t)]\right\| \leq \beta_h \forall t \tag{2.15}$$

Recall that, for a (not necessarily square) matrix (real-valued or complex-valued) $\mathbf{A}$ the 2-norm/spectral norm $||\cdot||$ can be computed as $||\mathbf{A}|| = \sqrt{\lambda_{max}(\mathbf{A}^*\mathbf{A})} = \delta_{max}(\mathbf{A})$, here $\lambda_{max}(\mathbf{A}^*\mathbf{A})$ denotes the maximum eigen-value of $\mathbf{A}^*\mathbf{A}$, which, by definition is the maximum-singular value of $\mathbf{A}$, denoted by $\delta_{max}(\mathbf{A})$ (we use δ instead of σ to denote singular values since σ is already used to denote the sigmoid function).

Let us denote the maximum-singular value of $\mathbf{W}_{hh}$ as β_W, then,

$$\left\|\mathbf{W}_{hh}\right\| = \beta_W \tag{2.16}$$

$\beta_W \in \mathbb{R}$ since $\mathbf{W}_{hh}$ is a real-valued matrix and therefore $\mathbf{W}^*_{hh}\mathbf{W}_{hh} = \mathbf{W}^T_{hh}\mathbf{W}_{hh} \Rightarrow \delta(\mathbf{W}_{hh}) \in \mathbb{R}$, i.e., the singular values of $\mathbf{W}_{hh}$ are real non-negatives numbers.

From eqs. (2.13), (2.15) and (2.16) we obtain:

$$\left\|\frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| = \left\|\mathbf{W}^T_{hh} \times diag[\phi'_h(\gamma_j)]\right\| \leq \left\|\mathbf{W}^T_{hh}\right\| \left\|diag[\phi'_h(\gamma_j)]\right\| \leq \beta_W \beta_h \tag{2.17}$$

And from eqs. (2.14) and eq.(2.17) we obtain:

$$\left\|\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}\right\| = \left\|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| \leq (\beta_W \beta_h)^{t-k} \tag{2.18}$$

If $\beta_W < 1/\beta_h$ then $\beta_W \beta_h < \frac{1}{\beta_h}\beta_h = 1$ and, from there and from eq.(2.18):

$$\left\|\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}\right\| = \left\|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| \leq (\beta_W \beta_h)^{t-k} < 1 \tag{2.19}$$

therefore, $(\beta_W \beta_h)^{t-k} \to 0$ as $t \to \infty$ and consequently $\left\|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| \to 0$ exponentially fast, thus, resulting in vanishing gradients. Since the gradients become exponentially small as they propagate backward through time, as result RNNs struggle to effectively propagate information across long time steps, *leading to a loss of important context and difficullty in capturing long-term dependencies.*

Analogously, if $\beta_W > 1/\beta_h$ then $\beta_W \beta_h > \frac{1}{\beta_h}\beta_h = 1$ and, from there and from eq.(2.18):

$$\left\|\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}\right\| = \left\|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| \leq (\beta_W \beta_h)^{t-k} \wedge (\beta_W \beta_h)^{t-k} > 1 \implies (\beta_W \beta_h)^{t-k} \geq \left\|\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}\right\| = \left\|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| > 1$$

from there, and from the axioms of real numbers follows, that:

$$\exists \eta \in \mathbb{R}^+ : \forall j \left\|\frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| \geq \eta > 1 \implies (\beta_W \beta_h)^{t-k} \geq \left\|\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}\right\| = \left\|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| \geq \eta^{t-k} > 1 \tag{2.20}$$

therefore, $(\beta_W \beta_h)^{t-k} \to \infty \wedge \eta^{t-k} \to \infty$ as $t \to \infty$ and consequently $\left\|\prod_{j=k+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}\right\| \to \infty$ exponentially fast, thus, resulting in exploding gradients.

As it can be seen from the results above, the main problem causing gradients to vanish/explode is the product of the recursive derivatives. During training of the model is easily detectable if the gradients are exploding, due to the overflow (resulting in NaN values in numpy, for example). There are several methods proposed as solutions for exploding gradients, for example, *Gradient Clipping* [3] [4] which consists in setting a threshold value that if, during training, the gradients reach this threshold they are set back to a small number. This heuristic, however does not solve the vanishing gradients problem, one of the approaches that was proposed to overcome this issue is another type of recurrent network architecture known as *Long Short-Term Memory (LSTM)* which we will discuss next.

2.3 Long Short-Term Memory Recurrent Neural Networks (LSTM)

Long Short-Term Memory (LSTM) [5] is a type of recurrent neural network, whose architecture is designed to address the limitations of traditional RNNs in capturing and retaining long-term dependencies in sequential data that arises due to the vanishing gradient problem [2]. To address these limitations, LSTMs introduces specialized memory units called *cells* which are computational units responsible for storing and updating information over time. The cell maintains a memory state which allows relevant information to persist and flow through the network. It incorporates *gate* vectors that are designed to regulate the amount of information that enters and exists the cell, allowing the network to selectively retain or discard information at different time steps. The gates are implemented using sigmoid activations functions, which squashes the values between 0 and 1, enabling them to act as adaptive switches for information flow. As in

with RNNs, LSTMs have hidden states, they are not replaced by the cell state, the conceptual difference is that, the cell state stores and preserves *long-term* (of all previous events) information throughout the sequence, it acts as a storage that can selectively retain or discard long-term information, while the hidden state is a representation of the relevant *short-term* (immediately previous events) information extracted from previous inputs. That is, the cell state encodes an aggregation of data from *all* previous time-steps that have been processed, while the hidden state encodes a characterization of the *previous* time-step data.

The "Vanilla" LSTM cell

The "vanilla" LSTM cell is an individual memory unit that stores and manipulates information over time, is completely described by the following set of equations:

$$
\begin{aligned}
\mathbf{i}_t &= \sigma(\mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{W}_{xi}\mathbf{x}_t + \mathbf{b}_i) \\
\mathbf{f}_t &= \sigma(\mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{W}_{xf}\mathbf{x}_t + \mathbf{b}_f) \\
\mathbf{o}_t &= \sigma(\mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{W}_{xo}\mathbf{x}_t + \mathbf{b}_o) \\
\\
\tilde{\mathbf{c}}_t &= \tanh(\mathbf{W}_{hc}\mathbf{h}_{t-1} + \mathbf{W}_{xc}\mathbf{x}_t + \mathbf{b}_c) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \\
\\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)
\end{aligned}
\tag{2.21}
$$

Where $\mathbf{W}_{hi}, \mathbf{b}_i, \mathbf{W}_{hf}, \mathbf{b}_f, \mathbf{W}_{ho}, \mathbf{b}_o, \mathbf{W}_{hc}, \mathbf{b}_c$ are the learnable parameters which are *shared-across all time-steps*, and:
$\mathbf{x}_t \in \mathbf{X} = (\mathbf{x}_1, \cdots, \mathbf{x}_T) \wedge \mathbf{x}_t \in \mathbb{R}^m \forall t \in \{1, \cdots, T\}, \mathbf{h}_t \in \mathbb{R}^n$,
$\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho}, \mathbf{W}_{hc} \in \mathbb{R}^{n \times n}$,
$\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo}, \mathbf{W}_{xc} \in \mathbb{R}^{n \times m}$,
$\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o, \mathbf{b}_c \in \mathbb{R}^n$.

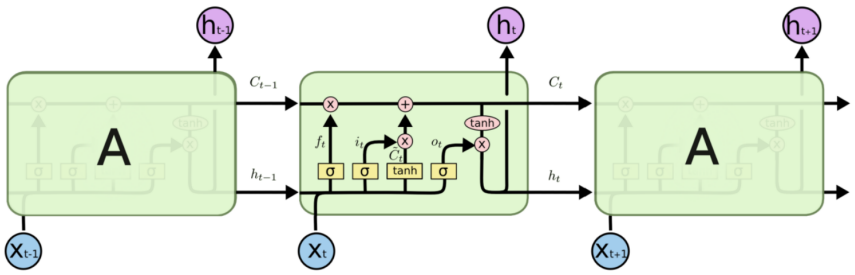

Figure 2.2: LSTM cell corpus.

LSTM cell connections from step $t-1$ to $t+1$.

- *Input/Read gate*: $\mathbf{i}_t = \sigma(\mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{W}_{xi}\mathbf{x}_t + \mathbf{b}_i)$
  Determines how much new information should be allowed into the cell state update by

gating the candidate update $\tilde{\mathbf{c}}_t$ (described further bellow).

- *Forget/Reset gate*: $\mathbf{f}_t = \sigma(\mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{W}_{xf}\mathbf{x}_t + \mathbf{b}_f)$
  Determines how much information from the previous cell state should be discarded or forgotten at the current time step. It gates the previous cell state $\mathbf{c}_{t-1}$ to control which parts of it should be retained or reset for the cell state update.

- *Output/Write gate*: $\mathbf{o}_t = \sigma(\mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{W}_{xo}\mathbf{x}_t + \mathbf{b}_o)$
  It determines which parts of the cell state should contribute to the hidden state and, consequently, to the output of the LSTM.

- *Candidate Update/Intermediate Cell State*: $\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_{hc}\mathbf{h}_{t-1} + \mathbf{W}_{xc}\mathbf{x}_t + \mathbf{b}_c)$
  It computes the candidate values that could potentially be added to the cell state. Since is gated by the input gate $\mathbf{i}_t$ during the cell state $\mathbf{c}_t$ update it allows the network to selectively update the cell state based on the current input and the context captured in the previous hidden state.

- *Cell State*: $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$
  Here $\odot$ is the element-wise multiplication between the corresponding vectors. The cell state $\mathbf{c}_t$ represents the long-term memory of the LSTM. It carries information across all the time steps and retains relevant information.

  The previous cell state $\mathbf{c}_{t-1}$ is gated by the forget gate $\mathbf{f}_t$ to determine the portion of the previous cell state that should be forgotten. The candidate update $\tilde{\mathbf{c}}_t$ is gated by the input gate $\mathbf{i}_t$ to determine the portion of the candidate update that should be added to the cell state. These two results are added to update the cell state $\mathbf{c}_t$.

- *Hidden State*: $\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$
  Here $\odot$ is the element-wise multiplication between the corresponding vectors.

  The hidden state represents the short-term memory of the LSTM. Is the output of the LSTM cell at a given time-step t.

  The activated cell state is gated by the output gate $\mathbf{o}_t$ to determine how much of it should be used for the hidden state. From this, follows that the hidden state vector $\mathbf{c}_t$, which is used to store long-time dependencies, influences the prediction at step t effectively using the long-term dependencies learned in it to influence the output of the network.

The prediction at step t is a function of the current internal hidden state $\mathbf{h}_t$, because of this the hidden state is considered as the output of the LSTM cell. The prediction is computed using a non-linear activation function $f_{\mathbf{y}}(\mathbf{h}_t) = \phi_{\mathbf{y}}(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y)$ over a linear matrix projection on top of the hidden state:

$$\hat{\mathbf{y}}_t = \phi_{\mathbf{o}}(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y) \tag{2.22}$$

where $\mathbf{W}_{hy} \in \mathbb{R}^{n \times n}$ represents the weight matrix for the projection from the hidden layer to the output, $\mathbf{b}_y \in \mathbb{R}^n$ is the bias vector and $\hat{\mathbf{y}}_t$ is the output probability distribution at step/time-step

t. $\mathbf{W}_{hy}$ and $\mathbf{b}_y$ are learnable parameters, and are shared across all time-steps.

In a manner similar to RNNs, LSTMs can be categorized based on the lengths of the input sequence and output, they can be classified as one-to-one, one-to-many, many-to-one or many-to-many. Also, LSTMs can stack several LSTMs cells to construct more complex architectures, however, to understand why LSTMs are also a recurrent type of neural network, let's examine the "vanilla" LSTM:

A "vanilla" LSTM is a many-to-many LSTM with a single cell unit. Is a recurrent type of network, because, as it can be seen in eq.(2.21), at each time-step t the previous cell state, which is the one that stores long-term dependencies, and the previous hidden state, which is the one that stores short-term dependencies, are fed as inputs to the LSTM cell. These state vectors, along with the current input vector, are processed by the LSTM cell to update its internal cell state and hidden state vectors. This process occurs recursively, with the updated states being fed back into the LSTM cell for the next step, effectively creating a loop. This flow of information from previous steps to the current step is a key characteristic of recurrent neural networks. Furhermore, this is a recurrent process, like regulars RNNs, the parameters from the cell equations eq.(2.21) are shared between across time-steps.

Thus, despite having a single cell, the vanilla LSTM exhibits recurrent behaviour due to its utilization of previous states vectors as inputs and the sharing of parameters.

By introducing cells with gating mechanisms, LSTMs can mitigate the vanishing gradient problem and better capture long-term dependencies in sequential data. The gates provide the flexibility to control the flow of information, allowing relevant information to be retained and propagated while filtering out irrelevant or redundant information, making it effective in tasks involving sequential data with long-term dependencies. This is suitable for tasks such as natural language processing, speech processing, and time series prediction.

2.4 Backpropagation Through Time for LSTM and Solving the Vanishing Gradients Problem

In backpropagation trough time (BPTT) for LSTMs, the algorithm "unfolds" the LSTM network over time, allowing the computation of the gradients at each time-step t. The unfolded network forms a chain-like structure, where each time-step corresponds to a "different state" of the LSTM cell. Different in the sense that, even though is the same cell, the internal states and components of the cell are different at every time-step. The gradients are backpropagated through the time-steps flowing through the LSTM cells. Since the parameters of a cell are shared across time-steps, the gradients from each time-step are accumulated, similar as in RNNs. BPTT for LSTMs involves more complex computations compared to traditional RNNs due to the presence of cells and gating mechanisms. The gradient calculation considers the interactions between the gates, cell state, hidden state, and weight matrices.

Consider the loss function

$$L(\hat{\mathbf{y}}, \mathbf{y}) = \sum_{t=1}^{T} L_t(\hat{\mathbf{y}}_t, \mathbf{y}_t)$$

First, let us compute the gradients with respect to $\mathbf{W}_{hy}$ and $\mathbf{b}_y$:

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{W}_{hy}} &= \sum_{t=1}^{T} \frac{\partial L_t}{\partial \mathbf{W}_{hy}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{W}_{hy}} \\ \frac{\partial L}{\partial \mathbf{b}_y} &= \sum_{t=1}^{T} \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{b}_y} \end{aligned} \tag{2.23}$$

Now, let's examine the gradients of L with respect to the weight matrices associated to the hidden state vector: $\mathbf{W}_{hi}$, $\mathbf{W}_{hf}$, $\mathbf{W}_{ho}$ and $\mathbf{W}_{hc}$, we introduce the following notation: to denote the weight matrices $\mathbf{W}_{h\alpha}$, and to denote the non-linear activation function we use $\phi_\alpha()$ where $\alpha \in \{i, f, o, c\}$ and, if $\alpha = c$ then $\phi_\alpha() = \tanh()$, else $\phi_\alpha() = \sigma()$.

At step t the gradient is $\frac{\partial L_t}{\partial \mathbf{W}_{h\alpha}} = \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{c}_t} \frac{\partial \mathbf{c}_t}{\partial \boldsymbol{\alpha}_t} \frac{\partial \boldsymbol{\alpha}_t}{\partial \mathbf{W}_{h\alpha}}$ if $\boldsymbol{\alpha}_t \neq \mathbf{o}_t \wedge \alpha \neq o$ else $\frac{\partial L_t}{\partial \mathbf{W}_{ho}} = \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{o}_t} \frac{\partial \mathbf{o}_t}{\partial \mathbf{W}_{ho}}$, however, as it can be seen from eq.(2.21), at step t the cell state vector $\mathbf{c}_t$ is a function of the previous cell state vector $\mathbf{c}_{t-1}$ who is also function of $\boldsymbol{\alpha}_{t-1}(\mathbf{W}_{h\alpha})$, which in turn, is a function of the previous cell state vector $\mathbf{c}_{t-2}$ who is also function of $\boldsymbol{\alpha}_{t-2}(\mathbf{W}_{h\alpha})$ and so on. $\mathbf{c}_t(\mathbf{c}_{t-1}(\mathbf{c}_{t-2}(\mathbf{c}_{t-3}(\cdots))))$. Therefore, the correct computation of the gradients at time-step t is given by:

$$\begin{aligned} \frac{\partial L_t}{\partial \mathbf{W}_{h\alpha}} &= \sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{c}_t} \frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_k} \frac{\partial \mathbf{c}_k}{\partial \boldsymbol{\alpha}_t} \frac{\partial \boldsymbol{\alpha}_t}{\partial \mathbf{W}_{h\alpha}}, \frac{\partial L_t}{\partial \mathbf{W}_{x\alpha}} = \sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{c}_t} \frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_k} \frac{\partial \mathbf{c}_k}{\partial \boldsymbol{\alpha}_t} \frac{\partial \boldsymbol{\alpha}_t}{\partial \mathbf{W}_{x\alpha}} \\ \frac{\partial L_t}{\partial \mathbf{W}_{ho}} &= \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{o}_t} \frac{\partial \mathbf{o}_t}{\partial \mathbf{W}_{ho}}, \frac{\partial L_t}{\partial \mathbf{W}_{xo}} = \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{o}_t} \frac{\partial \mathbf{o}_t}{\partial \mathbf{W}_{xo}} \end{aligned} \tag{2.24}$$

where $\boldsymbol{\alpha}_t \in \{\mathbf{i}_t, \mathbf{f}_t, \tilde{\mathbf{c}}_t\}$, $\alpha \in \{i, f, c\}$ and

$$\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_k} = \prod_{j=k+1}^{t} \frac{\partial \mathbf{c}_j}{\partial \mathbf{c}_{j-1}} = \frac{\partial \mathbf{c}_{k+1}}{\partial \mathbf{c}_k} \frac{\partial \mathbf{c}_{k+2}}{\partial \mathbf{c}_{k+1}} \frac{\partial \mathbf{c}_{k+3}}{\partial \mathbf{c}_{k+2}} \cdots \frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} \tag{2.25}$$

Here, it is important to highlight that, in contrast to RNNs, there is no recurrent product of derivative of the hidden state $\partial \mathbf{h}_t / \partial \mathbf{h}_k$, this is because, as can be observed from the hidden state vector formula: (from eq.(2.21)) $\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$, the hidden state vector $\mathbf{h}_t$ does not depend recurrently from the previous hidden state vector $\mathbf{h}_{t-1}$. However, in the LSTM case, the recurrent product of derivatives is $\partial \mathbf{c}_t / \partial \mathbf{c}_k$ which arises from the recurrent dependence of the cell state vector update formula: (from eq.(2.21)) $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$. Also, the formula for cell state update $\mathbf{c}_t$ does not depend from the output gate $\mathbf{o}_t$, that is why the computation of the gradient of L with respect to $\mathbf{W}_{ho}$ and $\mathbf{W}_{xo}$ is different from the other parameters, as shown in the last two equations of eq.(2.24).

Aggregating the gradients over all the T time-steps from the input sequence

$\mathbf{X} = (\mathbf{x}_1, \cdots, \mathbf{x}_{t-1}, \mathbf{x}_t, \mathbf{x}_{t+1}, \cdots, \mathbf{x}_T)$ yields the gradients:

$$\frac{\partial L}{\partial \mathbf{W}_{h\alpha}} = \sum_{t=1}^{T}\sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{c}_t}\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_k}\frac{\partial \mathbf{c}_k}{\partial \boldsymbol{\alpha}_t}\frac{\partial \boldsymbol{\alpha}_t}{\partial \mathbf{W}_{h\alpha}}, \frac{\partial L}{\partial \mathbf{W}_{x\alpha}} = \sum_{t=1}^{T}\sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{c}_t}\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_k}\frac{\partial \mathbf{c}_k}{\partial \boldsymbol{\alpha}_t}\frac{\partial \boldsymbol{\alpha}_t}{\partial \mathbf{W}_{x\alpha}}$$
$$\frac{\partial L}{\partial \mathbf{W}_{ho}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{o}_t}\frac{\partial \mathbf{o}_t}{\partial \mathbf{W}_{ho}}, \frac{\partial L}{\partial \mathbf{W}_{xo}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t}\frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t}\frac{\partial \mathbf{h}_t}{\partial \mathbf{o}_t}\frac{\partial \mathbf{o}_t}{\partial \mathbf{W}_{xo}} \tag{2.26}$$

where $\boldsymbol{\alpha}_t \in \{\mathbf{i}_t, \mathbf{f}_t, \tilde{\mathbf{c}}_t\}$, $\alpha \in \{i, f, c\}$ and $\partial \mathbf{c}_t / \partial \mathbf{c}_k$ is given by eq.(2.25)

Let us compute and examine the behavior of the recurrent product $\partial \mathbf{c}_t / \partial \mathbf{c}_k$ in the gradients. For simplicity, we introduce the following notation:

$$\gamma_{it} = \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{W}_{xi}\mathbf{x}_t + \mathbf{b}_i, \gamma_{ft} = \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{W}_{xf}\mathbf{x}_t + \mathbf{b}_f, \gamma_{ot} = \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{W}_{xo}\mathbf{x}_t + \mathbf{b}_o$$

For a single step $t-1 \to t$, considering the cell equations eq.(2.21), $\partial \mathbf{c}_t / \partial \mathbf{c}_k$ is computed as:

$$\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} = \frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} + \frac{\partial \mathbf{c}_t}{\partial \mathbf{f}_t}\frac{\partial \mathbf{f}_t}{\partial \gamma_{ft}}\frac{\partial \gamma_{ft}}{\partial \mathbf{h}_{t-1}}\frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{c}_{t-1}} + \frac{\partial \mathbf{c}_t}{\partial \mathbf{i}_t}\frac{\partial \mathbf{i}_t}{\partial \gamma_{it}}\frac{\partial \gamma_{it}}{\partial \mathbf{h}_{t-1}}\frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{c}_{t-1}} + \frac{\partial \mathbf{c}_t}{\partial \tilde{\mathbf{c}}_t}\frac{\partial \tilde{\mathbf{c}}_t}{\partial \gamma_{ct}}\frac{\partial \gamma_{ct}}{\partial \mathbf{h}_{t-1}}\frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{c}_{t-1}} \tag{2.27}$$

First, from equation eq.(2.27) consider $\frac{\partial \mathbf{c}_t}{\partial \mathbf{i}_t}\frac{\partial \mathbf{i}_t}{\partial \gamma_{it}}\frac{\partial \gamma_{it}}{\partial \mathbf{h}_{t-1}}\frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{c}_{t-1}}$:

since $\mathbf{i}_t \odot \tilde{\mathbf{c}}_t = \begin{bmatrix} i_{t_1} \cdot \tilde{c}_{t_1}, & i_{t_2} \cdot \tilde{c}_{t_2}, & i_{t_3} \cdot \tilde{c}_{t_3}, & \cdots, & i_{t_n} \cdot \tilde{c}_{t_n} \end{bmatrix}^T$ then

$$\frac{\partial \mathbf{c}_t}{\partial \mathbf{i}_t} = \frac{\partial \mathbf{i}_t \odot \tilde{\mathbf{c}}_t}{\partial \mathbf{i}_t} = \begin{bmatrix} \frac{\partial i_{t_1} \cdot \tilde{c}_{t_1}}{\partial i_{t_1}} & \frac{\partial i_{t_1} \cdot \tilde{c}_{t_1}}{\partial i_{t_2}} & \frac{\partial i_{t_1} \cdot \tilde{c}_{t_1}}{\partial i_{t_3}} & \cdots & \frac{\partial i_{t_1} \cdot \tilde{c}_{t_1}}{\partial i_{t_n}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial i_{t_n} \cdot \tilde{c}_{t_n}}{\partial i_{t_1}} & \frac{\partial i_{t_n} \cdot \tilde{c}_{t_n}}{\partial i_{t_2}} & \frac{\partial i_{t_n} \cdot \tilde{c}_{t_n}}{\partial i_{t_3}} & \cdots & \frac{\partial i_{t_n} \cdot \tilde{c}_{t_n}}{\partial i_{t_n}} \end{bmatrix} = \begin{bmatrix} \tilde{c}_{t_1} & 0 & 0 & \cdots & 0 \\ 0 & \tilde{c}_{t_2} & 0 & \cdots & 0 \\ 0 & 0 & \tilde{c}_{t_3} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \tilde{c}_{t_n} \end{bmatrix} = diag[\tilde{\mathbf{c}}_t] \tag{2.28}$$

$$\frac{\partial \mathbf{i}_t}{\partial \gamma_{it}} = \begin{bmatrix} \frac{\partial i_{t_1}}{\partial \gamma_{it_1}} & \frac{\partial i_{t_1}}{\partial \gamma_{it_2}} & \frac{\partial i_{t_1}}{\partial \gamma_{it_3}} & \cdots & \frac{\partial i_{t_1}}{\partial \gamma_{it_n}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial i_{t_n}}{\partial \gamma_{it_1}} & \frac{\partial i_{t_n}}{\partial \gamma_{it_2}} & \frac{\partial i_{t_n}}{\partial \gamma_{it_3}} & \cdots & \frac{\partial i_{t_n}}{\partial \gamma_{it_n}} \end{bmatrix} = \begin{bmatrix} \sigma'(\gamma_{it_1}) & 0 & 0 & \cdots & 0 \\ 0 & \sigma'(\gamma_{it_2}) & 0 & \cdots & 0 \\ 0 & 0 & \sigma'(\gamma_{it_3}) & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \sigma'(\gamma_{it_n}) \end{bmatrix}$$
$$= diag[\sigma'(\gamma_{it})] \tag{2.29}$$

Now, since $\gamma_{it_k} = \sum_{j=1}^{n} w_{hi_{kj}} h_{t-1_j} + \sum_{j=1}^{n} w_{xi_{kj}} x_j + b_{i_k}$ then $\frac{\partial \gamma_{it_k}}{\partial h_{t-1_j}} = w_{hi_{kj}}$ and

$$\frac{\partial \boldsymbol{\gamma}_{it}}{\partial \mathbf{h}_{t-1}} = \begin{bmatrix} \frac{\partial \gamma_{it_1}}{\partial h_{t-1_1}} & \frac{\partial \gamma_{it_1}}{\partial h_{t-1_2}} & \frac{\partial \gamma_{it_1}}{\partial h_{t-1_3}} & \cdots & \frac{\partial \gamma_{it_1}}{\partial h_{t-1_n}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \gamma_{it_n}}{\partial h_{t-1_1}} & \frac{\partial \gamma_{it_n}}{\partial h_{t-1_2}} & \frac{\partial \gamma_{it_n}}{\partial h_{t-1_3}} & \cdots & \frac{\partial \gamma_{it_n}}{\partial h_{t-1_n}} \end{bmatrix} = \begin{bmatrix} w_{hi_{11}} & w_{hi_{12}} & w_{hi_{13}} & \cdots & w_{hi_{1n}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{hi_{n1}} & w_{hi_{n2}} & w_{hi_{n3}} & \cdots & w_{hi_{nn}} \end{bmatrix} = \mathbf{W}_{hi} \tag{2.30}$$

For $\frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{c}_{t-1}}$ since $\mathbf{h}_{t-1} = \mathbf{o}_{t-1} \odot \tanh(\mathbf{c}_{t-1}) = \begin{bmatrix} o_{t-1_1} \cdot \tanh(c_{t-1_1}) \\ o_{t-1_2} \cdot \tanh(c_{t-1_2}) \\ o_{t-1_3} \cdot \tanh(c_{t-1_3}) \\ \vdots \\ o_{t-1_n} \cdot \tanh(c_{t-1_n}) \end{bmatrix}$ then

$$\begin{aligned} \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{c}_{t-1}} &= \begin{bmatrix} \frac{\partial h_{t-1_1}}{\partial c_{t-1_1}} & \frac{\partial h_{t-1_1}}{\partial c_{t-1_2}} & \frac{\partial h_{t-1_1}}{\partial c_{t-1_3}} & \cdots & \frac{\partial h_{t-1_1}}{\partial c_{t-1_n}} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial h_{t-1_n}}{\partial c_{t-1_1}} & \frac{\partial h_{t-1_n}}{\partial c_{t-1_2}} & \frac{\partial h_{t-1_n}}{\partial c_{t-1_3}} & \cdots & \frac{\partial h_{t-1_n}}{\partial c_{t-1_n}} \end{bmatrix} = \begin{bmatrix} o_{t-1_1} \cdot \tanh'(c_{t-1_1}) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & o_{t-1_n} \cdot \tanh'(c_{t-1_n}) \end{bmatrix} \\ &= diag[\mathbf{o}_{t-1} \odot \tanh'(\mathbf{c}_{t-1})] \end{aligned} \tag{2.31}$$

Equations (2.28), (2.29), (2.30), (2.31) yield:

$$\frac{\partial \mathbf{c}_t}{\partial \mathbf{i}_t} \frac{\partial \mathbf{i}_t}{\partial \boldsymbol{\gamma}_{it}} \frac{\partial \boldsymbol{\gamma}_{it}}{\partial \mathbf{h}_{t-1}} \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{c}_{t-1}} = diag[\tilde{\mathbf{c}}_t] \times diag[\sigma'(\boldsymbol{\gamma}_{it})] \times \mathbf{W}_{hi} \times diag[\mathbf{o}_{t-1} \odot \tanh'(\mathbf{c}_{t-1})]$$

Analogously the other terms from eq.(2.27) can be computed in the same way as the equations (2.28), (2.29), (2.30), (2.31), the main difference is that, $\tilde{\mathbf{c}}_t$ uses $\tanh(\cdot)$ instead of $\sigma(\cdot)$, so to calculate $\frac{\partial \tilde{\mathbf{c}}_t}{\partial \boldsymbol{\gamma}_t}$ the process is the same as in eq.(2.29), but the resultant matrix has $\sigma'(\cdot)$ at the diagonal instead of $\tanh'(\cdot)$. We obtain:

$$\begin{aligned} \frac{\partial \mathbf{c}_t}{\partial \mathbf{f}_t} \frac{\partial \mathbf{f}_t}{\partial \boldsymbol{\gamma}_{ft}} \frac{\partial \boldsymbol{\gamma}_{ft}}{\partial \mathbf{h}_{t-1}} \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{c}_{t-1}} &= diag[\mathbf{c}_{t-1}] \times diag[\sigma'(\boldsymbol{\gamma}_{ft})] \times \mathbf{W}_{hf} \times diag[\mathbf{o}_{t-1} \odot \tanh'(\mathbf{c}_{t-1})] \\ \frac{\partial \mathbf{c}_t}{\partial \mathbf{i}_t} \frac{\partial \mathbf{i}_t}{\partial \boldsymbol{\gamma}_{it}} \frac{\partial \boldsymbol{\gamma}_{it}}{\partial \mathbf{h}_{t-1}} \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{c}_{t-1}} &= diag[\tilde{\mathbf{c}}_t] \times diag[\sigma'(\boldsymbol{\gamma}_{it})] \times \mathbf{W}_{hi} \times diag[\mathbf{o}_{t-1} \odot \tanh'(\mathbf{c}_{t-1})] \\ \frac{\partial \mathbf{c}_t}{\partial \tilde{\mathbf{c}}_t} \frac{\partial \tilde{\mathbf{c}}_t}{\partial \boldsymbol{\gamma}_{ct}} \frac{\partial \boldsymbol{\gamma}_{ct}}{\partial \mathbf{h}_{t-1}} \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{c}_{t-1}} &= diag[\mathbf{i}_t] \times diag[\tanh'(\boldsymbol{\gamma}_{ct})] \times \mathbf{W}_{hc} \times diag[\mathbf{o}_{t-1} \odot \tanh'(\mathbf{c}_{t-1})] \end{aligned} \tag{2.32}$$

Then, for a single step $t-1 \rightarrow t$, $\partial \mathbf{c}_t / \partial \mathbf{c}_k$ eq.(2.27) is computed as:

$$\begin{aligned} \frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_{t-1}} &= diag[\mathbf{f}_t] + diag[\mathbf{c}_{t-1}] \times diag[\sigma'(\boldsymbol{\gamma}_{ft})] \times \mathbf{W}_{hf} \times diag[\mathbf{o}_{t-1} \odot \tanh'(\mathbf{c}_{t-1})] \\ &+ diag[\tilde{\mathbf{c}}_t] \times diag[\sigma'(\boldsymbol{\gamma}_{it})] \times \mathbf{W}_{hi} \times diag[\mathbf{o}_{t-1} \odot \tanh'(\mathbf{c}_{t-1})] \\ &+ diag[\mathbf{i}_t] \times diag[\tanh'(\boldsymbol{\gamma}_{ct})] \times \mathbf{W}_{hc} \times diag[\mathbf{o}_{t-1} \odot \tanh'(\mathbf{c}_{t-1})] \end{aligned} \tag{2.33}$$

Thus, if we backpropagate through $[k, t]$ steps, the gradient from eq.(2.25) will be:

$$\begin{aligned}\frac{\partial \mathbf{c}_t}{\partial \mathbf{c}_k} = \prod_{j=k+1}^{t} \frac{\partial \mathbf{c}_j}{\partial \mathbf{c}_{j-1}} = \prod_{j=k+1}^{t} \Big(& diag[\mathbf{f}_j] \\ & + diag[\mathbf{c}_{j-1}] \times diag[\sigma'(\gamma_{fj})] \times \mathbf{W}_{hf} \times diag[\mathbf{o}_{j-1} \odot \tanh'(\mathbf{c}_{j-1})] \\ & + diag[\tilde{\mathbf{c}}_j] \times diag[\sigma'(\gamma_{ij})] \times \mathbf{W}_{hi} \times diag[\mathbf{o}_{j-1} \odot \tanh'(\mathbf{c}_{j-1})] \\ & + diag[\mathbf{i}_j] \times diag[\tanh'(\gamma_{cj})] \times \mathbf{W}_{hc} \times diag[\mathbf{o}_{j-1} \odot \tanh'(\mathbf{c}_{j-1})] \Big)\end{aligned} \tag{2.34}$$

The difference between the recursive gradient $\partial \mathbf{c}_t / \partial \mathbf{c}_{t-1}$ and the one from the vanilla RNN $\partial \mathbf{h}_t / \partial \mathbf{h}_{t-1}$ (from 2.13, 2.14), is that, in vanilla RNNs, the terms $\partial \mathbf{h}_t / \partial \mathbf{h}_{t-1}$, as $t \to \infty$, will eventually take on values that are *always* > 1 or *always* in the range $[0, 1)$, as was shown in eqs. (2.19), (2.20), this is what leads to the vanishing and exploding gradients. However at $\partial \mathbf{c}_t / \partial \mathbf{c}_{t-1}$ at *any time-step* can take on values that are *either* > 1 *or* within the range $[0, 1]$. This is because, as it can be observed from equation (2.33) the gradient is modulated by the gate vectors, i.e., they directly influence and control it. During the training process the network learns to adjust the gate vectors in a way that their modulation over the gradients will allow the flow of gradients through the time-steps without being diminished exponentially. Thus, if $\partial \mathbf{c}_t / \partial \mathbf{c}_{t-1}$ starts converging to zero, the network will adjust the values of the gates in order to bring the value of $\partial \mathbf{c}_t / \partial \mathbf{c}_{t-1}$ closer to 1, preventing the gradients from vanishing (or preventing them from vanishing too quickly). This modulation provides a direct path for the gradients to flow across long sequences, effectively overcoming the vanishing gradients problem.

However, LSTMs do not inherently solve the exploding gradients problem. The intuition is the following, if all terms of eq.(2.33), but the first one $diag[\mathbf{f}_t]$, are converging towards zero as $t \to \infty$, the network can adjust the values of the gate $\mathbf{f}_t$ so that the gradient stops approaching zero and moved closer to 1. However, if all other terms of eq.(2.33) are exponentially increasing as $t \to \infty$, the term $diag[\mathbf{f}_t]$ will not be able to stop the gradient from exploding since the range of $\mathbf{f}_t$ is $[0, 1]$, so the network can't adjust it to stop the gradient from exploding.

To address the exploding gradients problem, additional techniques such as gradient clipping [3] [4] are typically implemented.

3 Problem Description

Voices are different from person to person. This uniqueness arises from a combination of physiological factors, the unique characteristics of an individual's voice, including pitch, tone, and timbre, are influenced by the interplay between the characteristics of the vocal cords, resonances of the vocal tract, and articulatory control. This factors collectively contribute to a wide range of voice variations among individuals. As a result, each person's voice possesses a distinct vocal identity that is recognizable and unique. Given the distinctiveness of each

individual's voice, voices can be employed as a form of biometric authentication, akin to fingerprints or face recognition.

Speaker Verification (SV) is the task of accepting or rejecting a speaker's identity claim through the analysis of their acoustic samples. Speaker verification methods are divided into *Text-Dependent (TD)* and *Text-Independent (TI)*. In TD-SV the model possess prior knowledge about the phrase, often referred to as the *global password*, which the user is expected to speak in order to authenticate their identity. In TI-SV the verification process, as the name suggest, is not dependent from a specific phrase for verification.

TD models achieve high speaker verification performance from training with relatively short utterances. In contrast, TI systems require a larger number of diverse utterances from each speaker to train reliable models and achieve good performance.

The standard speaker verification protocol is divided in three steps:

Development: background models are trained using a diverse dataset that represents a wide range of speakers. These models serve as a reference or representation of the general characteristics exhibited by speakers, capturing the variability of speech across multiple individuals. The objective of this phase is to construct a robust and comprehensive model that encompasses the speaker manifold. Here, the term *Speaker Manifold* refers to the underlying representation or space that captures the characteristics or properties of speakers, it can be analyzed as a high-dimensional space where each point corresponds to a distinct speaker. The ultimate aim of training these background models is to provide an effective internal representation (embedding) of an utterance, allowing for a simple scoring function.

Enrollment: The trained background model is used to enroll *new* speakers who were not part of the training data used during the development phase. Each new speaker provides a limited number of utterances, which are utilized to estimate the enrolled speaker model. These models capture the distinctive features and characteristics unique to each individual speaker.

The key difference between the development and enrollment phases lies in the nature of the models (embeddings) created: background models in the development phase represent the overall speaker manifold, while speaker-dependent models in the enrollment phase capture the individual speaker's unique characteristics.

Evaluation: The *test* utterances are matched against the *enrolled* speaker models to determine the claimed speaker's identity. The value of a scoring function of the test utterance model $\mathbf{x}_{eval}$ and the speaker model of the claimed speaker's $\mathbf{c}_{spk}$, denoted as $s(\mathbf{x}_{eval}, \mathbf{m}_{spk})$ is compared against a pre-defined threshold. If the score exceeds the threshold, indicating that the test utterance originates from the claimed comes from speaker *spk*, the verification is accepted, otherwise is not.

In the context of deep learning, the term *End-to-End* refers to an approach where a complex learning system or pipeline, consisting of multiple interconnected components, is represented by a *single unified model*. This approach removes the need for separate steps or models that

traditionally exist in pipeline designs. Instead, end-to-end models aim to learn and optimize the complete process.

The traditional speaker verification approach described above is not considered end-to-end because it involves separate stages for model training, speaker modeling and verification.

The present work aims to address the following task:

Consider a dataset consiting of several utterances from different speakers. Let $\mathbf{x}_{ji}$ be the l-dimensional log-Mel filterbank energy vector (Sec.1.3) from the i-th utterance of the j-th speaker. Our objectives are:

To generate meaningful embeddings $\mathbf{e}_{ji}$ for each $\mathbf{x}_{ji}$ that effectively capture the distinctive features and characteristics of the associated speaker from the speaker manifold, resulting in discriminative embeddings, allowing for effective differentiation between speakers.

To implement an end-to-end deep learning architecture for text-independent speaker verification for a small-scale dataset. The architecture must use a single network that emulates the three steps of the traditional speaker verification pipeline. It should directly map input utterances to their corresponding speaker embeddings, while optimizing an unified loss function for the entire architecture, to jointly learn the speaker modeling and verification process in a cohesive manner.

4 Literature Review

In this section we proceed to review some papers that introduced several keys components and methods that helped to design and build the network architecture that is implement on the experimental part of this work.

4.1 Speaker Embeddings - d-vectors

The paper *"Deep Neural Networks for Small Footprint Text-Dependent Speaker Verification"* [6], proposed a speaker verification technique based on the use of Deep Neural Networks (DNNs) for small footprint text-dependent speaker verification. Here, "small footprint" refers to the scenario in which the model is designed to be deployed within constrained computational resources (e.g., smartphones).

The authors, during the *development* phase, used a DNN-based background model to directly model the speaker space, i.e., the underlying space that captures the characteristics or properties of speakers out of the collection of data. The DNN was trained to take input features that describe short segments of speech (frames) within a given context (the phrase "ok google") and predict the speaker identity associated with those frames. As a result of this training, the DNN learns to capture and encode important speaker-specific characteristics in the output activations of its hidden layers, particularly the last hidden layer. These output activations represent

a condensed representation of the speaker's voice characteristics in a reduced-dimensional space.

During the *enrollment* phase, the trained DNN background model was used to extract speaker specific features out of the utterances from the *enrollment* speakers, to produce a "speaker model" in the form of an embedding vector out of such features. Which is computed as the accumulated output activations of the last hidden layer. That is, for every frame of a given utterance belonging to a new speaker (a speaker that wasn't part of the development set), the trained background DNN model computes the output activations of the last hidden layer using standard feedforward propagation, and then accumulate those activations to form a new compact representation (embedding) of that speaker. This speaker model is referred to as ***deep vector*** or ***d-vector***. The final representation of the speaker is derived by averaging all the d-vectors corresponding to the speakers' enrollment utterances. This compact representation captures essential information about the speaker's voice characteristics in a reduced-dimensional space.

Here, is noteworthy to mention that, the authors choose to use the output from the last hidden layer instead of the softmax output layer for the following reasons: First, this can reduce the DNN model size for runtime by removing that layer during enrollment and evaluation, which also enables the use of a large number of development speakers without increasing the DNN size at runtime. Second, the authors observed better generalization to unseen speakers from the last hidden layer output.

The authors' hypothesis surrounding why that works is that: the trained background DNN, having learned compact representations (d-vectors) from the development set speakers in the output of the last hidden layer, may also be capable of effectively represent unseen speakers.

For the *evaluation* phase, the authors extracted the normalized d-vector from the test utterance, then they computed the cosine distance between the test d-vector and the claimed speaker's d-vector. The verification decision was made by comparing the distance to a threshold.

About the architecture: The authors trained a maxout DNN with 4 hidden layers and 256 nodes per layer. A pool size of 2 was used per layer. The first two layers do not use dropout while the last two layers drop 50 percent of activations after dropout, as shown in the figure bellow. The authors used rectified linear units as the non-linear activation function on hidden units and a learning rate of 0.001 with exponential decay (0.1 every $5M$ steps). The authors don't specify what optimization algorithm was used. The input of the DNN is formed by stacking the 40-dimensional log Mel filter-bank energy features extracted from a given frame, together with its context, 30 frames to the left and 10 frames to the right. The dimension of the training target vectors was 496, which was the same as the number of speakers in the development set. The final maxout DNN model contained $\sim 600K$ parameters.

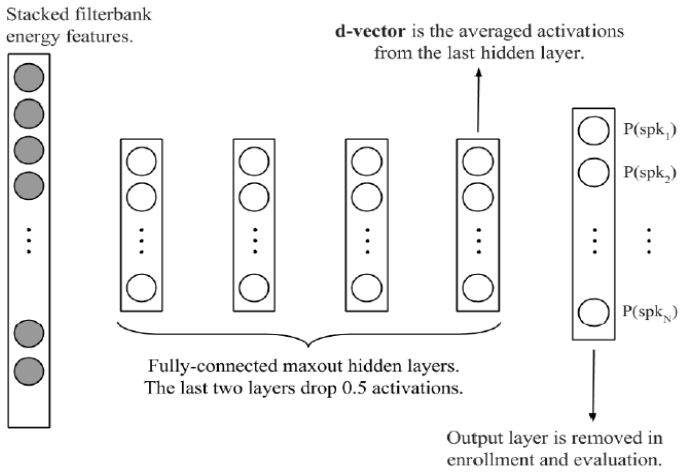

Figure 4.1: The proposed background DNN model at [6].

About the dataset: The data set contained 646 speakers speaking the same phrase, "ok google", many times in multiple sessions. 496 randomly selected speakers are used for training the background model and the remaining 150 speakers were used for enrollment and evaluation.

The authors' experimental results showed that the d-vector speaker verification system performs favorably compared to alternative approaches. Their results highlight the potential of the d-vectors as efficient embeddings that accurately models speaker's voice characteristics.

While the primary focus of of the paper centered around the text-dependent SV task, the authors also acknowledged the potential for extending their proposed technique can be extended to text-independent tasks.

The paper *"Locally-Connected and Convolutional Neural Networks for Small Footprint Speaker Recognition"* [7] introduced the idea of incorporating a locally-connected first hidden layer and convolutional first hidden layer as a modification to the baseline DNN model, instead of using fully-connected layers as originally proposed at [6]. We will not go into detail to explain what are locally-connected and convolutional neural networks since we won't make use of them for our final model. However, we will provide a brief overview of this paper since it presented some useful insight on how to approach modifications on the original d-vectors scheme that can be less computation expensive.

The authors explain that while d-vectors models have shown good performance on speaker recognition tasks, they can be computationally expensive and memory-intensive, especially when using long input sequences, such like in the case of text-independent speaker verification. This motivates the need for a more efficient neural network architecture that can reduce the memory footprint of the model without sacrificing performance.

The paper proposes a neural network architecture for speaker recognition that was designed to have a small footprint, i.e., for deployment on resource-constrained devices such as smartphones.

The authors tested two distinct network architectures to extract d-vectors. The first architecture comprised a locally-connected layer followed by fully-connected layers, while the second architecture incorporated a convolutional layer as the initial component, followed by fully-connected layers. The inclusion of these locally-connected and convolutional layers was proposed because, during the analysis of the baseline deep neural network (DNN) approach (introduced on the paper previously discussed), they performed a singular value decomposition (SVD) [8] [9] on the weight matrix of the initial layer, from which the authors observed that the weight matrix was sparse and low-rank.

Sparse matrices have a large number of zero entries relative to the total number of entries in the matrix. In the paper, the authors found that the weight matrices in the first fully-connected hidden layer had many small singular values, which is indicative of sparsity. This means that many of the connections between the neurons in the layer have very small weights, or no weights at all. The property of sparsity is important because it can lead to more efficient and interpretable models. By reducing the number of non-zero weights in the model, sparse models can be faster to compute and require less memory to store. Sparse models can also be easier to interpret because the non-zero weights correspond to the most important connections in the model.

To effectively exploit this sparsity and low-rank property, the authors proposed the adoption of locally-connected and convolutional layers. These layers not only feature a reduced number of trainable parameters compared to fully-connected layers but also facilitate weight sharing, which is conducive to capturing the inherent sparsity and low-rank structure of the weight matrices in the first hidden layer.

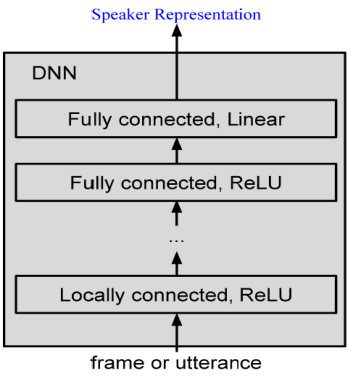

Figure 4.2: DNN with a locally-connected layer baseline approach. [7].

Upon their evaluation, the authors concluded that the network architecture incorporating a locally-connected layer yielded better performance. Ultimately reducing the number of parameters in the first hidden layer by an order of magnitude with minimal performance degradation.

4.2 Tuple based End-to-End Speaker Verification - TE2E

The paper *"End-to-End Text-Dependent Speaker Verification"* [10] introduces a tuple-based end-to-end architecture, in which the authors experimented with two different implementations

to compute the d-vectors, the first approach is based on the improvement to the original DNN approach proposed at [7], utilizing a DNN with a locally-connected layer followed by fully-connected layers to compute d-vectors. The second approach involves the use of long-short term memory recurrent neural networks(LSTMs) to compute the d-vectors.

The authors presented an end-to-end architecture that maps a test utterance and a few reference utterances to build the speaker model (d-vector), directly to a single score for verification and jointly optimizes the system's components using the same loss for training and evaluation. Such an approach results in simple and efficient systems, requiring little domain specific knowledge (e.g. vocabulary or language used in a specific application) and making few model assumptions. The authors implemented the idea by formulating the problem as a single neural network architecture (end-to-end) that emulates the three steps from the standard speaker verification approach, consisting of: a *training* component to compute the speaker representations (d-vectors), and *enrollment* component to estimate the speaker model (d-vector) from enrollment speakers, and an *evaluation* with a suitable loss function for optimization of the complete architecture, instead of separate loss functions for each component as is traditionally implemented in the non end-to-end approach. The architecture includes the estimation of a speaker model (the d-vector) on only a few utterances, and its evaluation on their internal "Ok Google" benchmark for text-dependent speaker verification. The choice of this global password relates to the Google Keyword Spotting system and Google Voice-Search system, which is of relevance to the authors since they conducted this research for Google.

About the architecture: The authors did an empirical comparison of two implementations of their end-to-end architecture, for which they used two different types of neural networks to obtain the speaker representation (d-vector) of an utterance. One consisting of a DNN with a locally-connected layer and fully connected layers as proposed at [7] and the other implementing a many-to-one long short-term memory recurrent neural network (LSTM) [5] [11].

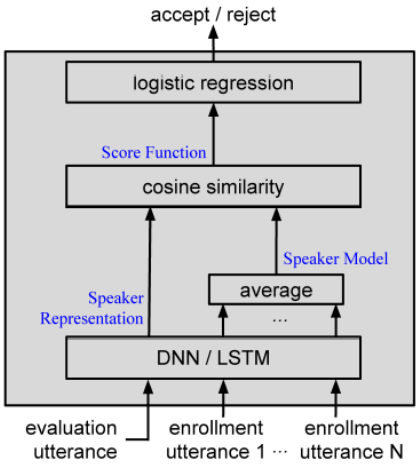

Figure 4.3: The end-to-end architecture proposed. [10].

Two networks were tested, one implementing a DNN with a locally-connected layer and fully-connected layers. The other one using a LSTM recurrent neural network. This difference is depicted in the figure as "DNN/LSTM".

DNNs assume a fixed-length input, to comply with this constraint, the authors stack the frames of a sufficiently large window of fixed length over the utterance and use them as the input. This trick is not needed for LSTMs (since they are designed to process sequential data), however, the authors used the same window of frames for better comparability. Also, unlike vanilla LSTMs which have multiple outputs, they only connected the last output to the loss layer to obtain a single, utterance-level speaker representation - the d-vector.

Is worth pointing out, that in the context of this paper, both speaker representation and speaker model are d-vectors, however, the difference is that, *speaker representation* refers to a d-vector obtained from a *testing* utterance and is computed during the "testing" phase, and *speaker model* is the average of d-vectors obtained from a speaker's *enrollment* utterances and is computed during the "enrollment" phase.

The input of the end-to-end architecture were *tuples* of $1+M$ utterances: for each training step, a *tuple* of *one evaluation utterance* $\mathbf{x}_{j\sim}$ and M *enrollment utterances* $\mathbf{x}_{km}$ (for $m = 1, \cdots, M$) is fed into the network: $\{\mathbf{x}_{j\sim}, (\mathbf{x}_{k1}, \cdots, \mathbf{x}_{kM})\}$, where $\mathbf{x}$ represents the features (log-mel-filterbank energies) from a fixed-length segment, j and k represent the speakers of the utterances, and j may or may not equal k. The tuple includes a single utterance from speaker j and M different utterances from speaker k.

A tuple is said to be *positive* if $\mathbf{x}_{j\sim}$ and the M enrollment utterances are from the same speaker, i.e., $j = k$, and *negative* otherwise. Positive and negative tuples were generated alternatively. For each input tuple, $L2$ normalized response of the LSTM was computed: $\{\mathbf{e}_{j\sim}, (\mathbf{e}_{k1}, \cdots, \mathbf{e}_{kM})\}$. Here each $\mathbf{e}$ is a d-vector (an embedding vector) of fixed dimension that results from the many-to-one mapping defined by the LSTM.

The *speaker model* is the average over a small number of "enrollment" representations

(d-vectors), is computed as the centroid of the tuple $(\mathbf{e}_{k1}, \cdots, \mathbf{e}_{kM})$:

$$\mathbf{c}_k = \mathbb{E}_m[\mathbf{e}_{km}] = \frac{1}{M}\sum_{m=1}^{M}\mathbf{e}_{km} \tag{4.1}$$

The authors used the same network to compute the internal representations of the *test* utterance and of the utterances for the speaker model. The cosine similarity between the speaker representation and the speaker model was used as scoring,

$$s(\mathbf{e}_{j\sim}, \mathbf{c}_k) = w \cdot \cos(\mathbf{e}_{j\sim}, \mathbf{c}_k) + b = w \cdot \mathbf{e}_{j\sim} \cdot ||\mathbf{c}_k||_2 + b \tag{4.2}$$

with learnable w and b. These scores were fed to a logistic regression including a linear layer with a bias. The architecture was optimized using the *Tuple-Based End-to-End loss (TE2E)* defined as:

$$l_{e2e} = -\log p(target),\ target \in \{accept, reject\}$$

where $target$ is a binary variable and: $p(accept) = 1/(1 + \exp(-ws(\mathbf{e}_{j\sim}, \mathbf{c}_k) - b))$ and $p(reject) = 1 - p(accept)$.

About the dataset: The authors tested the proposed end-to-end approach on a set composed of utterances of the phrase "Ok Google" collected from anonymized voice search logs. The utterances were forced aligned to obtain the ”Ok Google” snippets. The average length of these snippets was around 80 frames, for a frame rate of $100Hz$. Based on this observation, they extracted the last 80 frames from each snippet, possibly padding or truncating frames at the beginning of the snippet. The frames consisted of 40 log-Mel filter-banks energies each.

For the network using DNNs, they concatenated the 80 input frames, resulting in a 80×40-dimensional feature vector. the DNN consists of 4 hidden layers. All hidden layers in the DNN had 504 nodes and use ReLU activation except the last, which was linear. The patch size for the locally-connected layer of the DNN was 10×10.

For the network using LSTMs, they fed the 40-dimensional feature vectors frame by frame. They used a single LSTM layer with 504 nodes without a projection layer. The batch size was of 32 for both networks.

The authors reported that the proposed end-to-end network yielded an improvement in equal error rates in comparison with the non end-to-end DNN approach from [7]. Additionally, they reported that architecture implementing the LSTM network yielded better performance compared to the alternative architecture utilizing a DNN with locally-connected layer followed by fully-connected layers. The authors concluded that such an end-to-end approach offers several advantages when compared to existing approaches, including the direct modeling from utterances, which allows for capturing long-range context and reduces the complexity (one vs. number of frames evaluations per utterance), and the direct and joint estimation, which can lead to better and more compact models. Moreover, they mentioned that this approach often results

in considerably simplified systems requiring fewer concepts and heuristics.

The authors assured that while the paper focuses on text-dependent speaker verification, the proposed approach is more general and can be expanded to be used for text-independent speaker verification.

4.3 Batch based End-to-End Speaker Verification - GE2E

The paper *"Generalized End-to-End Loss for Speaker Verification"* [12] proposes a new loss function called *Generalized End-to-End Loss (GE2E)*, which makes the training of speaker verification models more efficient than the tuple-based end-to-end (TE2E) loss function proposed at [10]. Unlike TE2E, the loss function proposed in this paper, GE2E updates the network in a way that emphasizes examples that are difficult to verify at each step of the training process.

The authors introduced a generalization of their tuple-based architecture from [10]. The new architecture proposed constructs tuples from input sequences of various lengths in a more efficient way, leading to a significant boost of performance and training speed.

Generalized end-to-end (GE2E) training is based on processing a large number of utterances at once, in the form of a *batch*. As apposed to the previous tuple-based end-to-end approach (TE2E) [10] that on each iteration a single tuple of $1 + M$ utterances $\{\mathbf{x}_{j\sim}, (\mathbf{x}_{k1}, \cdots, \mathbf{x}_{kM})\}$ is processed per iteration, containing a up to two different speakers (in the case of $j \neq k$) as described in the previous paper overview.

In GE2E training a single batch consist of $N \times M$ utterances. These utterances are from N *different speakers*, and each speaker has M utterances. Each vector $\mathbf{x}_{ji}$ ($1 \leq j \leq N, 1 \leq i \leq M$) represents the log-Mel filterbank energies extracted from the j-th speaker's i-th utterance.

Similar to the TE2E approach, the features extracted from each utterance $\mathbf{x}_{ij}$ are fed into a many-to-one LSTM network. However, a linear layer is connected to the last LSTM layer as an additional transformation of the last frame response of the network. The embedding vector (d-vector) is the $L2$ normalization of the network output:

$$\mathbf{e}_{ji} = \frac{f(\mathbf{x}_{ji}, \mathbf{w})}{||f(\mathbf{x}_{ji}, \mathbf{w})||_2} \tag{4.3}$$

here $f(\mathbf{x}_{ji}; \mathbf{w})$ denotes the output of the entire neural network, where $\mathbf{w}$ represents all parameters of the network (including both LSTM layes and the linear layer). And $\mathbf{e}_{ji}$ represents the embedding vector of the j-th speaker's i-th utterance. The authors observed that, removing $\mathbf{e}_{ji}$ when computing the speaker model (centroid) of the true speaker makes the training stable and helps to avoid trivial solutions. So, the speaker model equation from eq.(4.1) is modified when calculating positive similarity (i.e., $k = j$):

$$\begin{cases} \mathbf{c}_j^{(-i)} = \frac{1}{M-1} \sum_{\substack{m=1 \\ m \neq i}}^{M} \mathbf{e}_{jm} & \text{if } k = j \\ \mathbf{c}_j = \frac{1}{M} \sum_{m=1}^{M} \mathbf{e}_{jm} & \text{otherwise} \end{cases} \tag{4.4}$$

The authors introduced the use of a *Similarity Matrix* for an entire speaker batch, rather then using single similarity scoring per tuple as it was performed in the previous TE2E implementation. The similarity matrix $\mathbf{S_{ji,k}}$ is defined as the scaled cosine similarities between *each* embedding vector (d-vector) $\mathbf{e}_{ji}$ to *all* centroids $\mathbf{c}_k$ *from the training batch*:

$$\mathbf{S}_{ji,k} = \begin{cases} w \cdot \cos(\mathbf{e}_{ji}, \mathbf{c}_j^{(-i)}) + b = w \cdot \mathbf{e}_{ji} \cdot ||\mathbf{c}_j^{(-i)}||_2 + b & \text{if } k = j \\ w \cdot \cos(\mathbf{e}_{ji}, \mathbf{c}_k) + b = w \cdot \mathbf{e}_{ji} \cdot ||\mathbf{c}_k||_2 + b & \text{otherwise} \end{cases} \tag{4.5}$$

where w and b are learnable parameters, $j, k \in \{1, \cdots, N\}$ and $j \in \{1, \cdots, M\}$. Also, the cosine similarity of two $L2$-normalized vectors is their dot product.

The authors constrained the weight to be positive $w > 0$, because they wanted for the similarity to increase as the cosine similarity becomes larger.

The major difference between TE2E and GE2E is:

- TE2E's similarity from eq.(4.2) is a *scalar* value that defines the similarity between the embedding vector $\mathbf{e}_{j\sim}$ and a *single tuple* centroid $\mathbf{c}_k$
- GE2E builds a similarity *matrix* eq.(4.5) that defines the similarities *between each* $\mathbf{e}_{ji}$ and *all* centroids (speaker models) $\mathbf{c}_k$.

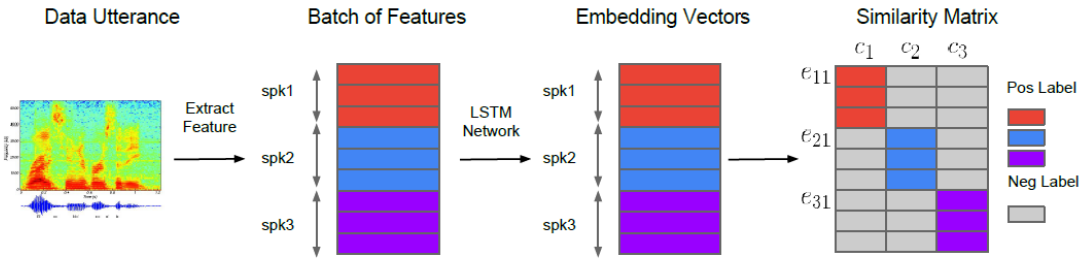

Figure 4.4: System overview. [12].

Depiction of the process involving features extraction, embedding vectors, and similarity scores from different speakers, represented by different colors.

Generalized End-to-End (GE2E) loss: The intuition behind the GE2E loss function is as follows: during the training, *we want the embedding of each utterance to be similar to the centroid of all that speaker's (true speaker) embeddings*, while at the same time, *far from other speakers' (false speakers) centroids*. This means that, for the similarity matrix from Figure 4.4, we want the similarity values of colored areas to be large (these represent similarity scores between positive pairs, i.e., the embedding's utterance correspond to the same speaker as the centroid), and the ones of gray areas to be small (these represent similarity scores between negative pairs, i.e., the embedding's utterance does not correspond to the same speaker as the centroid).

This concept is also depicted in the figure bellow: we want the blue embedding vector to be close to its own centroid - the blue triangle, and far from the others centroids - red and purple triangles, specially the closest one - the red triangle.

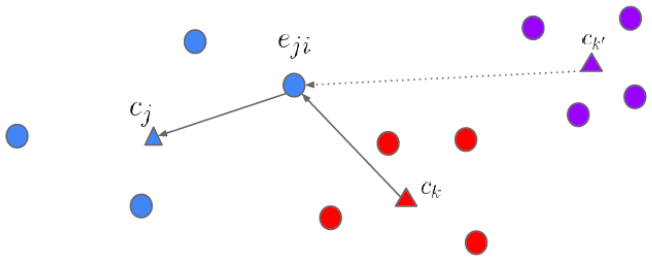

Figure 4.5: Depiction of GE2E loss. [12].

GE2E loss pushes the embeddings towards the centroid of the true speaker, and away from the centroid of the most similar different speaker.

This can be implemented as: Given an embedding vector $\mathbf{e}_{ij}$, all centroids $\mathbf{c}_k$, and the corresponding similarity matrix $\mathbf{S}_{ji,k}$, For $k = 1, \cdots, N$ we applied a softmax on the similarity matrix $\mathbf{S}_{ji,k}$ that makes the output equal to 1 if $k = j$, otherwise makes it 0. Thus, the loss on each embedding vector $\mathbf{e}_{ij}$ is defined as:

$$L_{ge2e}(\mathbf{e}_{ji}) = -\mathbf{S}_{ji,j} + \log \sum_{k=1}^{N} \exp(\mathbf{S}_{ji,k}) \tag{4.6}$$

The first term, $-\mathbf{S}_{ji,j}$, acts as a "pull" term. Here is important to remark that *all similarities scores are non-negative*. Therefore, by minimizing this term, the GE2E loss "pushes" the embedding $\mathbf{e}_{ji}$ vector towards its respective centroid $\mathbf{c}_j$, ensuring intra-speaker compactness.

The second term, $\log \sum_{k=1}^{N} \exp(\mathbf{S}_{ji,k})$, acts as a "push" term. The logarithm and exponential functions ensure a non-linear transformation that amplifies the false pairs similarities scores, this amplification is important because, as previously stated, in eq.(4.3) the embedding vectors are $L2$ normalized, consequently resulting in normalized centroids and similarities scores that are within the unitary sphere, therefore, the amplification of these scores will result in a more significant impact in the loss, emphasizing its importance in the optimization process. Encouraging the embedding vector to be far away from other speakers, thus prompting inter-speaker separability.

In combination, these two terms create a loss function that pushes each embedding vector close to *its own* centroid while simultaneously pulling it away *from all* other centroids. Also, as apposed to the TE2E loss, GE2E loss operates on a *set* of speaker embeddings instead of individual pairs of embeddings.

Equations (4.3), (4.4), (4.5), (4.6) yield the final GE2E loss L_{ge2e} which is the sum of all losses over the similarity matrix:

$$L_{ge2e}(\mathbf{x}; \mathbf{w}) = L_{ge2e}(\mathbf{S}) = \sum_{j,i} L_{ge2e}(\mathbf{e}_{ji}) \tag{4.7}$$

where $1 \leq j \leq N$ and $1 \leq i \leq M$.

Comparison between TE2E and GE2E: A *single batch* in GE2E has N speakers, each with

M utterances. Each *single step update* will push all $N \times M$ embedding vectors towards their own centroids, and pull them away from all other centroids. This mirrors what happens *with all possible tuples* in the TE2E loss function for each $\mathbf{x}_{ji}$. In the case of TE2E training:

Assume we randomly choose P utterances from speaker j when comparing speakers:

- Positive tuples (all utterances come from the same speaker, i.e, $k = j$): $\{\mathbf{x}_{ji}, (\mathbf{x}_{j,i1}, \cdots, \mathbf{x}_{j,iP})\}$ for $1 \leq i_p \leq M$ and $p = 1, \cdots, P$. There are $\binom{M}{P}$ such positive tuples.
- Negative tuples (all utterances come from a different speaker, i.e, $k \neq j$): $\{\mathbf{x}_{ji}, (\mathbf{x}_{k,i1}, \cdots, \mathbf{x}_{k,iP})\}$ for $k \neq j$ and $1 \leq i_p \leq M$ for $p = 1, \cdots, P$. *For each* $\mathbf{x}_{ji}$, we have to compare it with all other $N - 1$ centroids: for each fixed $\mathbf{x}_{ji}$, we will have $\binom{M}{P}$ negative tuples for each of the $N - 1$ speakers, resulting in a total of $(N - 1)\binom{M}{P}$ negative tuples for that $\mathbf{x}_{ji}$.

Each generated tuple is balanced with an opposite type of tuple, thus the total number is the maximum number of positive and negative tuples times 2. So, the total number of tuples in TE2E loss is:

$$2 \times \max\left(\binom{M}{P}, (N-1)\binom{M}{P}\right) = 2(N-1)\binom{M}{P} \geq 2(N-1)$$

The lower bound of the previous Equation occurs when $P = M$ ($\binom{M}{M} = \frac{M!}{M!(M-M)!} = \frac{1}{0!} = \frac{1}{1} = 1$). Thus, *each update* for $\mathbf{x}_{ji}$ in GE2E loss *is identical to at least* $2(N - 1)$ steps in TE2E loss. This analysis shows why GE2E updates models more efficiently than TE2E.

About the architecture: The paper describes the models employed as implementing a 3-layer LSTM with a final linear layer as a projection. The embedding vector (d-vector) size if the same as the LSTM projection size. The authors tested both text-dependent implementation (TD) and text-independent (TI) implementations. For TD they used 128 hidden nodes and projection size of 64. For TI they used 768 hidden nodes with projection size of 256. Batches of $N = 64$ speakers and $M = 10$ utterances per speaker were used. The authors trained their models using SGD with learning rate of 0.01 decreased by half every $30M$ steps. The $L2$-norm gradient was clipped at 3.

About the data The authors transformed the audio signal into frames of width $25ms$ and step $10ms$, then they extracted 40-dimension log-Mel filter-bank energies as the features for each frame. The authors tested both text-dependent implementation (TD) and text-independent (TI) implementations.

For TD the authors used: 1) a dataset of $\sim 630K$ speakers and $\sim 150M$ utterances of the phrase "OK Google". 2) A mixed data set of $\sim 18K$ speakers and $\sim 1.2M$ of the utterances "OK/Hey Google". For evaluation they used the Equal Error Rate.

For TI the authors the authors divided the training utterances into smaller segments, referred to as "partial utterances". Since each partial utterances in the same batch must be of the same length, for each batch data they randomly choose a time length t within $[lb, ub] = [140, 180]$ frames and enforced that all partial utterances in that batch are of length t. During evaluation, for

every utterance the authors applied a window of fixed size $(lb + ub)/2 = 160$ frames with 50% overlap. They computed the d-vector for each window, and the final utterance-wise d-vector was generated by $L2$ normalizing the window-wise d-vectors, then taking the element-wise average. The authors used a dataset of $\sim 36M$ utterances from $18K$ speakers, extracted from anonymized logs.

From their experiments, the authors reported that GE2E converges to a better model in shorter time than TE2E, which is consistent with the theoretical comparison between the two approaches. Also they reported that the GE2E loss function achieves better model performance compared to the TE2E loss function, it was observed significant improvements in the equal error rate (ERR) when using the GE2E model.

5 Approach Description

5.1 Dataset

The data samples used for training and testing our model are from the "train-other-500" subset of the "LirbisSpeech ASR corpus" [13] dataset, which consists of audio samples featuring read english speech. The "train-other-500" subset has a size of 30 GB and a total duration of ~ 500 hours of recorded audio samples, from $1,166$ speakers, comprising 564 female speakers and 602 male speakers. Each speaker's samples add to a total duration of ~ 30 minutes. Most of the speech is sampled at $16kHz$.

It is worth to point that, since the audio samples from the dataset is sourced from audio books, the tone and style of speech can differ significantly between utterances from the same speaker.

5.2 Metrics

Accuracy: During training we want to track if the model is accurately learning to classify the utterances' d-vectors (embeddings) $\mathbf{e}_{ij}$. Each iteration of the model computes the similarity matrix $\mathbf{S}$ as described by equation (4.5), the resulting similarity matrix $\mathbf{S}$ has dimensions $N * M \times N$ for the current bath of size $N \times M$ consisting of N speakers and M utterances per speaker. Each of the $N * M$ rows correspond to an utterance' d-vector, and each column corresponds to one speakers' embedding centroid. As illustrated in Figure 6.4. This means that each row in the range $[(i-1) \cdot M + 1, i \cdot M]$ correspond to the M utterances' embeddings of the i-th speaker.

The j-th embedding of the i-th speaker, $\mathbf{e}_{ij}$, has been correctly classified if the maximum of the cosine similarities between $\mathbf{e}_{ij}$ and all speakers' centroids corresponds to the cosine similarity between its own speaker's centroid (true speaker) $\mathbf{c}_i$:

$$\max_{1 \leq k \leq N} s(\mathbf{e}_{ij}, \mathbf{c}_k) = s(\mathbf{e}_{ij}, \mathbf{c}_i)$$

That is, if the j-th column of the row corresponding to $\mathbf{e}_{ij}$ on the similarity matrix $\mathbf{S}$ contains the maximum similarity score compared to all other columns in that row, then the utterance represented by the embedding vector $\mathbf{e}_{ij}$ has been correctly classified to its true speaker.

The accuracy score of the an iteration is the accuracy score of the similarity matrix S which is computed by the ratio of correctly classified embeddings to the total number of utterances from a batch:

$$Accuracy = \frac{\#CorrectlyClassifiedEmbeds}{N * M}$$

The accuracy score takes values between 0 and 1, where 1 is the highest accuracy score value representing that the complete batch of utterances was correctly classified.

Equal Error Rate (EER) [14]: Provides a measure of the point at which the false acceptance rate (FAR) and and false rejection rate (FRR) are equal. FAR represents the rate of incorrectly accepting impostors as genuine speakers, indicating the probability of a false match or false positive. FRR denotes the rate of incorrectly rejecting genuine speakers as impostors, indicating the probability of a false non-match or false negative. Conceptually, the EER represents the threshold at which the system makes an equal number of false acceptances and false rejections.

Mathematically, the EER is determined by finding the point on the Receiver Operating Characteristic (ROC) curve where the FAR and FRR intersect. The EER is the value where FAR equals FRR. For our implementation, at each iteration we calculated the ROC curve to obtain the False Positive Rates (FPR) and True Positive Rates (TPR), we then used the Brent's root-finding algorithm [15] implementation from 'scipy' to find the root of a lambda function that calculates the difference between FPR and 1 - TPR (which is FRR) at a given $x : x \in [0, 1]$. This root represents the threshold at which the FPR and FRR (1 - TPR) are equal, giving the EER value.

In practical terms, a lower EER indicates a more accurate system, meaning that the likelihood of incorrectly accepting impostors as genuine and incorrectly rejecting genuine instances is low.

Is worth pointing out that, while EER provides an important evaluation metric for speaker verification systems, it is not typically considered a general-purpose metric used in a wide range of deep learning tasks. It is mostly used in the domain of biometric security systems, such as face recognition, hand geometry and speaker verification.

5.3 Methodology

In this section, we proceed to describe our end-to-end approach for the text-independent speaker verification over the librispeech dataset.

During the preproccesing stage, we sampled the dataset audio files with a sampling rate of $16kHz$. By resampling all audio samples to the same sampling rate potential inconsistencies arising from mismatched sampling rates are eliminated. This ensures that all audio samples can be seamlessly integrated and processed together, with consistent sampling rates, the data can be

easily divided into fixed-length segments, minimizing any computational overhead associated with handling mismatched sampling rates enabling efficient subsequent preprocessing steps.

Next, to avoid selecting segments that are mostly silent when sampling the utterances, we employed a Voice Activity Detection (VAD) over the dataset audio samples. The VAD generates a binary flag over the audio indicating whether the segment is voiced or not. To improve the accuracy of the VAD, we applied a moving average, which is a statistical method that calculates the average over a specific window or interval, around this flag. By doing so, we obtained a smoother representation in the detection, reducing short-term fluctuations. After applying the moving average, we employed the VAD again and subsequently, we trimmed the audio by removing the unvoiced parts that are of a duration greater to $0.2s$. This ensured that the segments primarily contained voiced speech, enhancing the quality and relevance of the utterances.

Afterward, each utterance from the dataset was framed using a frame length of $25ms$ and a step of $10ms$, resulting in overlaps of $15ms$ between consecutive frames. The frame length and overlap were selected to effectively capture the temporal dynamics and preserve the contextual dependencies within the utterances. The frames were Hann windowed, ensuring smooth transitions and reducing spectral leakage during the subsequent feature extraction step. Next, we extracted 40-channel log-Mel filter-bank energies from the windowed utterances. These are the inputs to the model. Lastly, we split these vectors into disjointed train and test subsets. The disjointness between these subsets is important in the evaluation of the speaker verification system, since, if the same speakers were present in both subsets during testing we would not be able to properly assess how well the trained model generalizes to unseen speakers during the training phase.

Our implementation uses a 3-layer LSTM recurrent network with 256 hidden nodes per layer. A fully-connected linear layer of 64 of nodes is connected to the last LSTM layer as an additional transformation of the last frame response from the LSTM layers. In addition, we implemented a Rectified Linear Unit (ReLU) [16] layer connected to the output of the linear layer. The output of the model is the $L2$-normalized activations of the last hidden layer, resulting in 64-dimensional embedding vectors (d-vectors). The inclusion of the ReLU layer was made to potentially improve the separation between speaker embeddings. In contrast to the approach described in [12], which, for text-independent speaker verification, employed 3 LSTM layers with 768 hidden nodes and a linear projection layer of 256 nodes without ReLU layer connected to the linear projection. Consequently, the original approach yielded embedding vectors of size 256.

Our approach results in smaller embedding vectors, this reduction in dimensionality, of both the hidden layers and the embedding, is motivated by the following reasoning: in comparison to the data set used in [12] for the text-independent approach, which consisted of $\sim 36M$ utterances from $18K$ speakers, the dataset we implemented included $1,166$ speakers. Due to these differences in dataset sizes, it is reasonable to conclude that training our model without modifying the dimensions of the original proposed architecture could potentially result in overfitting, because

of the variation in magnitudes between the two datasets. Additionally, the authors of the paper trained their implementation for $50M$ steps, which if replicated on a smaller dataset will most certainly result in overfitting the model. Furthermore, one of the main goals of our work, is to modify the GE2E architecture and adapt it for implementation on a smaller scale, while still achieving good generalization.

During training, the network is fed batches of $N \times M$ utterances, where $N = 64$ is the number of speakers per batch, and $M = 10$ is the number of utterances per speaker. These dimensions were chosen based on the following considerations:

Firstly, during enrollment of a new speaker on a deployed model, it is expected to have multiple utterances from the new user, this is a common practice in biometric systems such as facial recognition (sometimes referred as "face unlock"), where multiple samples of a user's facial features are captured for enrollment. Therefore, 10 enrollment utterances provides a reasonable choice for a system to enroll a new speaker.

Secondly, the complexity of computing the similarity matrix is $O(N^2M)$, because of this, to ensure efficient training without significant slowdowns, the parameters N and M should be chosen carefully. Instead of choosing the largest possible batch size that fits on the GPU.

The utterances of each training batch are of a fixed number of 160 frames, these are partial utterances extracted from the complete of variable-duration utterances present in the dataset. To achieve this, we implemented a class that, when creating a batch, samples the log-Mel filterbank energy vectors of each utterance by randomly selecting a segment consisting of 160 frames from the utterance. Additionally, since there are multiple (partial) utterances for each speaker present in a batch, the sampling process assures that the same partial utterance will not be selected more than once in a single batch.

During training, for a single batch, the model computes the embeddings (d-vectors) for M *utterances* of fixed duration from N speakers as the $L2$-normalized activations of the last hidden layer: $\mathbf{e}_{ji}$ - the embedding vector (d-vector) of the j-th speaker's i-th utterance where $1 \leq i \leq M$, $1 \leq j \leq N$. Each training iteration yields a batch of $N * M = 640$ *utterance* d-vectors. Next, towards the end of constructing the similarity matrix $\mathbf{S}$, the *speaker* embeddings are computed. The j-th speaker embedding is computed as the centroid of its utterance d-vectors:

$$\mathbf{c}_j = \frac{1}{M} \sum_{m=1}^{M} \mathbf{e}_{jm}$$

Additionally, *exclusive* centroids are computed by excluding one utterance embedding from the computation:

$$\mathbf{c}_j^{(-i)} = \frac{1}{M-1} \sum_{\substack{m=1 \\ m \neq i}}^{M} \mathbf{e}_{jm}$$

The similarity matrix $\mathbf{S}$ is then constructed by computing the scaled cosine similarities between each utterance embedding vector $\mathbf{e}_{ji}$ and all speaker centroids $\mathbf{c}_k$ (from eq.(4.5)):

$$\mathbf{S}_{ji,k} = \begin{cases} w \cdot \cos(\mathbf{e}_{ji}, \mathbf{c}_j^{(-i)}) + b = w \cdot \mathbf{e}_{ji} \cdot ||\mathbf{c}_j^{(-i)}||_2 + b & \text{if } k = j \\ w \cdot \cos(\mathbf{e}_{ji}, \mathbf{c}_k) + b = w \cdot \mathbf{e}_{ji} \cdot ||\mathbf{c}_k||_2 + b & \text{otherwise} \end{cases}$$

where w and b are learnable parameters, $1 \leq j, k \leq N$ and $1 \leq i \leq M$. The motivation behind using exclusive centroids is discussed at [12]: In the non-exclusive centroid computation, all M utterance embeddings $\mathbf{e}_{ji}$ contribute to their respective speaker centroid $\mathbf{c}_j$ when computing the loss. This potentially introduces a bias towards the correct speaker j regardless of the model's accuracy. To address this, the authors found that excluding an utterance embedding $\mathbf{e}_{ji}$ during the similarity score computation for positive pairs (an embedding utterance and its true speaker centroid) led to more stable training.

However, the paper does not explicitly mention how to choose which utterance embedding to remove when $k = j$. Our implementation randomly selects an utterance embedding from the set of embeddings of a given speaker and removes it from the calculation of the speaker's centroid. This process is repeated multiple times during training, with different utterances being excluded each time. This approach aims to prevent overfitting to a specific set of utterances and promote generalization to new utterances from the same speaker.

Subsequently, the similarity matrix $\mathbf{S}$ is used to compute the generalized end-to-end (GE2E) loss (from eqs. (4.6), (4.7)):

$$L_{ge2e}(\mathbf{S}) = \sum_{j,i} L_{ge2e}(\mathbf{e}_{ji})$$

where

$$L_{ge2e}(\mathbf{e}_{ji}) = -\mathbf{S}_{ji,j} + \log \sum_{k=1}^{N} \exp(\mathbf{S}_{ji,k})$$

and $1 \leq j \leq N$, $1 \leq i \leq M$

The model is trained to optimize this GE2E loss (eq.(4.6), eq.(4.7)), so that the embeddings of utterances from the same speaker have high cosine similarity, while those of utterances from different speakers are far apart in the embedding space. The network parameters are optimized using an Adaptive Moment Estimation (Adam) [17] optimizer using an initial learning rate of $1e^{-4} = 0.0001$. In contrast, at [12] was used a Stochastic Gradient Descent (SGD) with a learning rate of 0.1. Our choice of using Adam over SGD is motivated by the following reasons:

SGD [18] can take a high number of iterations to reach the minima due to its inherent randomness in the descent process. In addition SGD uses the same learning rate for all parameters, which may not be the most effective approach for sparse data such as in our case where random segments from complete utterances are sampled, as we would prefer larger updates for rarely occurring features compared to frequently occurring ones. Because of this, choosing a proper learning rate is a challenge in SGD. Adam, on the other hand, is an extension of SGD that dynamically adjusts the learning rates during training, leading to a faster convergence of the

model and a more stable and efficient optimization. By using Adam as the optimizer we aim to achieve faster convergence of the model while effectively handling the characteristics of the data.

In addition, we clipped the gradient $L2$-norm at 3 since we are employing LSTM layers, which, as discussed previously (Sec. 2.3), effectively address the vanishing gradients problem but do not solve the exploding gradients problem. By setting a maximum threshold for the gradient norm, we ensure that the gradients do not become excessively large and destabilize the training process.

In the traditional non end-to-end approach, the development network is typically trained to generate embeddings while optimizing a separate loss function specifically for this process. During the verification step, a scoring function is used to compare the speaker model and enrollment model, determining the verification decision. To ensure that embeddings coming from the same speaker have high similarity score and embeddings coming from different speakers have a low similarity score, a separate verification model must be trained, this involves the optimization of a different loss function from the one used in the development step.

In contrast, the model that we implemented integrates the development, enrollment, and evaluation steps within a single network architecture. The scoring function outputs, the similarity matrix $\mathbf{S}$, serve as the inputs to the loss function. This enables the optimization of not only the embedding process (development and enrollment) but also the verification process under a *single* loss function. Consequently, these components are jointly learned, resulting in a unified approach to speaker verification.

5.4 Results

We trained the model for $150K$ steps and kept track of the accuracy score, loss and equal error rate for both the training and testing sets throughout the training process. In addition to this, we incorporated the use of the Uniform Manifold Approximation and Projection (UMAP) [19] technique, which is a dimensionality reduction technique that aims to capture the underlying structure and relationships in the data by mapping it to a lower-dimensional space. We implemented UMAP to periodically (every 100 epochs) project a batch of 64-dimensional embeddings into two-dimensional space to monitor how the model clusters speakers and to further contextualize our results, since, as the model's training and test losses decrease, it is expected that this would be reflected in the embedding space, as suggested by the GE2E loss (eq.(4.6), eq.(4.7)). Meaning that, we anticipate the formation of tight clusters of utterance embeddings from the same speaker as the training evolves, it is also expected to find a certain degree of separation between different clusters. The projections through UMAP provides a contextual understanding of the model's performance. We present some of these projections, for different epochs, where this behavior is presented:

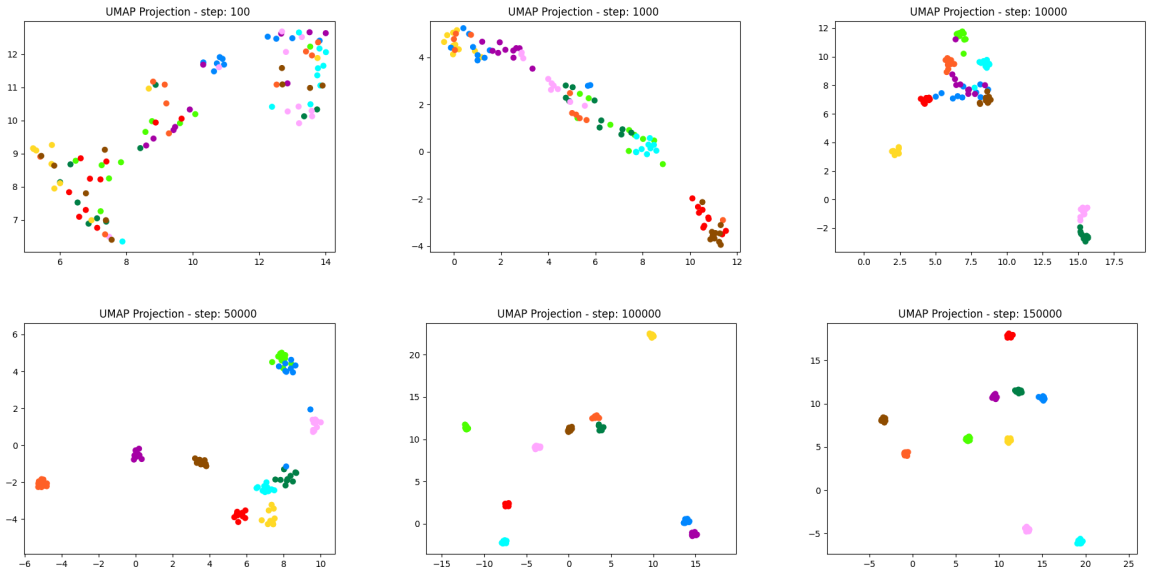

Figure 5.1: Embedding Space two-dimensional Projections over the train subset.

As can be observed in Figure 5.1, the initial 100 steps show the training embedding space as highly unclustered. The embeddings corresponding to the same speaker are scattered and mixed with others. This pattern continued for roughly the first $1,000$ steps from which after clusters start to become noticeable. By step $10,000$, embeddings of the same speaker are closer, but intersections with embeddings from different speakers are still very prominent. At step $50,000$ most of the same-speaker embeddings have significantly move closer to each other, although some remain distant or intersect with different clusters. Step $100,000$ reveals that tight clusters have been already formed, however, some of these clusters remain close to each other. At this point, the model learned to produce same-speaker embeddings that are close to each other, however, we continue the training process, allowing the model to further optimize the distance between different speakers clusters. Finally, by step $150,000$ the embedding space projection shows that the distance between clusters has significantly increased.

Although the model architecture is designed to handle inputs of variable length, it is anticipated that its performance would be better when dealing with utterances of similar duration as those encountered during training. Therefore, at testing time the utterance were split in frames of $1.6s$ with overlap of 50%, and the model was fed each segment individually. The outputs were averaged and $L2$-normalized to produce the utterance d-vector.

We halted the training at $150K$ steps since we determined that the model had converged by this point. The learning curves had flattened, showing no significant improvements. The loss values for the training set were already very low, fluctuating around 0.05. The training accuracy value stabilized at approximately 0.97 and the equal error rate remained bellow 1%.

All of these are indicators that the model has converged. To continue the training further would yield no significant improvements.

Bellow, we present the evolution of the training curves (Figure 5.2) and the values obtained from the training at step 150, 000 (Table 1).

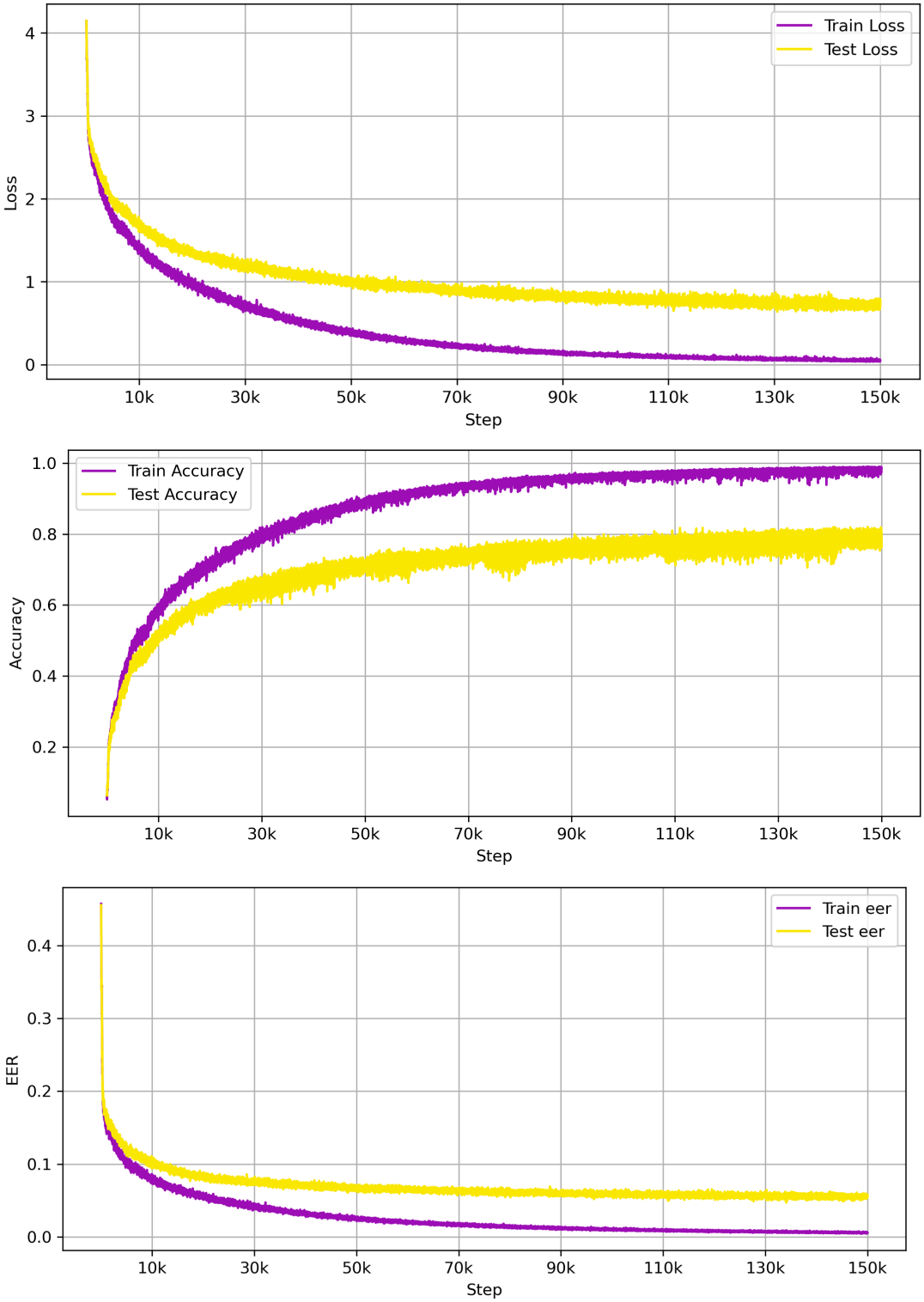

Figure 5.2: Evolution of the loss, accuracy and equal error rate curves over the course of the 150, 000 training steps.

Table 1: Training Values at Step 150, 000.

Train Loss	Train Accuracy	Train EER
0.048	0.987	0.005 = 0.5 %
Test Loss	Test Accuracy	Test EER
0.725	0.793	0.057 = 5.7 %

Subsequently, after stopping the training, we loaded the trained model and ran 1, 000 test iterations to gather concluding statistics on the final model performance. Each iteration consisted of batches of 64 different speakers each with 10 utterances, i.e., each batch contained 640 utterances. The speakers included at these batches were randomly selected from the 292 distinct speakers from the testing subset, these speakers were not present during the training of the model. For each iteration, we computed the loss, accuracy score and equal error rate. These values were then averaged to obtain the final results from this study presented in the table bellow:

Table 2: Trained Model Testing Results.

Loss	Accuracy	EER
0.723	0.790	0.055 = 5.5 %

To further contextualize our results, we present additional visualizations for three different testing batches. Each of these testing batches consists of 50 utterances from 5 speakers, with 10 utterances per speaker. The reduced dimensions of these testing batches (5×10), compared to the batches used for the main test results presented in Table 2 (64×10), were chosen for ease of interpretation of the figure presented bellow. Plotting a similarity matrix of size $64 * 10 \times 10$ would make difficult to discern the scorings and draw meaningful conclusions, while a similarity matrix of size $5 * 10 \times 10$ allows for more visual interpretability.

The heatmap of the similarity matrix provides a visual representation of the pairwise similarity scores between utterance embeddings and speakers centroids. Each embedding, $\mathbf{e}_{ji}$, is associated with a row in the matrix, representing the i-th utterance of the j-th speaker. Each speaker centroid, $\mathbf{c}_k$, is associated with a column.

Brighter colors (closer to orange/yellow) indicate regions of high similarity scores. In these regions, the utterance embedding associated with that row is closer to the corresponding speaker centroid associated with that column, meaning that the speaker verification system would accept the verification of the claimed identity. That is, if speaker j claims their identity to be k, and the similarity score between the i-th utterance embedding $\mathbf{e}_{ji}$ and the k-th speaker centroid $\mathbf{c}_k$ (represented by the value in the corresponding row and the k-th column from the similarity matrix) is high (closer to orange/yellow), the system will accept the claim.

Conversely, darker colors (closer to purple/dark purple) represent lower similarity scores. These regions indicate that the utterance embedding associated to that row is far from the speaker centroid associated to that column. Consequently, the system would reject the claimed speaker identity. For instance, if speaker j claims their identity to be k, and the similarity score between

the i-th utterance embedding $\mathbf{e}_{ji}$ and the k-th speaker centroid $\mathbf{c}_k$ (represented by the value in the corresponding row and the k-th column from the similarity matrix) is low (closer to purple/dark purple), the system will reject the claim.

The two-dimensional embedding space projection provides visualization of the model's ability to generate clustered embeddings, while also illustrating how the similarity scores reflect the distances between embedding utterances in the embedding space.

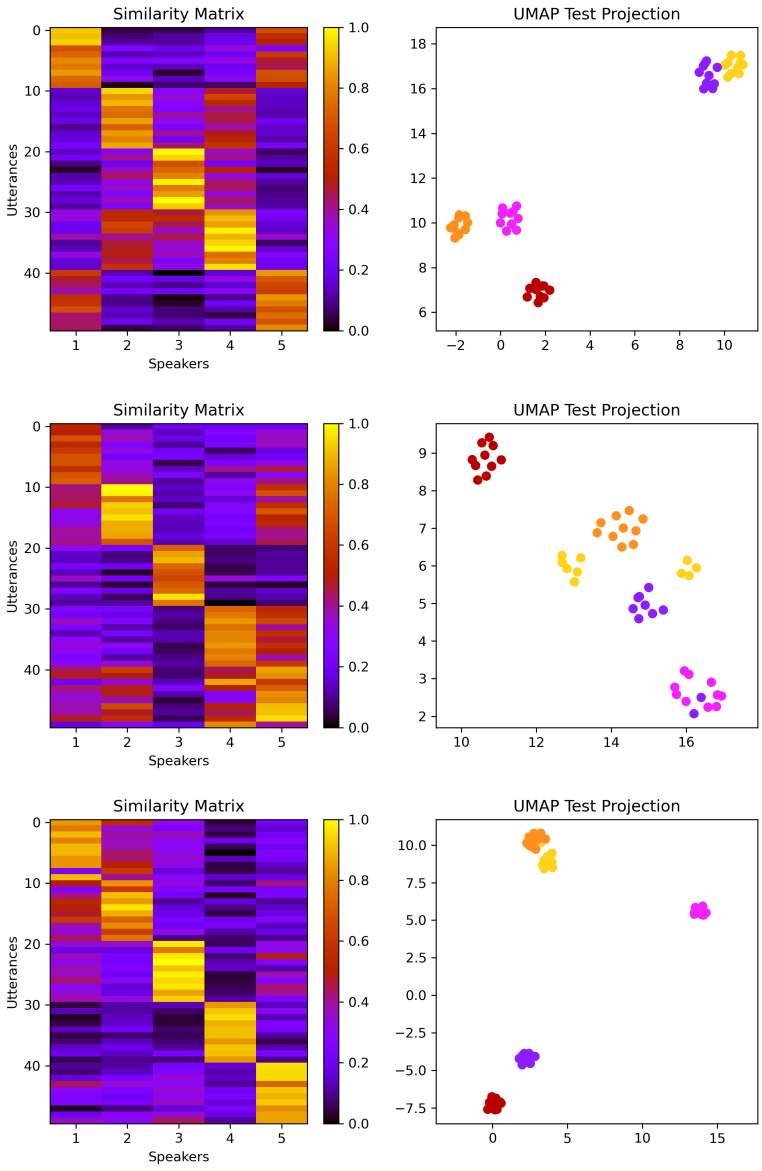

Figure 5.3: Similarity Matrix and Embedding Space two-dimensional Projections over three test batches.

The interpretation of the similarity matrix heatmap provides insights into the model's performance in speaker verification, providing a representation of the verification outcomes. Additionally, the two-dimensional embedding space projection complements this analysis by displaying the embeddings positions in the embedding space.

5.5 Conclusions

Our implementation effectively generates discriminative embeddings from utterances' log-Mel filterbank energy vectors, capturing the distinctive features of each speaker and enabling accurate differentiation between speakers.

By employing a single network architecture that emulates the three steps of the traditional speaker verification pipeline, our approach directly maps utterances' log-Mel filterbank energy vectors to their corresponding speaker embeddings while optimizing a unified loss function for the entire architecture, ensuring a cohesive learning process for both speaker modeling and verification.

Although we cannot directly compare the loss and accuracy scores of our model with the implementation from [12], as they were not provided by the authors, the study reported an EER of 3.55% on text-independent speaker verification. It is worth remarking that the dataset used by [12] consisted of $36M$ utterances from $18K$ speakers, whereas our work was conducted a more limited dataset of $1,166$ speakers. Despite the difference in dataset magnitudes, our architecture achieved an EER of 5.5%, which compares favorably to [12]. This demonstrates the effectiveness of our approach, which simplifies the speaker verification process and reduces the complexity associated with traditional multi-step pipelines.

References

[1] Robin M. Schmidt. Recurrent Neural Networks (RNNs): A gentle Introduction and Overview, 2019.

[2] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *ICML (3)*, volume 28 of *JMLR Proceedings*. JMLR.org, 2013.

[3] Jingzhao Zhang, Tianxing He, Suvrit Sra, and Ali Jadbabaie. Why gradient clipping accelerates training: A theoretical justification for adaptivity, 2020.

[4] Xiangyi Chen, Zhiwei Steven Wu, and Mingyi Hong. Understanding gradient clipping in private sgd: A geometric perspective, 2021.

[5] Klaus Greff, Rupesh K. Srivastava, Jan Koutnik, Bas R. Steunebrink, and Jurgen Schmidhuber. LSTM: A Search Space Odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 2016.

[6] Erik McDermott Ignacio Lopez Moreno Javier Gonzales-Dominguez Ehsan Variani, Xin Lei. Deep Neural Networks for Small Footprint Text-Dependent Speaker Verification. *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2014.

[7] Tara N. Sainath Mirkó Visontai-Raziel Alvarez Carolina Parada Yu-hsin Chen, Ignacio Lopez-Moreno. Locally-connected and Convolutional Neural Networks for Small Footprint Speaker Recognition. *Conference of the International Speech Communication Association (INTERSPEECH)*, 2015.

[8] V. Klema and A. Laub. The singular value decomposition: Its computation and some applications. *IEEE Transactions on Automatic Control*, 25(2):164–176, 1980.

[9] Carla D. Martin and Mason A. Porter. The extraordinary svd, 2012.

[10] Samy Bengio Naom Shazeer Georg Heigold, Ignacio Moreno. End-to-End Text-Dependent Speaker Verification. *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2016.

[11] Hasim Sak, Andrew W. Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *INTERSPEECH*, pages 338–342, 2014.

[12] Alan Papir Ignacio Lopez Moreno Li Wan, Quan Wang. Generalized End-to-End Loss for Speaker Verification. *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2020.

[13] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: An asr corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210, 2015.

[14] Jyh-Min Cheng and Hsiao-Chuan Wang. A method of estimating the equal error rate for automatic speaker verification. In *2004 International Symposium on Chinese Spoken Language Processing*, pages 285–288, 2004.

[15] Richard P. Brent. An algorithm with guaranteed convergence for finding a zero of a function. *Comput. J.*, 14:422–425, 1971.

[16] Abien Fred Agarap. Deep Learning using Rectified Linear Units (ReLU), 2019.

[17] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, 2014. Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[18] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[19] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction, 2020.

[20] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

[21] Johan Schalkwyk, Doug Beeferman, Françoise Beaufays, Bill Byrne, Ciprian Chelba, Mike Cohen, Maryam Kamvar, and Brian Strope. *"Your Word is my Command": Google Search by Voice: A Case Study*, pages 61–90. Springer US, 08 2010.

[22] Ronald W. Schafer Lawrence R. Rabiner. *Introduction to Digital Speech Processing*. now Publishers Inc., 2007.

[23] J. John R. Deller, J. H. L. Hansen, and J. G. Proakis. *Discrete-Time Processing of Speech Signals*. Wiley-IEEE Press, 2nd ed. edition, 1999.

[24] Joseph Picone. Fundamentals of Speech Recognition: A Short Course. Technical report, Institute for Signal and Information Processing, Department of Electrical and Computer Engineering, Mississippi State University, 1996.