POLITECNICO DI TORINO
Corso di Laurea Magistrale
in Ingegneria del Cinema
e dei Mezzi di Comunicazione

TESI DI LAUREA MAGISTRALE

Neural Networks

for Language and Speaker Recognition

Cinzia Ferrero

Relatori | **prof. Pietro Laface**
Sandro Cumani

Settembre 2018

I'm sorry, my responses are limited.
You must ask the right questions.

Isaac Asimov | *I, Robot*

Abstract

In this thesis we consider two major fields in which machine learning is applied to human voice: language and speaker recognition. For both we provide an overview of the whole recognition chain, from the acoustic signal to the classifier, and we present applications of neural networks for classification.

In particular, since language and speaker systems share some techniques, the initial part of this thesis is an overview of the common approaches to the recognition problem. We first analyze state-of-the-art techniques to pre-process the speech signal, to extract its relevant features and to represent them by means of statistical models. We then focus on the working principles of neural networks, and on several different methods for their training and regularization.

Within the context of language recognition, we propose a neural network architecture to classify i-vectors, which are modelled on the basis of the recently presented Stacked Bottleneck Neural Network (SBN) features. Comparing this solution to a Gaussian Linear classifier, we show that the former performs lightly better than the latter.

For speaker recognition, we focus on the pairwise approach, which consists in establishing whether a pair of i-vectors belongs to the same-speaker or to the different-speaker class. In particular, we present a Siamese neural network architecture, which performs the binary classification of a pair of i-vectors. We propose different techniques to share its layer weights. The obtained architecture improves the scores of a previously proposed Siamese network, but it does not provide better performance with respect to systems that implement Probabilistic Linear Discriminant Analysis (PLDA) or Pairwise Support Vector Machines (PSVM) techniques.

To my family

Acknowledgements

I would like to thank professor Pietro Laface and Sandro Cumani for their guidance. Thanks to them I learnt many concepts and techniques concerning language and speaker recognition.

Thanks to my parents for always supporting me unconditionally. Thanks to Fabio for always encouraging me and for helping me to design the graphic of this work. Thanks to Adam for being near me and for inspiring me to reach higher goals. Thanks to Arek, Margherita, my grandparents and all my family for supporting me.

Thanks to Luana, my friends and colleagues for sharing many moments and experiences during these years of studies.

Contents

Chapter 1
Introduction

In the last years a growing interest towards speech technologies has pervaded both industries and scientific community. Virtual assistants, conversational human-machine interfaces, more human-like speech synthesis, audio surveillance, voice biometric authentication, audio indexing, all of these are examples that audio is and will become a substantial segment of technology. One of the reason of this success is due to machine learning and to the possibilities that it offers to build more and more intelligent systems. Furthermore, the increasing availability of data and computational power allows training more effective and accurate systems.

Three main fields on which research is focusing are speech, speaker and language recognition. Speech recognition develops automatic systems that enable the recognition of spoken language, and its transcription to text. Since it is a required step for voice user interfaces, this technology has many commercial applications, like mobile virtual assistants, audio typing and domotics.

Speaker recognition is the automatic process to predict the identity of the speaker of a given utterance. This technology can be adopted in a wide range of applications: authentication procedures, forensic activities, telephone-based services, and speaker diarization.

Language Recognition is the automatic process that tries to identify the language spoken in a given utterance. Many applications are possible for this technology, like spoken language translation, emergency call routing, surveillance and security information distillation or as front-end for language-dependent speech recognizers.

In this thesis, we focus on language and speaker recognition. For both we provide a good overview of the whole recognition chain, starting from the processing of the audio signal up to the actual classification, and the final answer regarding the identity of the person or the language. Since language and speaker recognition share some problems, and rely on similar techniques, the initial part of this thesis presents the common approaches to the recognition problem. In particular, we describe state-of-the-art methods to pre-process the speech signal, to extract its relevant features and to represent them by means of statistical models. Given audio samples of different duration, these techniques are able to represent them with low-dimensional, fixed dimension, vectors, called i-vectors.

The main objective of this thesis is to apply the artificial neural network framework to the classification problem of both speaker and language recognition. A neural network is a computing architecture including many simple processing elements, which are highly interconnected. Providing input data and the expected outputs, the network is able to learn patterns by adjusting the weights of the connections between its elements.

Within the context of language recognition, we will introduce a recent technique to extract features from the audio signal: the Stacked Bottleneck Neural Network (SBN) framework. We will then propose a neural network architecture to classify these features, and to identify the language of the given spoken utterances.

For speaker recognition, we focus on a pairwise approach for classification, which consists in establishing whether a pair of i-vectors belongs to the same-speaker or to the different-speaker class. Thus, instead of performing a multiclass classification (1-to-many-speakers), we perform binary classification of i-vector pairs. We will introduce state-of-the-art models, which execute pairwise classification, namely Probabilistic Linear Discriminant Analysis (PLDA) or Pairwise Support Vector Machines (PSVM). Furthermore, we will present a Siamese neural network architecture, which performs the binary classification of pairs of i-vectors. This architecture has two sub-networks, that share the weights, and learns an internal representation of the i-vectors, that aims at enhancing speaker discrimination. An objective cost function between these representations is minimized, and after training, the Siamese network is able to return a classification prediction for an input pair of i-vectors. We will propose different architectures and techniques to share the weights of the network.

The outline of this thesis is the following:

- Chapter 2 describes the preliminary processing of the audio signal and the techniques to extract the features, enhancing meaningful information, and discarding the redundant one.
- Chapter 3 presents statistical techniques to model the distribution of the acoustic features. In particular, the Gaussian Mixture Model will be introduced together with methods for creating background models, and for compensating speaker and channel variability.
- Chapter 4 introduces the working principles of neural networks and several different methods for their training and regularization.
- Chapter 5 deals with language recognition. It describes the SBN features, and presents the experimental results obtained for classification with different neural network architectures.
- Chapter 6 focuses on speaker recognition. It introduces state-of-the-art pairwise classifiers and the Siamese neural network architecture. Then it presents the experimental results obtained with different Siamese architectures.

Chapter 2
Feature extraction

For any ASR system, the starting point to analyze speech utterances is a front-end processing that is used to transform the acoustic signal in a set of features. It is fundamental that this set is a good representation of the signal. So the features are chosen with the purpose of emphasizing the discriminative information and suppress the statistical redundancies or the useless information [1].

For example in the case of speaker recognition, an ideal feature set [2] would:

- have large variability between different speakers, but small variability within the same speaker
- be robust against noise and distortion
- contain frequent and natural characteristics of the speech
- be easy to evaluate
- be difficult to fake
- not be affected by variations of the speaker's voice due to health conditions or the fact that a long time has passed
- have a relative small number of features

If the last point is not respected, one falls into the problem known as the curse of dimensionality. In fact sometimes traditional statistical models (e.g. Gaussian mixture model) can't handle data with high-dimensionality [2]. Furthermore if the number of features grows, the number of training samples to reliably estimate the density also grows, but potentially in an exponential way. So the condition to have a low number of features is fundamental.

Front-end processing generally consists of three steps.

The first step for feature extraction is to digitize the signal and to convert it from a continuous signal in time and amplitude to a discrete signal in time and amplitude as well. After this, the signal can be processed by a machine.

Next, sspeech activity detection is required to remove the portions of the signal of silence. In these portions the value of the audio signal is not equal to zero because of noise, so some sort of strategy is needed to distinguish speech and non-speech signal.

In the third step the relevant information for our task has to be selected and enhanced. To handle the fact that the audio signal is not stationary,

a short-term analysis must be performed. Then the final features are extracted to form a vector of coefficients. The sample utterances converted to feature vectors will then be used for model training, which will be dealt with in the next chapter.

Often a post-processing step is added to the feature extraction chain, to perform a preliminary channel compensation and noise attenuation. Since the recognition system can operate with utterances coming from different channels, it is important to attenuate the channel effects, that typically modify the spectrum (e.g. limiting the band or changing the shape). For these reasons, different techniques can be applied to make the system more robust.

2.1 | Analog-to-digital conversion

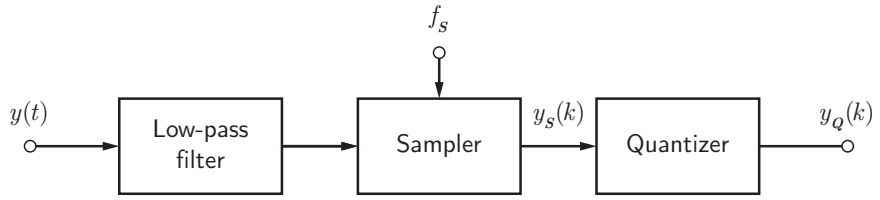
The systems that digitalize audio signals use time sampling and amplitude quantization to encode an analog waveform with continuous values to a discrete signal with finite amplitude values in time.

The Nyquist theorem provides sufficient conditions that allow the original waveform to be reconstructed from a set of discrete samples without any loss of information. In particular, the theorem states that a signal can be recovered without loss of any information provided that the sampling frequency is at least twice the highest signal frequency

In practice to respect the theorem, the input signal has to be low-pass filtered, in order to have the maximum frequency equals to half the chosen sampling frequency. Since the human voice mainly includes frequencies that are below 4 KHz, telephone applications sample audio signals at a frequency of 8KHz. In this way there are no consistent losses of intelligibility, even if there are some losses of naturalness.

Unlike the sampling process, quantization introduces a measuring error. With uniform quantization, the continuous possible amplitude values of a time sample are mapped to a finite number of quantization levels of the same size. As with any analog measurement, since the resolution of the system is finite, the accuracy is limited and an error is introduced.

< Figure 2.1: Analog-to-Digital converter schema



Hereafter we analyze more in detail the various steps to digitalize a speech signal.

Sampling

Sampling can be written as the multiplication of the input signal by a periodic impulse train:

$$y_s(k) = \sum_k [y(kt_s)\delta(t - kt_s)] \qquad < (2.1)$$

where $y(t)$ is the input signal, $y_s(k)$ is the sampled signal and the impulses δ are distant from each other of the sampling interval t_s.

Since the Fourier transform of a periodic train of impulses is still a periodic train of impulses

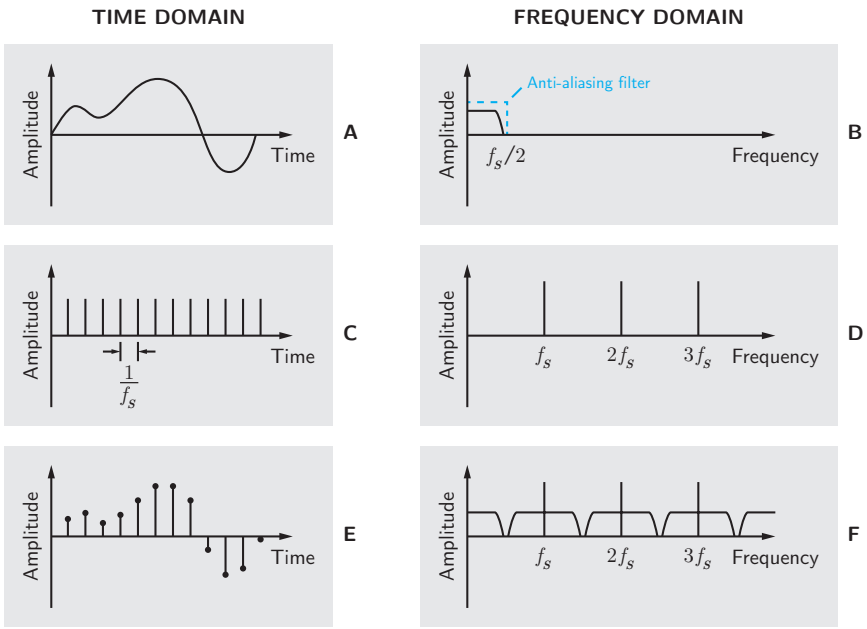
$$\sum_k \delta(t - kt_s) \xrightarrow{FT} \Omega \sum_p \delta(t - p\Omega) \quad \text{with} \quad \Omega = \frac{2\pi}{t_s} \qquad < (2.2)$$

and the multiplication in the time domain corresponds to the convolution in the frequency domain, we get that the Fourier transform of the sampled signal is

$$Y_s(\omega) = \frac{1}{t_s} \sum_k Y\left(\omega + \frac{2\pi k}{t_s}\right) \qquad < (2.3)$$

where $Y(\omega)$ is the Fourier transform of $y(t)$ and $Y_s(\omega)$ is the Fourier transform of $y_s(t)$.

If $Y(\omega)$ is bandlimited to $1/2f_s$, the replicas do not overlap and so there is no aliasing. In this work we will consider a sampling frequency equal to 8 kHz.



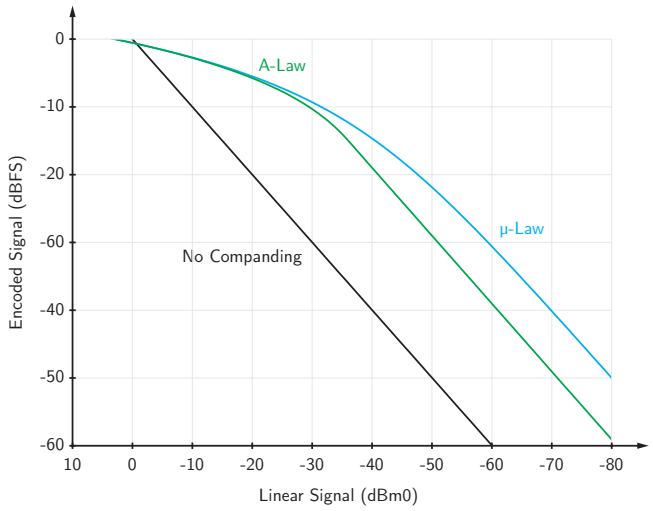
< Figure 2.2: Schema to illustrate the process of sampling a bandlimited signal On the left there are the signals in the time domain and on the right in the frequency domain.
A) Input signal
B) Spectrum of the input signal
C) Sampling signal
D) Spectrum of the sampling signal
E) Sampled input signal
F) Spectrum of the sampled input signal

Quantization

The audio signal amplitudes are not uniformly distributed, but statistically there are more samples with low amplitude. So a uniform quantization would not perform well and a logarithmic quantization is preferred. In practice we realize the process of companding, that is to compress the dynamic range of the signal and then linear-quantize it. To simplify the computation and to solve the problem that the logarithm is not defined in zero, functions like A-law (for American communication nets) and μ-law (for European communication nets) are applied [3]. These are piece-wise linear approximations of the logarithm.

By applying this sort of quantization, samples with low amplitude are represented with greater precision (more bits). Furthermore the signal-to-noise ratio (SNR) is less sensitive to changes of the input signal dynamic range.

In particular telephone speech samples are represented on 8 bits. The μ-law and A-law algorithms map 13-14 bit linear samples to 8-bit logarithmic samples. Thus, we get a 64 kbit/s bitstream for a signal sampled at 8 kHz.



< Figure 2.3: Companding of μ-law and A-law functions

Filtering

From sampling and quantization we obtain the digitized signal $y(k)$, which is characterized by a drop in the power at higher frequencies. This property is caused by the glottal voice source [2]. Therefore the signal spectrum is flatten by means of a first order pre-emphasis filter with the following transfer function [4]

$$H(z) = 1 - az^{-1} \quad 0.9 < a < 1.0 \tag{2.4}$$

< (2.4)

where usually a is chosen equals to 0.95.

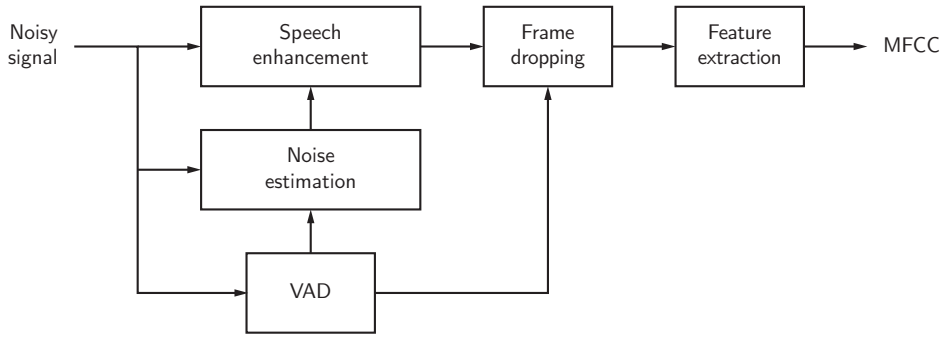
In time domain, the output of the pre-emphasizer is

$$\hat{y}(k) = y(k) - ay(k-1) \tag{2.5}$$

< (2.5)

2.2 | Voice activity detection

Voice activity detection (VAD) is a technique used to discriminate the segments of an audio signal that contain speech from the ones that contain only noise. This is especially important for signals recorded in noisy environments, because VAD can enhance the quality of the speech signal identifying the segments of interest. Usually the segments that according to the VAD decision are not speech segments are ignored.



< Figure 2.4: Voice activity detection is used to identify the non-speech frames that are dropped

One of the most simple solutions is to use the signal energy to locate the speech segments [2]. In particular, the signal is divided in frames, the frame energy is computed, and the VAD decision threshold is set close to the maximum energy, considering a tolerance. The disadvantage of this approach is that the whole utterance has to be processed to set the decision threshold.

Instead, a real-time solution is the Long-Term Spectral Divergence (LTSD) technique [5]. The most relevant information to locate the speech is assumed to remain on the time-varying signal spectrum magnitude. The algorithm is based on the estimation of the Long-Term Spectral Envelope (LTSE) and on the computation of the LSTD between the speech and the noise, which is used to evaluate the decision rule.

Energy-based VAD is very popular due to its simplicity, but it's also sensitive to environmental noise. Some form of pre-process to enhance the signal can be performed, however in [6] a more robust solution has been proposed, which is described by the following steps:

- Mel-Frequency Cepstral Coefficients (Section 2.3) are extracted from the original noisy signal.
- The signal is enhanced to increase the energy contrast between the speech and non-speech segments. In particular, the spectral subtraction technique is used, whose details can be found in [6].
- The frame energies of the enhanced signal are computed.
- The lowest and highest energy frames are selected, considering a fixed percentage. The former are considered the non-speech frames,

and the latter the speech ones. Using this method, two subsets of reliable labelled frames are built.

- Two Gaussian Mixture Models (Section 3.1) are trained using the MFCCs of the subsets' frames. Thus, the parameters of the speech and non-speech models are obtained.
- Computing the likelihood ratio using the models, all the frames are labelled as speech or non-speech. In particular, given the speech model parameters λ^{speech}, the non-speech model parameters $\lambda^{nonspeech}$, and a feature vector x_t, the likelihood ratio is

$$\log p\left(x_t|\lambda^{speech}\right) \geq \log p\left(x_t|\lambda^{nonspeech}\right) \quad < (2.6)$$

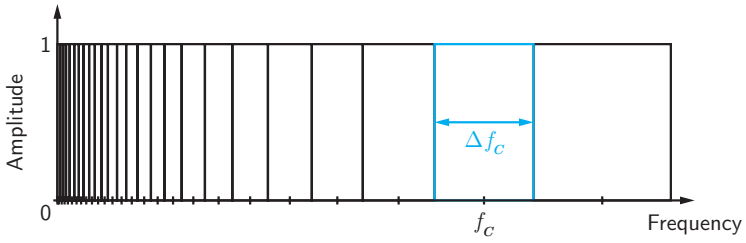
2.3 | Mel-Frequency Cepstral Coefficients

After the conversion of the signal from analog to digital and the removal of non-speech portions of the signal, the next phase consist of extracting the distinctive features.

The Mel-Frequency Cepstral Coefficients (MFCC) were introduced by Davis and Mermelstein in [1] and they are one of the most common representations of the acoustic signal in speech recognition. In particular MFCCs provide a short-term representation of the power spectrum of the acoustic signal. They combine the advantages of the *cepstrum* with a frequency scale based on ear's critical bands.

Bogert et al. defined the cepstrum as the inverse Fourier transform of the log magnitude spectrum of a signal and it was developed to separate convoluted signals [7]. Thus by applying the cepstrum operator it is possible to separate the excitation signal from the vocal tract signal in the speech production model. In fact, the speech signal is given by the excitation signal, produced by the lungs and approximated by a white noise, passed through the vocal tract, which according to its shape assigns various formants to the signal spectrum.

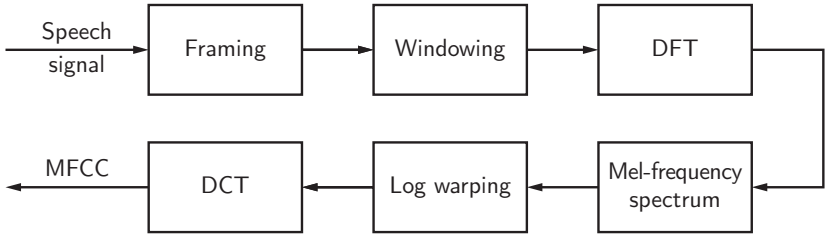
Human ear can perceive a range of frequencies from 20 Hz to 20 kHz, but the resolution in this range is not uniform. At low frequencies, the ear can distinguish differences of frequency more easily than at high frequencies. This non-uniform frequency analysis performed by the basilar membrane can be modelled with a set of bandpass filters, with narrower bands for low frequencies and wider bands for high frequencies. These bands, called critical bands, are used in psychoacoustic to quantify the ability of the human ear to distinguish between individual frequency tones. In particular the Mel scale is a perceptual scale of pitches judged by listeners to be equal in distance from one another.



< Figure 2.5: Representation of the bandpass filters based on critical bands

2.3.1 MFCCs computation

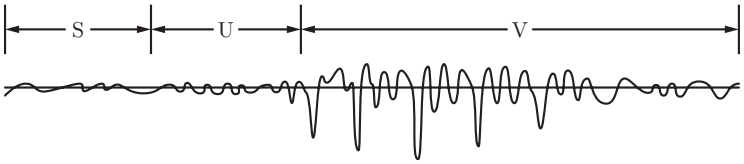
The system to extract MFCCs from the signal can be outlined as a chain of different processes which are illustrated by the Figure 2.6 and described more in detail in the following sections.



< Figure 2.6: System to compute MFCCs

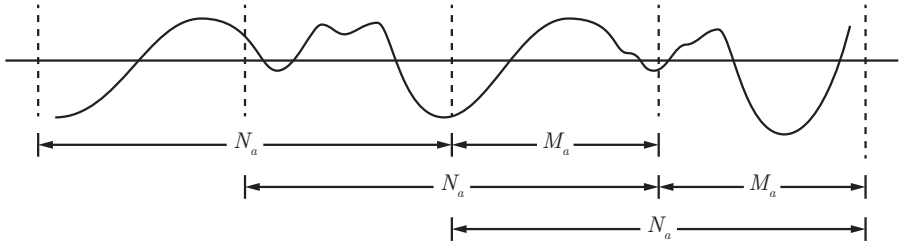
Framing

The speech signal is a slowly time varying signal [4]: if it is analyzed in a short period of time (between 5 and 100 ms), it is quasi stationary; but if we consider a longer period of time (on the order of 1/5 sec or more), the signal depends on the speech sounds being spoken.



< Figure 2.7: Waveform of a speech signal composed by a part of silence (S), a part of unvoiced signal (U) and a part of voiced signal (V)

Therefore, the first step towards MFCC extraction consists in splitting the acoustic signals into frames. These are usually 10 ms long. In this period of time the signal can be considered stationary.



< Figure 2.8: Splitting of the audio signal into overlapping frames

The framing operation can be written as

$$x_t(n) = \hat{y}(M_a t + n) \quad 0 \leq n \leq N_a - 1, \quad 0 \leq t \leq T - 1 \qquad < (2.7)$$

where N_a is the frame size (i.e. the number of samples in a frame), M_a is the size of the shift, T is the number of frames within the entire signal. So for example the first frame, $x_0(n)$, includes the speech samples $\hat{y}(0), \hat{y}(1), ..., \hat{y}(N_a - 1)$.

Windowing

A framing process like the one described above causes distortions of the spectrum due to the discontinuities introduced between two successive frames. To extract a frame this way is equivalent to the multiplication of the signal by a rectangular window. Since the Fourier transform of a rectangular window is a sinc function, in the spectrum of the framed signal are introduced non-zero values, commonly called spectral leakage, around the frequencies of the original signal. Leakage can prevent from distinguishing two neighbouring frequencies or can also obscure the weaker ones.

To solve this problem we can use a window function $w(n)$ that minimizes the discontinuities between successive frames:

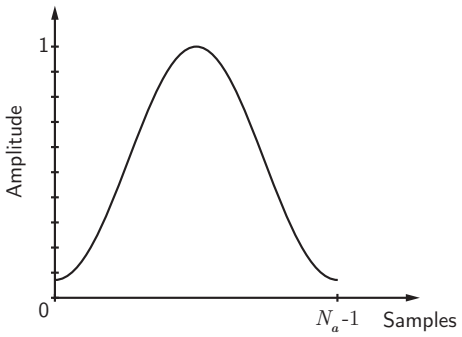
$$\hat{x}_t(n) = x_t(n)w(n) \quad 0 \leq n \leq N_a - 1, \quad 0 \leq t \leq T - 1 \qquad < (2.8)$$

One of the most used window function is the Hamming window

$$w(n) = a_0 - (1 - a_0) \cdot cos\left(\frac{2\pi n}{N_a - 1}\right) \quad 0 \leq n \leq N_a - 1 \qquad < (2.9)$$

Typically the value a_0 is chosen equal to 0.54.

< Figure 2.9: Hamming window



Furthermore to avoid losing the information of the samples near the borders frame, the frames are overlapped, i.e. the shift between consecutive frames is smaller than the frame length. In this case it is chosen $M_a = N_a/2$. In this way, the samples that are near the end of the frame will be in the middle of the next frame and their information will not be lost.

Fourier Transform

The next step to compute the audio features is to calculate the Discrete Fourier Transform (DFT) of the windowed frame. This can be done using the Fast Fourier Transform (FFT), that decomposes the signal into its frequency components.

$$X_f(j) = \mathcal{F}(\hat{x}_t(n))(j) \quad 0 \leq n, j \leq N_a - 1, \quad 0 \leq t, f \leq T - 1 \tag{2.10}$$

Usually the phase of the Fourier Transform is ignored, because it is believed to be of little perceptual importance [2]. Whereas the envelop of the magnitude spectrum is the most informative part of the spectrum because it contains the information about the resonance properties of the vocal tract.

Mel-frequency spectrum

Psychoacoustic studies show that the human auditory system can be effectively modeled as a filter bank. Therefore the most simple model to extract features from the spectrum is a bandpass filter bank to get the energy of neighbouring frequencies. Furthermore in order to consider the human critical bands, the filter bank's bands are chosen according to the Mel scale. In this way the low frequencies are represented with higher resolution because a larger number of narrower filters is assigned to them.

In most implementations the shape of the filters is triangular.

There are different formulas to convert frequencies from the hertz scale to the mel scale. A popular one is the following

$$m = 2595 \log_{10}\left(1 + \frac{f}{700}\right) \tag{2.11}$$

where f denotes the real frequency (hertz) and m denotes the perceived frequency (mels). There are also other approximations that consider linear and logarithmic segments with 1000 Hz as turning point.

Given the mel-frequency bands, the DFT values can be grouped together in these bands and weighted by the triangular function. The energy for each frequency band can be calculated as

$$E_i(f) = \sum_{j=L_i}^{H_i} |X_f(j)|^2 \quad 1 \leq i \leq N_f \tag{2.12}$$

where $E_i(f)$ is the energy of the i-th band, L_i and H_i are respectively its lower and upper bound and N_f is the number of bands (e.g. usually $N_f = 13$).

The mel-frequency spectrum is defined as [7]

$$\mathrm{MF}_i(f) = \frac{1}{A_i} \sum_{j=L_i}^{H_i} |V_i(j) X_f(j)|^2 \tag{2.13}$$

where $V_i(j)$ is the triangular weighting function for the i-th filter and

$$A_i = \sum_{j=L_i}^{H_i} |V_i(j)|^2 \qquad \text{(2.14)}$$

is a normalizing factor. This is needed to obtain a flat mel-spectrum from a flat input Fourier spectrum.

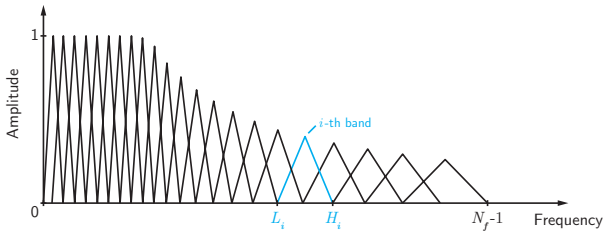


Figure 2.10: Mel spectrum

Log warping

Following the computation of the Mel-frequency power spectrum, a logarithmic transformation is applied to mimic the human perception of loudness.

Discrete Cosine Transform

Finally to compute the MFCCs we apply the inverse DFT to the log Mel-frequency spectrum. Since applying the IDFT is in general complex, the Discrete Cosine Transform is performed instead.

Therefore the p-th MFCC for frame k is given by [7]

$$\text{mfcc}_p(k) = \frac{1}{N_f} \sum_{j=1}^{N_f} \log(\text{MF}_j(k)) \cos\left[\frac{2\pi}{N_f}\left(j + \frac{1}{2}\right) p\right] \qquad \text{(2.15)}$$

If we don't use filters with triangular shape to evaluate the energy of the mel-frequency bands, we obtain a simpler formula

$$C_p(k) = \sum_{j=1}^{N_f} \log(E_j(k)) \cos\left[p\left(j - \frac{1}{2}\right)\frac{\pi}{N_f}\right] \quad 0 \le p \le N_{\text{mfcc}} - 1 \qquad \text{(2.16)}$$

Cepstral parameters have decreasing variance as their indices grow, so high index parameters carry less information and they can be discarded. Usually, the number of cepstral parameters N_{mfcc} used is between 12 and 24. Furthermore the cepstral parameter $C_0(k)$ is discarded because it contains the same information given by $E(k)$, the total energy of the frame

$$E(k) = \sum_{j=1}^{N_f} E_j(k) \qquad \text{(2.17)}$$

2.3.2 MFCCs and derivatives

The processing described above doesn't include any time evolution information in the MFCCs. To represent the dynamic nature of speech, cepstral derivatives are taken into account. In particular the first order derivative of cepstral coefficients, called Delta coefficients, and the second order derivative, called Delta-Delta coefficients, are included in the feature set.

These parameters can be calculated through an approximation of the temporal derivative of MFCCs. One possibility is to use a polynomial expression over a certain number of successive frames.

$$\Delta\bar{C}_p(k) = G \sum_{j=-N}^{N} j\bar{C}_p(k-j) \quad 1 \leq p \leq N_{\text{mfcc}} \tag{2.18}$$

where G is a gain used to obtain a similar variance between the set of MFCCs and the set of differential parameters and N is half the size of the window used to approximate the derivative.

With a similar procedure, the differential energy can be computed as

$$\Delta E(k) = \sum_{j=-N}^{N} jE(k-j) \tag{2.19}$$

The second order derivative can be evaluated in the same way.

Finally the set of mel-frequency cepstral coefficients and their first and second order derivatives can be combined to form the features vector

$$\begin{aligned} O_t = &\{\bar{C}_1(t), ..., \bar{C}_{N_{\text{mfcc}}}(t), \Delta\bar{C}_1(t), ..., \Delta\bar{C}_{N_{\text{mfcc}}}(t), \\ &\Delta\Delta\bar{C}_1(t), ..., \Delta\Delta\bar{C}_{N_{\text{mfcc}}}(t), E(t), \Delta E(t), \Delta\Delta E(t)\} \end{aligned} \tag{2.20}$$

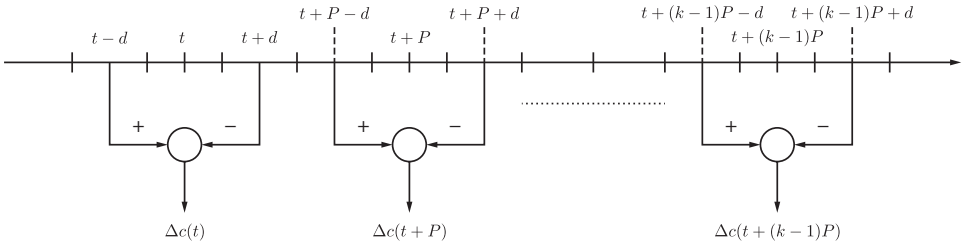
2.4 | Shifted Delta Coefficients

MFCCs allow capturing the short-term speech dynamics and give good results for speaker recognition. Shifted Delta Cepstral (SDC) [8][9] are more useful, instead, for language recognition because they allow capturing the dynamics of a wider time interval. This characteristics allows SDC features to improve the performance of the model (Chapter 3).

In particular, SDC feature vectors are obtained by stacking delta cepstra evaluated in different speech frames. The SDC vector is characterized by 4 parameters: N is the number of cepstral coefficients computer for each frame, d is the time delay for delta computations, k is the number of blocks whose delta coefficients are stacked, and P is the time shift between consecutive blocks (Figure 2.11). Given time t, the SDC coefficients are computed as follows

$$\Delta c_j(i,t) = \bar{C}_j(t + iP + d) - \bar{C}_j(t + iP - d) \qquad < (2.21)$$

where $\bar{C}_j$ is the j-th coefficient at time t.



< Figure 2.11: Schema of the computation of the SDC feature vector

Chapter 3
Statistical modeling

In order to model a certain language (or speaker), we use statistical models that try to describe the distribution of the acoustic features, extracted from audio samples of this language (or speaker). Therefore, the feature vector can be interpreted as a realization of a random variable. For each language (or speaker) we estimate a model which describes accurately the realizations. In particular, during the training the model parameters are evaluated by means of a sufficient wide set of audio samples. One of the most popular models is the Gaussian Mixture Model which we will consider in this work.

We are interested in computing the likelihood ratio between the hypothesis that an utterance is from the considered language (or speaker), which is the target, and the hypothesis that it is of another language (or speaker), which is the non-target. For this reason it is important not only to model correctly the target, but also to be able to model the non-target population. The proposed solution is to estimate a generic target-independent model, called Universal Background Model. Then the UBM is adapted to the specific target models.

Furthermore we will introduce some techniques to compensate for the effects due to variabilities that are not language (or speaker)specific, like channel effects. Indeed these variabilities can have negative consequences on the recognition. In particular, we will consider the technique of Joint Factor Analysis, which has been successfully used in the past, and also the more recent i-vector technique.

3.1 | Gaussian Mixture Model

The Gaussian distribution, also known as normal distribution, is a commonly used model in many engineering and scientific applications, because it has very good computational properties, and it can approximate many real-world data. Furthermore by combining a sufficient number of Gaussian distributions with the right parameters, we obtain a Gaussian Mixture Model (GMM) is a stochastic model that allows accurately estimating

almost any distribution. Due to these important characteristics, GMM are widely used both in language and speaker recognition.

3.1.1 Gaussian mixture distribution

The probability density function for a single, Gaussian distributed, continuous random variable x can be written in the form

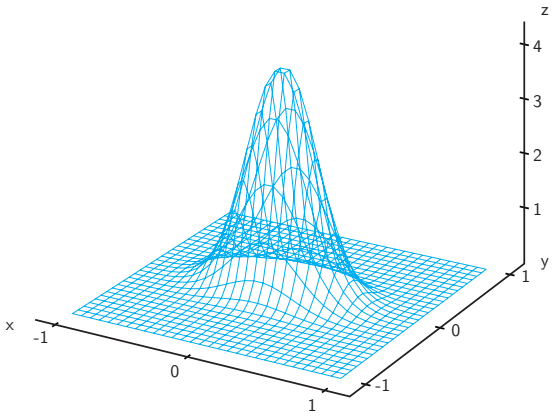
$$\mathcal{N}(x|\mu,\sigma^2) = \frac{1}{(2\pi\sigma^2)^{1/2}}\exp\left\{-\frac{1}{2\sigma^2}(x-\mu)^2\right\} \quad < (3.1)$$

where μ and σ^2 are its mean and variance, respectively.

For a D-dimensional normal random vector $\boldsymbol{x} = (x_1, x_2, ..., x_D)$, the multivariate Gaussian distribution can be defined as

$$\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu},\boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{D/2}|\boldsymbol{\Sigma}|^{1/2}}\exp\left\{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})\right\} \quad < (3.2)$$

where $\boldsymbol{\mu}$ is a D-dimensional mean vector and $\boldsymbol{\Sigma}$ is a $D \times D$ covariance matrix.



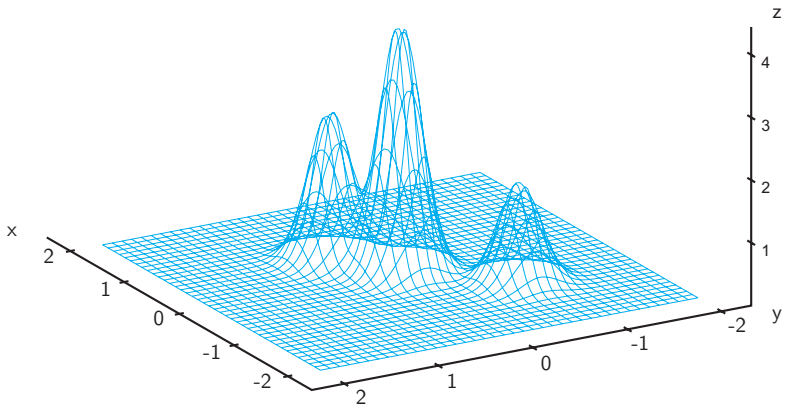
< Figure 3.1: Multivariate Gaussian distribution

Gaussian distributions are unimodal, however more complex distributions with multiple local maxima can be approximated by Gaussian mixtures, that are linear combinations of Gaussians and can be defined as

$$p(\boldsymbol{x}) = \sum_{k=1}^{K}\pi_k\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_k) \quad < (3.3)$$

where K is the number of Gaussians and π_k is the prior probability of picking the k-th Gaussian component. In particular each Gaussian $\mathcal{N}(\boldsymbol{x}|\boldsymbol{\mu}_k,\boldsymbol{\Sigma}_k)$ is a component of the mixture, characterized by a mean $\boldsymbol{\mu}_k$ and a covariance $\boldsymbol{\Sigma}_k$. The parameters π_k are called the mixture weights and must satisfy the following constraints:

$$\sum_{k=1}^{K}\pi_k = 1 \quad \text{and} \quad 0 \leq \pi_k \leq 1 \quad < (3.4)$$



< Figure 3.2: Gaussian mixture

With a sufficient number of Gaussians, characterized by the right means and covariances, and combined with the right coefficients, it is possible to approximate almost any distribution with an arbitrary accuracy [10]. Indeed from a set of samples $\boldsymbol{x} = \{x_1, ..., x_T\}$ of a random variable X, it is possible to approximate the p.d.f. of X with a Gaussian Mixture Model λ characterized by

$$p(\boldsymbol{x}|\lambda) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{x}|\mu_k, \Sigma_k) \qquad < (3.5)$$

In the case of speaker recognition, for example, x_t is the feature vector for a given frame at time t and the GMM are used to approximate the acoustic distribution of the speaker frames.

Usually the covariance of the GMM is assumed to be diagonal (i.e. the axis of the hyper-ellipsis are restricted in the same direction of the coordinate axes) to simplify the computation and to reduce overfitting of the training population [2]. Furthermore Gaussian mixtures with a sufficient number of components and diagonal covariances, are capable of modelling the correlations among feature vectors as well as full covariance GMMs with less Gaussians. Moreover the inaccuracy introduced by diagonal covariance matrices can be compensated considering more Gaussian components [11].

A possible approach to estimate the GMM parameters λ from a training data set is the Maximum Likelihood Estimation (MLE).

3.1.2 Maximum Likelihood Estimation

For a given parametric family of probability distributions, $p(\boldsymbol{x}|\lambda)$ with parameters λ, the ML method finds the values of λ that maximize the likelihood function $\mathcal{L}(\lambda|\boldsymbol{x})$, i.e. the values of the parameters that make the observed data the most probable [3][12]

$$\lambda_{ML} = \arg\max_{\lambda} \mathcal{L}(\lambda|\boldsymbol{x}) = \arg\max_{\lambda} \prod_{i=1}^{T} p(x_t|\lambda) \qquad < (3.6)$$

To simplify the problem, the logarithm of the likelihood is taken, so the products are transformed into sums

$$\lambda_{ML} = \arg\max_{\lambda} \sum_{i=1}^{T} \log p(x_t|\lambda) \tag{3.7}$$

In the case of the Gaussian mixture, the logarithm of the likelihood function is given by

$$\log p(\boldsymbol{x}|\lambda) = \sum_{i=1}^{T} \log \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(x_t|\mu_k, \Sigma_k) \right\} \tag{3.8}$$

However in this case there is no closed-form analytical solution [10]. The problem can be solved using iterative methods, for example the Expectation-Maximization (EM) algorithm. This is an iterative procedure that allows estimating the parameters of a probabilistic model in presence of latent variables.

Therefore, in our case we have to write a formulation of the Gaussian mixture that explicitly involves latent variables [10][13]. We introduce a K-dimensional binary random variable z that satisfies $z_k \in \{0, 1\}$ and $\sum_{k=1}^{K} z_k = 1$. This means that in a realization of z only one element z_k is equal to 1 and all the others are zero. Furthermore these realizations can be considered as K possible states of z that are mutually exclusive.

We can write the marginal distribution of z in terms of the mixing coefficients π_k as

$$p(z_k = 1) = \pi_k \tag{3.9}$$

and the distribution of z as

$$p(z) = \prod_{k=1}^{K} \pi_k^{z_k} \tag{3.10}$$

Since only one of the K possible states is active, the conditional distribution of x given the state $z_k = 1$ is

$$p(x|z_k = 1) = \mathcal{N}(x|\mu_k, \Sigma_k) \tag{3.11}$$

which can be rewritten as

$$p(x|z) = \prod_{k=1}^{K} \mathcal{N}(x|\mu_k, \Sigma_k)^{z_k} \tag{3.12}$$

The marginal distribution of x can be obtained by summing the joint distribution over all the possible states of z, with the joint distribution given by $p(z)p(x|z)$

$$p(x) = \sum_{z} p(x, z) = \sum_{z} p(z)p(x|z) \tag{3.13}$$

Using (3.10) and (3.12) we obtain

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k) \qquad < (3.14)$$

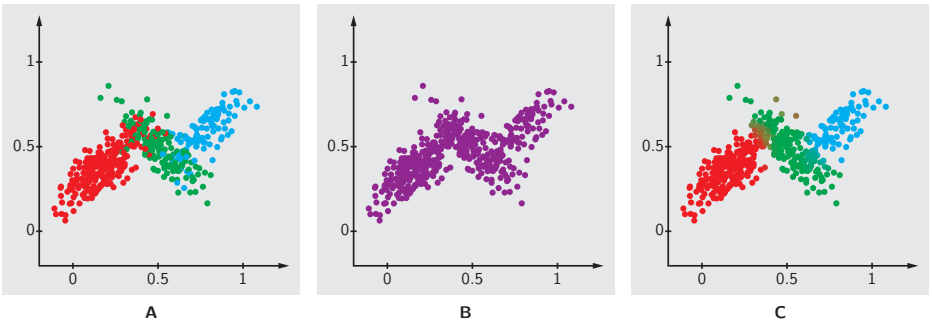
which has the same form of the Gaussian mixture model (3.5).

Therefore, we can interpret the GMM as a latent variable model where a latent vector z_t is associated to each different data point x_t.

The EM procedure requires the computation of the latent variable posterior distribution given the data. Applying Bayes theorem we have

$$\begin{aligned} \gamma_k(\lambda) = p(z_k = 1|x, \lambda) &= \frac{p(z_k = 1)p(x|z_k = 1)}{\sum_{j=1}^{K} p(z_j = 1)p(x|z_j = 1)} \\ &= \frac{\pi_k \mathcal{N}(x|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x|\mu_j, \Sigma_j)} \end{aligned} \qquad < (3.15)$$

We can see π_k as the prior probability of $z_k = 1$ and $\gamma_k(\lambda)$ as the corresponding posterior probability once we have observed x. The last one is also called *responsibility* because it gives the probability that the data point x was produced by the k-th Gaussian component.



< Figure 3.3: A) Samples from the joint distribution. The three states of z are represented by the colours red, green and blue B) Samples from the marginal distribution C) Same samples of B. The colours represent the value of the responsibilities associated to the data point

The Expectation-Maximization (EM) algorithm is an iterative technique that is used to compute the maximum likelihood estimates of model's parameters, with the model depending on latent variables (or missing data). In particular the EM algorithm alternates an Expectation (E) step and a Maximization (M) step, until the convergence of the parameters' values is reached:

- E step: given the data and the current parameters values, we estimate the missing data using the conditional expectation

$$\sum_t E_{z_{tk}|x_t,\lambda^c}[\log p(x_t, z_{tk}|\lambda)] \qquad < (3.16)$$

- M step: using the estimates of the previous step, the parameters are updated maximizing the likelihood function

$$\lambda^{c+1} = \arg\max_{\lambda} \left(\sum_{t} E_{z_{tk}|x_t,\lambda^c}[\log p(x_t, z_{tk}|\lambda)] \right) \quad < (3.17)$$

It can be demonstrated [10] that for Gaussian mixtures the conditions which have to be satisfied at a maximum of the log-likelihood function are the followings

$$\mu_k = \frac{1}{N_k} \sum_{t=1}^{T} \gamma_{tk}(\lambda) x_t \quad < (3.18)$$

$$\Sigma_k = \frac{1}{N_k} \sum_{t=1}^{T} \gamma_{tk}(\lambda)(x_t - \mu_k)(x_t - \mu_k)^{\mathrm{T}} \quad < (3.19)$$

$$\pi_k = \frac{N_k}{T} \quad < (3.20)$$

where

$$N_k = \sum_{t=1}^{T} \gamma_{tk}(\lambda) \quad < (3.21)$$

This can be interpreted as the expected number of points associated to the k-th Gaussian.

We can observe that the mean of the k-the Gaussian μ_k is calculated as the weighted mean of all the data points. The weight factor of each x_t is $\gamma_{tk}(\lambda)$, i.e. the responsibility of the k-th Gaussian to have generated the data point x_t. In the same way the covariance of the k-the Gaussian Σ_k is estimated over all the data points and each one of these has a weighting factor given by the responsibility. Finally the mixing weight π_k is the average responsibility of the k-th Gaussian for the generation of the data points.

Summarizing, the EM procedure for GMM parameter estimation consists of the following steps:

1. The means μ_k, covariances Σ_k and mixing weights π_k are initialized randomly and the log-likelihood is evaluated.
2. E step: we evaluate the responsibilities of each z_t associated to the data point x_t, using the current parameters values π^c, μ^c, Σ^c, with (3.15)

$$\gamma_{tk}(\lambda^c) = \frac{\pi_k^c \mathcal{N}(x_t|\mu_k^c, \Sigma_k^c)}{\sum_{j=1}^{K} \pi_j^c \mathcal{N}(x_t|\mu_j^c, \Sigma_j^c)} \quad < (3.22)$$

3. M step: using these responsibilities, the parameters are updated with (3.18), (3.19) and (3.20)

$$\pi_k = \frac{\sum_{t=1}^{T} \gamma_{tk}(\lambda^c)}{\sum_{t=1}^{T} \sum_{j=1}^{K} \gamma_{tj}(\lambda^c)} \quad < (3.23)$$

$$\mu_k = \frac{\sum_{t=1}^{T} \gamma_{tk}(\lambda^c) x_t}{\sum_{t=1}^{T} \gamma_{tk}(\lambda^c)} \quad < (3.24)$$

$$\Sigma_k = \frac{\sum_{t=1}^{T} \gamma_{tk}(\lambda^c)(x_t - \mu_k)(x_t - \mu_k)^{\mathrm{T}}}{\sum_{t=1}^{T} \gamma_{tk}(\lambda^c)} \quad < (3.25)$$

4. We evaluate the log-likelihood

$$\log p(\boldsymbol{x}|\lambda) = \sum_{t=1}^{T} \log \left\{ \sum_{k=1}^{K} \pi_k \mathcal{N}(x_t|\mu_k, \Sigma_k) \right\} \quad < (3.26)$$

   until the convergence of the likelihood. If the convergence criterion is not passed, we return to step 2.

Using these iterations we are able to estimate the parameters of a Gaussian mixture distribution of the features extracted from the audio input.

3.2 | Universal Background Model and MAP adaptation

Giving a test utterance as input to the front-end processing, the output we get is a set of feature vectors $\boldsymbol{x} = \{x_1, ..., x_T\}$ where x_t is the feature vector for the frame at time $t \in [1, ..., T]$. To model a language (or a speaker) we can then model the acoustic features assuming that these can be represented by a GMM distribution $p(\boldsymbol{x}|\lambda)$ with parameters $\lambda = \{\pi_k, \mu_k, \Sigma_k\}$.

The goal of a speaker verification or language detection system is to establish whether a test utterance belongs to an hypothesized speaker or language respectively or whether it doesn't. The hypothesized speaker and language are also called target. Thus, we define two hypothesis

H_S: $\boldsymbol{x}$ comes from the target
H_D: $\boldsymbol{x}$ does not come from the target

where S is for same and D is for different.

To make the decision we want to be able to compute the following likelihood ratio [14]

$$\frac{p(\boldsymbol{x}|H_S)}{p(\boldsymbol{x}|H_D)} \begin{cases} \geq \theta & \text{accept } H_S \\ < \theta & \text{reject } H_D \end{cases} \quad < (3.27)$$

where $p(\boldsymbol{x}|H_S)$ is the distribution of the acoustic features for the given target, and similarly $p(\boldsymbol{x}|H_D)$ is the distribution for the non-targets. θ is a predefined decision threshold.

The hypothesis H_S can be represented by the model λ_{H_S} that characterizes the target in the features space. Whereas H_D can be represented by the model λ_{H_D} that characterizes the non-targets in the same features space. Furthermore often the logarithm of the likelihood ratio is considered

$$\Lambda(\boldsymbol{x}) = \log p(\boldsymbol{x}|\lambda_{H_S}) - \log p(\boldsymbol{x}|\lambda_{H_D}) \quad < (3.28)$$

We already showed how to estimate the model and evaluate its log-likelihood for a set of feature vectors. Thus we can train the model λ_{H_S} using speech utterances from the target. Unlike this model, which is well defined, the definition of the non-target model λ_{H_D} is more complex because it has to potentially represent all the non-targets. There are two main approaches to deal with this problem. The first is to train a set of non-target models $\{\lambda_1, ..., \lambda_N\}$, called background models, and then to combine their likelihoods with some function $\mathscr{F}$ (e.g. average or maximum):

$$p(\boldsymbol{x}|\lambda_{H_D}) = \mathscr{F}(p(\boldsymbol{x}|\lambda_1), ..., p(\boldsymbol{x}|\lambda_N)) \quad < (3.29)$$

Many researches have been performed to understand which is the right size and composition of the background models set, and for example in [14] it was shown that for better performances the background set must be specific for the target.

The second approach is to train a GMM using a large set of speech samples gathered from several speakers or languages. The resulting model is called Universal Background Model (UBM). Since this is not dependent on any specific speaker or language, it can be interpreted as a model of the acoustic characteristics shared among different speakers/languages. The UBM can be trained with the EM algorithm (Section 3.1.2). Thus we obtain

$$p(\boldsymbol{x}|\lambda_{H_D}) = p(\boldsymbol{x}|\lambda_{UBM}) \quad < (3.30)$$

The number of available target feature vectors is usually quite limited. Therefore ML solutions are not very reliable and tend to overfit the training utterances. It was shown in [15] that a much better model λ_{H_S} can be estimated by adapting the UBM to the specific target speaker or language. In particular, better models can be obtained by replacing the ML estimation by a Maximum-a-Posteriori (MAP) adaptation.

The MAP estimation is similar to the Maximum Likelihood one, but it takes into account the prior distribution $g(\lambda)$ of the parameters that we want to estimate

$$\lambda_{MAP} = \arg\max_{\lambda} p(\boldsymbol{x}|\lambda)g(\lambda) \quad < (3.31)$$

Typically, the adaptation involves only the UBM mean. A successful variant of MAP adaptation is known as relevance MAP. Relevance MAP requires estimating the posterior responsibilities γ_{kt} for a set of enrollment utterances $\boldsymbol{x} = \{x_1, ..., x_T\}$

$$\gamma_{kt} = \frac{\pi_k \mathcal{N}(x_t|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x_t|\mu_j, \Sigma_j)} \quad < (3.32)$$

where $\lambda_{UBM} = \{\pi_k, \mu_k, \Sigma_k\}_{k=1}^{K}$ are the UBM parameters and K is the number of Gaussian components.

The adaptation is then obtained from the weighted sum of the γ_{kt} and from the UBM means. The weighting factor is data dependent: if the number of training data is high, the adapted parameters will be influenced more by the new statistics and viceversa. The adapted means can be computed with the following formula [14]:

$$\hat{\mu}_i = \alpha_i F_{\boldsymbol{x},i} + (1 - \alpha_i)\mu_i \quad < (3.33)$$

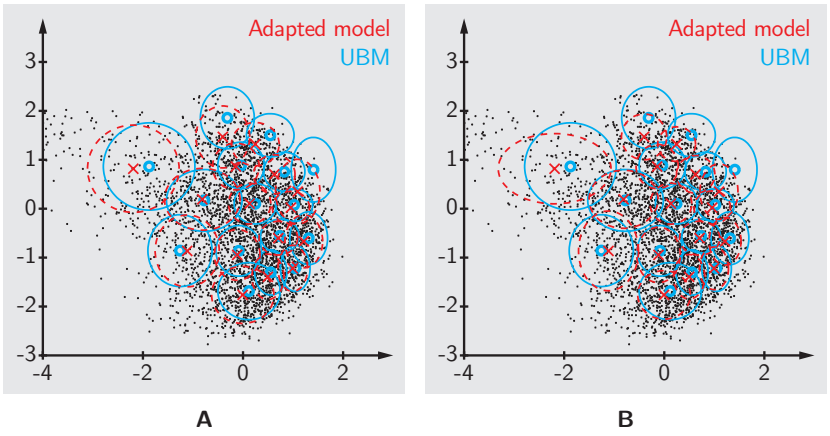
where

$$F_{\boldsymbol{x},i} = \frac{1}{N_i} \sum_{t=1}^{T} \gamma_{it} x_t \quad < (3.34)$$

$$N_i = \sum_{t=1}^{T} \gamma_{it} \quad < (3.35)$$

$$\alpha_i = \frac{N_i}{N_i + r} \quad < (3.36)$$

The relevance parameter r, and accordingly the data-dependent coefficient α_i, is used to control the effect of the enrollment utterances on the adapted model compared to the UBM [2].



< Figure 3.4: A) Example of GMM adaptation. Only the means are adapted B) All parameters are adapted

3.3 | Joint Factor Analysis

The GMM-UBM method allows evaluating both the non-target model, which we assume to be the UBM, and the target model, which is obtained by relevance MAP adaptation of the UBM, considering the target's speech utterances. However, we have to consider the problem of variability. Indeed the same speaker may pronounce the same utterance in different ways, or the speech signal may be recorded after passing through different channels. These conditions determine variations in the utterance samples of a same speaker. Thus to accurately model the speaker we have to take into account the intersession variability. This can be defined as the possible variability between several utterances coming from the same target (i.e. hypothesized speaker or language) but in different recordings.

Factor analysis models provide a technique to perform MAP adaptation of the speaker taking into account the intersession variability. In the following sessions we will introduce three MAP methods: classical MAP, eigenvoice MAP and eigenchannel MAP. The first two techniques are used for speaker variability and the third one for channel effects. Then we will show Joint Factor Analysis (JFA), which is a method to consider both speaker and channel variabilities.

3.3.1 Alignment statistics

Before showing the factor analysis methods, we need to introduce some definitions that will be used in the following sections.

Given a set of observations $X(s) = \{x_1, ..., x_n\}$ for the speaker s, we need to evaluate the alignment of $X(s)$ over the components of the GMM. In other words we have to associate each observation to a single component of the GMM. We denote μ_i as the mean of the i-th mixture component and $X_i(s)$ as the set of observations associated to this component. To compute the alignments, the full GMM should be used in the likelihood evaluations, however it is more convenient to compute an approximation of the likelihood using the Baum-Welch statistics [16].

The zero-order, first-order and centered first-order statistics are defined as

$$N_i(s) = \sum_{t=1}^{n} \gamma_{it} \qquad < (3.37)$$

$$F_i(s) = \sum_{t=1}^{n} \gamma_{it} x_t \qquad < (3.38)$$

$$F_{X,i}(s) = F_i(s) - N_i(s)\mu_i \qquad < (3.39)$$

and the second-order, and centered second-order statistics as

$$S_i(s) = \sum_{t=1}^{n} \gamma_{it} x_t x_t^{\mathrm{T}} \tag{3.40}$$

$$S_{X,i}(s) = S_i(s) - 2F_i(s)\mu_i + N_i(s)\mu_i\mu_i^{\mathrm{T}} \tag{3.41}$$

where γ_{it} are the responsibilities given by (3.32).
Using these statistics the log-likelihood for a GMM supervector is given by

$$\begin{aligned}\log P(X) &= \sum_{c=1}^{C}\left[N_c(s)\log\frac{1}{(2\pi)^{\frac{F}{2}}|\Sigma|^{\frac{1}{2}}} - \frac{1}{2}\sum_{t|x_t\in X_i(s)}\gamma_{ct}(x_t-\mu_c)^{\mathrm{T}}\Sigma_c^{-1}(x_t-\mu_c)\right] \\ &= \sum_{c=1}^{C}\left[N_c(s)\log\frac{1}{(2\pi)^{\frac{F}{2}}|\Sigma|^{\frac{1}{2}}} - \frac{1}{2}\operatorname{tr}\left(\Sigma_c^{-1}S_{X,c}(s)\right)\right]\end{aligned} \tag{3.42}$$

Finally, it is useful to define the following matrices of the stacked statistics in order to vectorize computations. $\boldsymbol{N}(s)$ is a block-diagonal matrix of size $CF \times CF$ composed by matrices $N_i\boldsymbol{I}$ of size $F \times F$. $\boldsymbol{F}(s)$ and $\boldsymbol{F}_X(s)$ are matrices given by stacking respectively $F_i(s)$ and $F_X(s)$ vectors. $\boldsymbol{S}(s)$ and $\boldsymbol{S}_X(s)$ are the block-diagonal matrices whose elements are respectively $S_i(s)$ and $S_{X,i}(s)$ matrices.

3.3.2 Classical MAP

Let $\boldsymbol{m}$ be the UBM supervector, which is a $CF \times 1$ vector obtained concatenating the mixture components' means $m_1, ..., m_C$, where C is the number of components and F is the dimension of the acoustic feature vectors. For a given speaker s, we can consider the following supervector [17]

$$\boldsymbol{M}(s) = \boldsymbol{m} + \boldsymbol{D}\boldsymbol{z}(s) \tag{3.43}$$

where $\boldsymbol{D}$ is a diagonal matrix with dimensions $CF \times CF$ and $\boldsymbol{z}(s)$ is a CF vector of speaker-dependent hidden variables with normal distribution

$$\boldsymbol{z}(s) \sim \mathcal{N}(\boldsymbol{z}|\boldsymbol{0}, \boldsymbol{I}) \tag{3.44}$$

It is possible to show that the latent variable model of (3.43) is similar to the model used in relevance MAP.

Omitting for simplicity the reference to s, let $\boldsymbol{N}$ and $\boldsymbol{F}_X$ be the stacked Baum-Welch statistics defined in Section 3.3.1. We can compute the joint likelihood of the observed data X and the latent variables $\boldsymbol{z}$ as follows

$$P(X, \boldsymbol{z}) = P(X|\boldsymbol{z})P(\boldsymbol{z}) \propto \left(\boldsymbol{z}^T\boldsymbol{D}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{F}_X - \frac{1}{2}\boldsymbol{z}^T\boldsymbol{D}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{N}\boldsymbol{D}\boldsymbol{z} - \frac{1}{2}\boldsymbol{z}^T\boldsymbol{z}\right) \tag{3.45}$$

where $\boldsymbol{\Sigma}$ is a block-diagonal matrix whose elements are the UBM covariance matrices. Thus the posterior of $\boldsymbol{z}$ can be evaluated as

$$P(\boldsymbol{z}|X) \propto P(X,\boldsymbol{z}) \propto \left(\boldsymbol{z}^T\boldsymbol{D}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{F}_X - \frac{1}{2}\boldsymbol{z}^T\boldsymbol{D}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{N}\boldsymbol{D}\boldsymbol{z} - \frac{1}{2}\boldsymbol{z}^T\boldsymbol{z}\right) \quad < (3.46)$$

By inspection of (3.45) we can assume that the posterior of $\boldsymbol{z}$ is Gaussian

$$\boldsymbol{z}|X \sim \mathcal{N}(\boldsymbol{z}|\boldsymbol{\mu}_{\boldsymbol{z}}, \boldsymbol{\Lambda}_{\boldsymbol{z}}^{-1}) \quad < (3.47)$$

where the mean and the precision matrix are

$$\boldsymbol{\Lambda}_{\boldsymbol{z}} = \left(\boldsymbol{D}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{N}\boldsymbol{D} + \boldsymbol{I}\right) = \left(\boldsymbol{D}^2\boldsymbol{\Sigma}^{-1}\boldsymbol{N} + \boldsymbol{I}\right) \quad < (3.48)$$

$$\boldsymbol{\mu}_{\boldsymbol{z}} = \boldsymbol{\Lambda}_{\boldsymbol{z}}^{-1}\boldsymbol{D}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{F}_X \quad < (3.49)$$

Therefore the adapted GMM is given by

$$\boldsymbol{M} = \boldsymbol{m} + \boldsymbol{D}\mathbb{E}[\boldsymbol{z}] = \boldsymbol{m} + \left(\boldsymbol{D}^2\boldsymbol{\Sigma}^{-1}\boldsymbol{N} + \boldsymbol{I}\right)\boldsymbol{D}^2\boldsymbol{\Sigma}^{-1}\boldsymbol{F}_X \quad < (3.50)$$

Defining $\boldsymbol{r} = \boldsymbol{D}^2\boldsymbol{\Sigma}^{-1}$ we obtain

$$\boldsymbol{M} = \boldsymbol{m} + (\boldsymbol{N} + \boldsymbol{r})^{-1}\boldsymbol{F}_{\boldsymbol{x}} \quad < (3.51)$$

which as the same form as (3.33).

3.3.3 Eigenvoice MAP

The eigenvoice MAP model represents a supervector as

$$\boldsymbol{M}(s) = \boldsymbol{m} + \boldsymbol{V}\boldsymbol{y}(s) \quad < (3.52)$$

where $\boldsymbol{V}$ is a matrix with dimension $CF \times R_S$, with $R_S \ll CF$, and $\boldsymbol{y}(s)$ is a normal distributed hidden vector of size $R_S \times 1$. Compared to the classical MAP, the adaptation is performed in a subspace of considerably smaller dimension (R_S), without losing too much accuracy. Indeed it is assumed that the variability between the different speakers is mainly confined in this smaller subspace. Furthermore, the eigenvoice method is more efficient when the enrollment data is sparse because it requires the estimation of a smaller latent variable.

In particular let $\boldsymbol{m}$ and $\boldsymbol{B}$ be the mean and covariance of the supervectors for the UBM. Since most of the eigenvalues of $\boldsymbol{B}$ equal zero, the speaker-dependent supervectors can be contained in a low-dimensional space, which is known as eigenspace. Furthermore the eigenvoices are defined as the eigenvectors of $\boldsymbol{B}$ corresponding to non-zero eigenvalues [18].

It is possible to compute the posterior of $\boldsymbol{y}(s)$ in a similar way to the one proposed for classical MAP

$$\boldsymbol{y}|\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{y}|\boldsymbol{\mu_y}, \boldsymbol{\Lambda}_{\boldsymbol{y}}^{-1}) \quad < (3.53)$$

where mean and precision matrix are given by

$$\boldsymbol{\Lambda_y} = \left(\boldsymbol{V}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{N} \boldsymbol{V} + \boldsymbol{I}\right) \quad < (3.54)$$

$$\boldsymbol{\mu_y} = \boldsymbol{\Lambda}_{\boldsymbol{y}}^{-1} \boldsymbol{V}^T \boldsymbol{\Sigma}^{-1} \boldsymbol{F}_X \quad < (3.55)$$

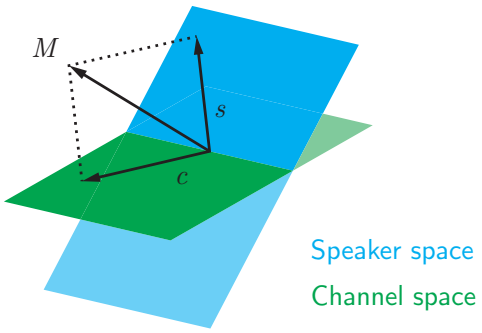
3.3.4 Eigenchannel MAP

The models presented so far do not explicitly model the channel effects. Even if there are some techniques which perform a compensation of the channel effects directly in the feature space, successful methods use factor analysis, and perform model compensation.

Let h be a given recording and s a given speaker. The GMM supervector can be written as the sum of two components [19][20]: $\boldsymbol{s}(s)$ is a speaker-dependent component and $\boldsymbol{c}_h(s)$ is a channel-dependent component

$$\boldsymbol{M} = \boldsymbol{s}(s) + \boldsymbol{c}_h(s) \quad < (3.56)$$

It is assumed that the speaker component and the channel component lie in different and orthogonal subspaces of the supervector space.



< Figure 3.5: Speaker and channel subspaces

A popular technique for channel compensation is eigenchannel adaptation, which defines the channel component as follows

$$\boldsymbol{c}_h(s) = \boldsymbol{U} \boldsymbol{x}_h(s) \quad < (3.57)$$

where $\boldsymbol{U}$ is a $CF \times R_C$ low rank matrix, with $R_C \ll CF$, and $\boldsymbol{x}_h(s)$ is an hidden vector with normal distribution, which represents the channel

effects. This method assumes that the channel component lies in a subspace of small dimension.

Therefore, a supervector is given by

$$\boldsymbol{M}_h(s) = \boldsymbol{M}(s) + \boldsymbol{U}\boldsymbol{x}_h(s) \tag{3.58}$$

Similarly to the previous models, the posterior of $\boldsymbol{x}_h$ is given by

$$\boldsymbol{x}_h|\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{x}_h|\boldsymbol{\mu}_{\boldsymbol{x}_h}, \boldsymbol{\Lambda}_{\boldsymbol{x}_h}^{-1}) \tag{3.59}$$

with

$$\boldsymbol{\Lambda}_x = \left(\boldsymbol{U}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{N}\boldsymbol{U} + \boldsymbol{I}\right) \tag{3.60}$$

$$\boldsymbol{\mu}_x = \boldsymbol{\Lambda}_x^{-1}\boldsymbol{U}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{F}_X \tag{3.61}$$

3.3.5 JFA model

Finally, Joint Factor Analysis (JFA) is a combination of these three modelling techniques that takes into account both speaker and session variability. In particular, putting together (3.43), (3.52) and (3.58), the model can be formalized as [16]

$$\boldsymbol{M}_h(s) = \boldsymbol{m} + \boldsymbol{V}\boldsymbol{y}(s) + \boldsymbol{U}\boldsymbol{x}_h(s) + \boldsymbol{D}\boldsymbol{z}_h(s) \tag{3.62}$$

where $\boldsymbol{V}$ is the eigenvoice matrix that defines the speaker subspace and $\boldsymbol{U}$ is the eigenchannel matrix that defines the session subspace. Moreover, $\boldsymbol{m}$ is the UBM supervector (speaker- and session-independent), $\boldsymbol{y}(s)$ represents speaker-dependent factors and $\boldsymbol{x}_h(s)$ channel factors. The term $\boldsymbol{D}\boldsymbol{z}_h(s)$ is used to represent residual variability and $\boldsymbol{z}_h(s)$ are called common factors.

During training it is possible to estimate the subspaces ($\boldsymbol{V}$, $\boldsymbol{D}$, $\boldsymbol{U}$) with labelled data, and to compute the speaker and session factors for a given utterance of a target, by means of the posterior distributions ((3.47), (3.53), (3.59)) and the EM algorithm.

3.4 | I-vectors

The JFA modelling is an effective technique for the use of low-dimensional vectors, however experiments have shown that the channel factors do not contain only channel effects, but also speaker information [21]. Therefore,

a similar method has been proposed which contemplate a single subspace, called total variability space [22]. This contains both speaker and channel variability, which are not distinguished in this model. It is possible to represent a given utterance in this subspace with an hidden vector, called total factors vector. The total variability space is low-dimensional, and maintains the proved advantages of JFA.

An utterance supervector can be modelled as

$$\boldsymbol{M}(s) = \boldsymbol{m} + \boldsymbol{T}\boldsymbol{w}(s) \tag{3.63}$$

where $\boldsymbol{M}(s)$ is the speaker and channel dependent supervector, $\boldsymbol{m}$ is the UBM supervector, $\boldsymbol{T}$ is a low rank matrix with size $CF \times R_T$, and $\boldsymbol{w}(s)$ is an hidden variable with normal distribution $\mathcal{N}(0, I)$. Variable $\boldsymbol{w}$ represents a GMM in the total variability space. Its posterior distribution is Gaussian and its mean corresponds to the i-vector.

$\boldsymbol{T}$ matrix training is similar to eigenvoice matrix training, with the distinction that in the latter the utterances of a speaker are assumed to be actually spoken by the same speaker, while in the former they are considered as belonging to different speakers. In other words in the eigenvoice MAP we assume that the utterances of a speaker share the same hidden vectors, whereas in the i-vector technique each utterance is considered to have its own hidden vector.

The posterior of $\boldsymbol{w}$ is given by

$$\boldsymbol{w}|\boldsymbol{x} \sim \mathcal{N}(\boldsymbol{w}|\boldsymbol{\mu}_{\boldsymbol{w}}, \boldsymbol{\Lambda}_{\boldsymbol{w}}^{-1}) \tag{3.64}$$

where the mean is

$$\boldsymbol{\mu}_{\boldsymbol{w}} = \boldsymbol{\Lambda}_{\boldsymbol{w}}^{-1}\boldsymbol{T}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{F}_{\boldsymbol{x}} \tag{3.65}$$

and the precision matrix is

$$\boldsymbol{\Lambda}_{\boldsymbol{w}} = \boldsymbol{T}^T\boldsymbol{\Sigma}^{-1}\boldsymbol{N}\boldsymbol{T} + \boldsymbol{I} \tag{3.66}$$

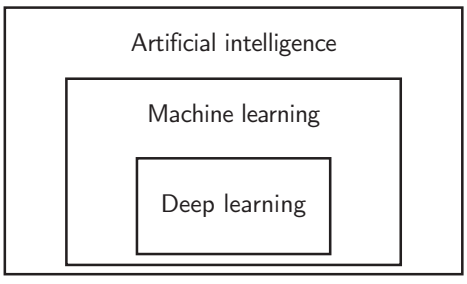
Chapter 4
Artificial neural networks

Artificial Neural Networks (ANN) are essential deep learning models, and very useful computational tools. The aim of an ANN is to approximate a function f^*, which, for example, classifies an input x, mapping it to the category y: $y = f^*(x)$ [12]. In particular the ANN learns the parameters θ which give the best approximation of the function: $y = f(x, \theta)$ where f is the function applied by the network on the input x to obtain the output y.

The term networks refers to the fact that ANN are composed by various nodes grouped in layers, and can be represented by a directed graph of these nodes. The term neural refers to the fact that the network of nodes was loosely inspired by the human neurons and how they are connected.

The number of layers determines the depth of the network, and because of this terminology ANNs can be deep learning models, a specific subfield of machine learning.

4.1 | Artificial intelligence, machine learning and deep learning

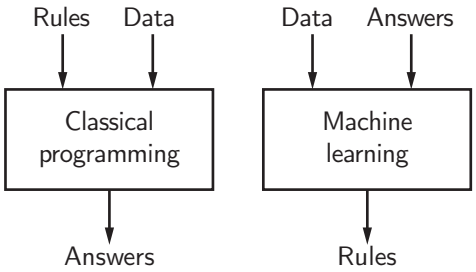


< Figure 4.1: Artificial intelligence, machine learning and deep learning

Artificial intelligence is a field of computer science born in the 1950s to answer the question "could a computer think like a human?". The aim of

artificial intelligence is to automate intellectual tasks usually performed by humans and to give cognitive functions to machines, such as learning and problem solving [23]. Artificial intelligence is a broad field which includes many approaches, including the ones in which rules are coded by hand, and the most recent ones in which the rules are automatically learned.

Machine learning covers the latter approach. The rules to perform a given task are not coded by programmers, but are learnt automatically from a sufficiently large set of data. Machine learning introduces a new programming paradigm. The classical way of programming is to write the rules (the code), and to give input data that will be processed, according to the rules, to obtain the answers. The machine learning paradigm instead requires as input pairs of data and expected anwers, and produces as results the rules, which can then be applied to process new data.



< Figure 4.2: Machine learning as a new programming paradigm

The procedure to find the rules is called training: many relevant data must be presented to the system, a statistical structure can be eventually found, which corresponds to the rules to automate the desired task.

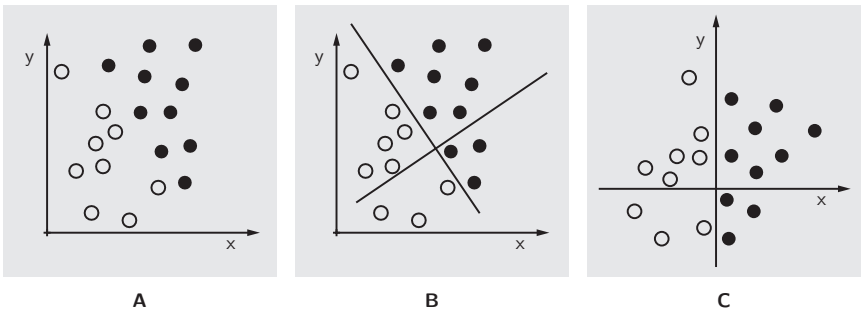
Three elements are needed for machine learning:

- Input data. For example, in language recognition the input data are recordings of people speaking different languages.
- Expected output. For instance, the label of the language spoken in each spoken segment.
- A way to measure the accuracy of the algorithm. The accuracy is measured as the "distance" between the output of the system and the given expected output.

The measure of accuracy is used as feedback to adjust the parameters of the system (θ) to enhance the system performance. This procedure is essentially what is called *learning*.

In order to get closer to the expected output, the input data are transformed in more meaningful representations. For example, having white and black points on a plane, the task could be to give the likely colour of a point given its coordinates. In this case, the input data are the coordinates of the points, the expected output their colours and the measure of accuracy could be the percentage of correctly classified points. A meaningful representation

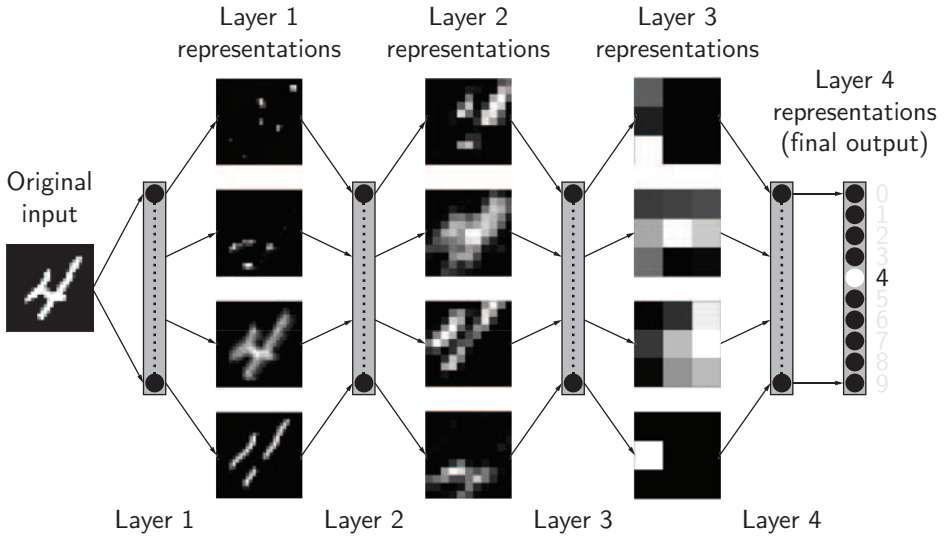
of the data is one that allows to easily separate white and black points. Thus a change of coordinates (like the one depicted in Figure 4.3) can be performed. Therefore the black points can be identified by the simple rule "if $x > 0$".



< Figure 4.3:
A) Raw data
B) Change of coordinates
C) Better representation

A machine learning algorithm can be used to automatically find such meaningful transformation, which can be coordinate changes, linear projections, translations, non-linear operations, and so on. The feedback signal is used as a guide to find the most useful representation of the input data for the given task.

One of the possible machine learning techniques is deep learning, where the algorithm can learn different levels of representations: the deeper is the layer of the network, the more meaningful is the representation which can be learnt. The layers can be thought as filters that incrementally extract more relevant information. Thus, the input data are transformed by each layer, and gradually purified to better perform the task of the system (Figure 4.4).



< Figure 4.4:
Representations learnt by the model.
The deeper is the layer, the more filtered are the representations

4.2 | Working principles of neural networks

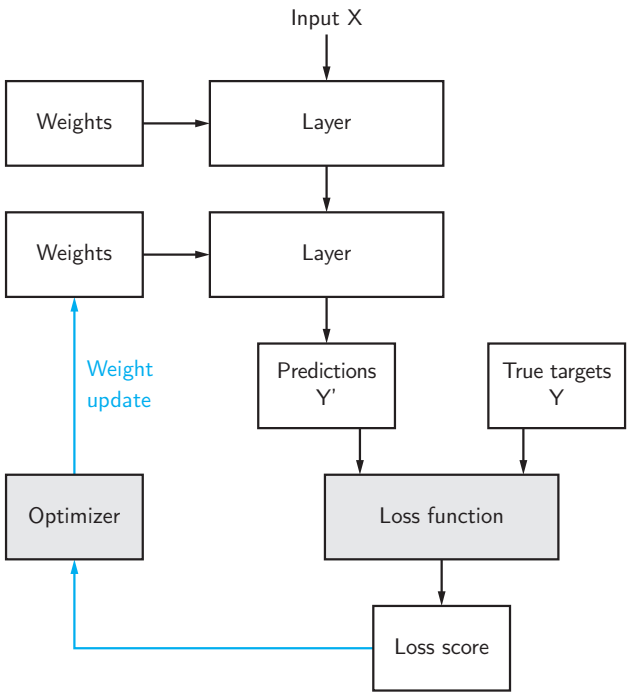
Neural networks perform a mapping of inputs to targets. A good mapping is found out during training if the network is exposed to many examples of inputs and targets. In particular, the network performs the mapping by means of sequential data transformations realized in the layers.

A layer transforms the data depending on its weights, also called parameters. During training, the weights are learnt having as objective the most accurate mapping between inputs and targets.

To adjust the weights of the layers, it is necessary to control the output of the network, and to measure how much the predictions of the network are different from the expected target. To compute this difference, the loss function (also called objective function) is evaluated (Figure 4.5). If the distance score between predictions and target, which is called loss score, is high, the accuracy is low and the weights will be adjusted.

The loss score is used as feedback signal to adjust the weights: these are slightly moved in a direction that will cause a decrement of the loss score for the current example. The weights update is performed by the optimizer (Figure 4.5).

< Figure 4.5: Schema of the functioning of a neural network



At the beginning, the weights of the network are initialized randomly. Thus the predictions will usually be distant from the target. Then, every

example is processed by the network, and the weights adjusted to make the loss score decrease. This cycle is executed a sufficient number of times to have the loss function minimized, and to obtain a trained network that will produce predictions close enough to the expected outputs.

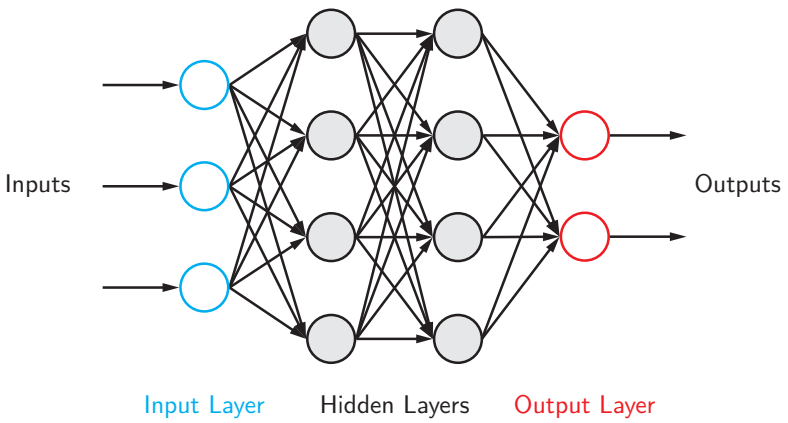
As it has been said, the aim of a network is to approximate a function f^* with a function f that allows mapping the inputs to the targets. With more than one layer, the function f can be written as a chain of the specific functions of each layer $f^{(i)}$. For example, the function of a network with three layers is

$$f(x) = f^{(3)}\left(f^{(2)}\left(f^{(1)}(x)\right)\right) \qquad < (4.1)$$

During training, the weights θ are adjusted to make $f(x)$ close enough to $f^*(x)$. The training data are the examples x associated with the labels $y \approx f^*(x)$. Thus, given the input x, the output layer of the network should return a value close to y. However, the value returned by the previous layers is not specified by the training data, thus the learning algorithm has to establish how to parameterize these layers so that the overall network function is a good approximation of f^*. Since the training data does not provide the expected output values of these layers, they are called hidden layers (Figure 4.6).

One of the reasons that make deep learning very powerful is that all the layers of representation are learnt jointly: if one weight is adjusted, automatically all the other weights that depend on it will be updated. The overall training depends on a single feedback signal, thus every change in the model aims to get closer to the end goal.

Each layer is composed by a given number of units (Figure 4.6), or nodes, that implement a vector-to-scalar function. Indeed, each unit receives input from all the units of the previous layer, and computes its output scalar, also called activation value. For each node a function is defined that establishes how the output of the node itself has to be computed, and which is its activation function.



< Figure 4.6: Artificial neural network architecture

4.3 | Neural network architecture

An ANN can be represented by a directed graph (Figure 4.6)

$$G(I, N, O, E) \tag{4.2}$$

where I, N and O are respectively the sets of input, internal and output nodes and E is the set of edges [13]. The nodes are grouped in layers, which can have different widths (which correspond with the number of nodes of the layer). The input nodes receive the data to be processed, and the output nodes return the result of the computation. Every node of the sets N and O has a set of parent nodes as defined by the connections given by the edges. Furthermore, each node of a hidden layer receives input from all the nodes of the previous layer and gives the output to all the nodes of the successive layer.

A node of the network is denoted as n_j, and it has associated the input values x_i, the activation value a_j, and the output value z_j. An edge of the network is denoted as $\xi_{j,i}$, and it connects the nodes n_j and n_i. Each edge has associated a weight $w_{j,i}$.

4.3.1 Single node

The node is the building block of the neural network. It receives as input a vector of values, performs a computation, and returns an output value. The computation is a weighted sum of the input values with the weights of the edges and a bias term. Thus, given the input vector $X = \{x_1, ..., x_i, ..., x_M\}$, the set of corresponding weights $W = \{w_{j,1}, ..., w_{j,i}, ..., w_{j,M}\}$, and the bias b_j, the activation value of the j-th node is

$$a_j = \sum_{i=1}^{M} w_{j,i} x_i + b_j \tag{4.3}$$

Instead of using the activation value, which is a linear function, a non-linear function $h()$ is applied to a_j. Then the output value for a node is

$$z_j = h(a_j) \tag{4.4}$$

where $h()$ is the activation function of the node.

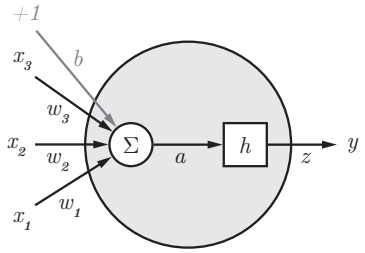


Figure 4.7: Single node schema

4.3.2 Network function

If we consider a network, the nodes' outputs of a layer are again combined by the nodes of the successive layer. In the usual architecture, the layers are fully-connected: this means that every node of a layer takes as inputs all the results of the nodes of the previous layer. In other words, each node of a k-th layer is connected with an edge to all the nodes of the k-*1*-th layer, and its activation is a weighted sum of all their outputs. Thus, the activation of the j-th node of the k-th layer can be written as

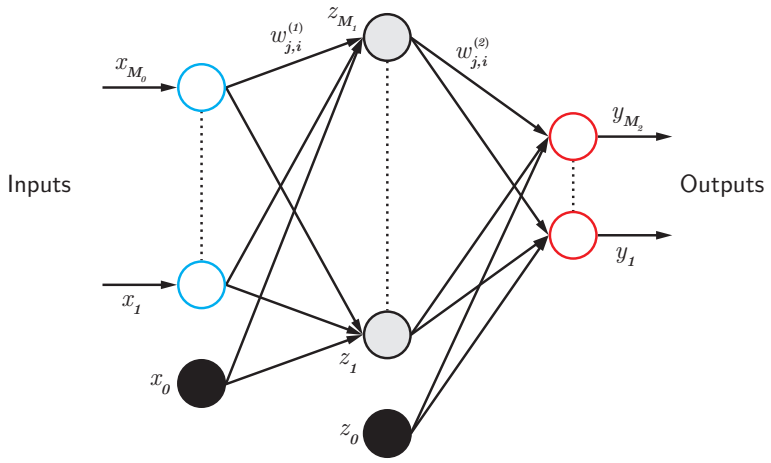
$$a_j^{(k)} = \sum_{i=1}^{M_{k-1}} w_{j,i}^{(k)} z_i + b_j^{(k)} \qquad < (4.5)$$

where k identifies the layer, M_{k-1} is the number of nodes of the k-*1*-th layer and $j = 1, ..., M_k$ identifies the node of the layer (with M_k being the total number of nodes of the current layer). For each node, the activation function $h()$ is then applied to the activation value.

If we consider a neural network with one hidden layer (Figure 4.8), the overall network function is

$$y_j(X, W) = h\left(\sum_{i=1}^{M_1} w_{j,i}^{(2)} h\left(\sum_{i=1}^{M_0} w_{j,i}^{(1)} x_i + b_j^{(1)}\right) + b_j^{(2)}\right) \qquad < (4.6)$$

where the vector of biases has been grouped with the weights in the matrix W.



< Figure 4.8: Network diagram

Therefore, a neural network model is a non-linear function which takes as input the data X and the parameters W, and returns as output the set Y.

4.3.3 Activation functions

According to the task that the network has to solve, a convenient type of activation functions has to be properly used. A large number of functions

exists, and we show below the main ones for the purposes of this work.

Sigmoid

For binary classification, the activation function of the output layer is usually a sigmoid function, given by

$$h(a) = \frac{1}{1 + e^{-a}} \tag{4.7}$$

Since the value produced by this function is between 0 and 1, the output of the network can be interpreted as a probability. Thus the result is the probability that the input belongs to one class or the other. Moreover, when the argument is very positive or very negative, the function tends to saturate to 0 or 1 [12]. This means that it becomes insensitive to small changes for inputs above 2 or below -2, for example. This property makes the function work better to clearly distinguish the predictions.

The fact that the function doesn't change significantly for small changes of the argument can also be a disadvantage, because it raises the problem of vanishing gradients. This means that the gradients become smaller and smaller, so that the network can't learn very well.

The sigmoid function can be applied not only on the activations of the output nodes, but also on the activations of the hidden nodes.

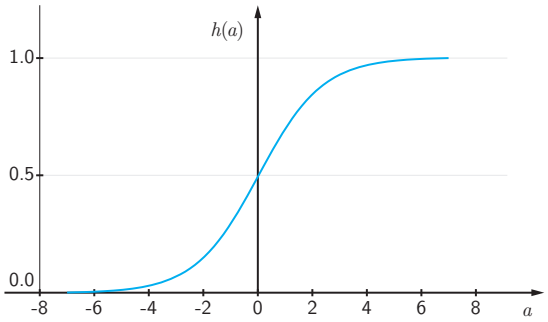


Figure 4.9: Sigmoid function

ReLU

An activation function that can also be used for the hidden nodes is the Rectified Linear Unit (ReLU) [24][25], which is written as

$$h(a) = \max(0, a) \tag{4.8}$$

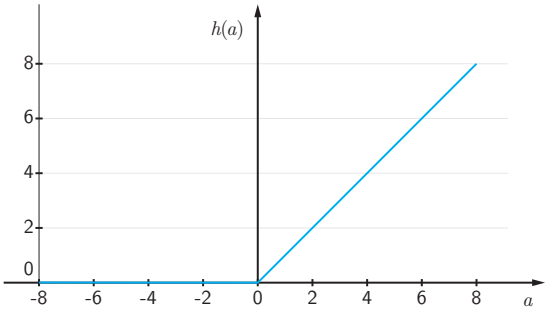
Unlike the previous one, this function doesn't tend to flatten for very high or very low input values. However, since the range of ReLU is $[0, +\infty)$, this also means that this function can make the activations become higher and higher.

Even if ReLU is a non-linear function, it is composed by two linear pieces (piecewise linear), thus the network model will be easier to optimize with gradient-based methods, and it can generalize well [12].

Since the ReLU brings to 0 all negative values, the network will have sparse activations. For example if we consider a network initialized with random weights, ReLU will yield almost half of the nodes to be 0. This fact makes the network easier to compute. On the other side, many nodes won't react to changes in the network, because every negative value is brought to 0.

Another advantage of ReLU is that, in comparison with sigmoid and tanh functions, it is less computationally expensive.

< Figure 4.10: ReLU function



Tanh

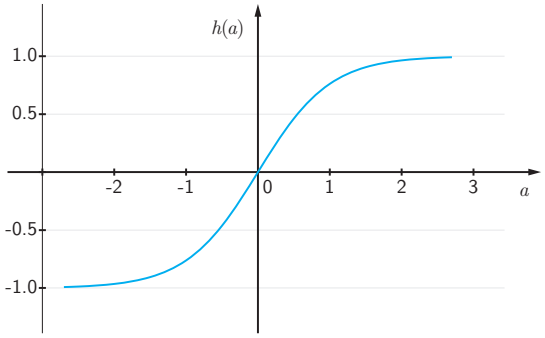
Another activation function widely used is tanh:

$$h(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}} \tag{4.9}$$

< (4.9)

It is essentially a sigmoid scaled between -1 and 1. Thus all previously mentioned properties are still valid.

< Figure 4.11: Tanh function



Softmax

The softmax function is only used in the output layer for multiclass classification tasks. This function can take as input a vector of real numbers, and returns a vector that encodes a probability distribution, i.e. all the values of the vector will be in the range 0 and 1, and the sum of them will be 1 [26]. For a vector of dimensionality D, the softmax function is given by

$$h(a_i) = softmax(a_i) = \frac{e^{a_i}}{\sum_{j=1}^{D} e^{a_j}} \tag{4.10}$$

< (4.10)

4.4 | Training

Training a neural network essentially means to set the weights $w_{j,i}$ and biases b_j for each layer. This is done choosing the values of these parameters that minimize the loss score. We will refer to $w_{j,i}$ for both weights and biases, since a bias can be considered as a weight of an additional node with input fixed to 1.

Various optimization methods can be used to minimize the loss score, like the stochastic gradient descent (page 46). The basic idea is that we initialize the weights randomly, and we iteratively update the weights in a direction that improves the accuracy of the model.

Each iteration is composed by two main steps:

- Forward propagation: given inputs from dataset and the current weights of the network, the final output is computed;
- Backward propagation: having the output, it is possible to compute the loss score, and to update the weights proceeding from the output layer to the input layer.

As we previously introduced, the forward propagation can be performed by means of the following equations:

$$a_j^{(k)} = \sum_{i=1}^{M_{k-1}} w_{j,i}^{(k)} z_i^{(k-1)} + w_{j,0}^{(k)} \quad \forall j \tag{4.11}$$

$$z_j^{(k)} = h\left(a_j^{(k)}\right) \quad \forall j \tag{4.12}$$

where $z_j^{(k)}$ is the output value of the j-th node of the k-th layer. Computing the values from the first to the last layer, it is possible to obtain the values y_j of the output layer.

Now we have to evaluate the loss score and efficiently propagate backward the error to update the weights.

4.4.1 Loss function

Since our aim is to have output values as close as possible to the target values, we need a metric to evaluate if the system is improving or not its accuracy on the training data each time we change the weights. This computation is performed by the loss function, also called cost function.

Squared errors

The sum of the squared residuals over all the training cases is a often used function to evaluate the distance between the target value t and the

model's output value y. It can be written as

$$E = \frac{1}{2}\sum_n (t^n - y^n)^2 \qquad \text{< (4.13)}$$

where n is the index of the training case considered. The factor is needed to simplify the derivative which will be computed next.

Cross entropy

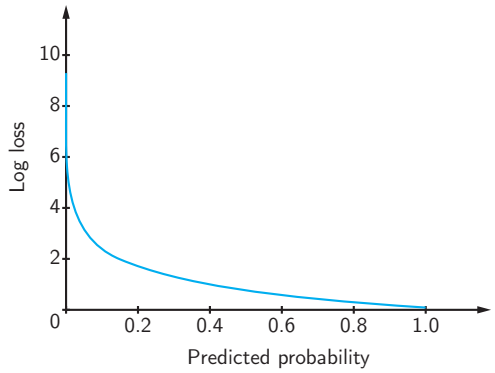
Another common function is the cross entropy loss, which is also called the negative log likelihood. In particular for softmax neurons, the most appropriate cost function is the negative log probability of the correct answer. The equation is

$$E = -\sum_n t^n \log y^n \qquad \text{< (4.14)}$$

where n is the number of the classes between which the model has to do the classification. Furthermore t^n and y^n are probability distributions.

A model which does a correct classification assigns a probability of 1 to the right class, and a probability of 0 to the others. The negative log function gives a value of 0 for a probability of a correct classification example (no loss), whereas it gives a value of infinite for a probability of a wrong one (infinite loss) (Figure 4.12).

The cross entropy function has a good property: it has a big gradient if the target value is 1, and the output of the model is almost 0.

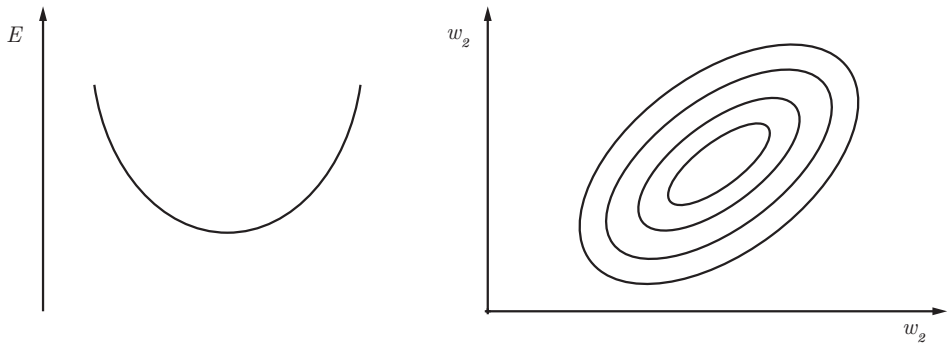


< Figure 4.12: Log loss when the true label is equal to 1

Geometric interpretation

If we consider a linear neuron and a squared error function, it is possible to visualize the error surface in a 3D space, with the weights on the horizontal axis and the error on the vertical axis. The shape of the error surface is a quadratic bowl. The aim of the training is to find the point at the bottom of the bowl. This point gives the weights which minimize the error.

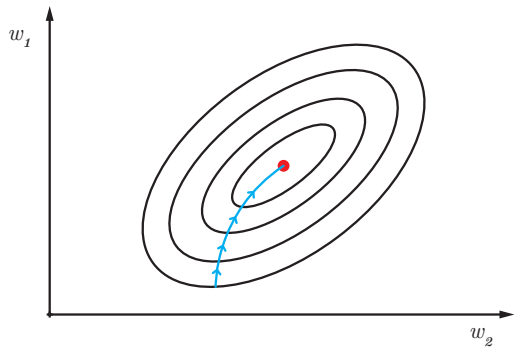
For multi-layer non-linear neural networks the error surface becomes more complex. However, if the weights are not too big, the error surface is still smooth, but it may have many local minima.



< Figure 4.13: Representation of the error surface

4.4.2 Gradient descent

The minimum of the loss function can't be found analytically. Instead, iterative methods are used, like the gradient descent algorithm or other extensions or variants of this. As the name says, the gradient descent is a method to find the minimum of a function taking iterative steps in the direction of the negative gradient, which is computed at the current point. This means that we are moving on the error surface following the direction along the steepest slope.



< Figure 4.14: Gradient descent on the error surface

Since we are in the weights space, at each iteration we update the weights according to the gradient descent algorithm. However, the amount of the step doesn't depend only on the magnitude of the gradient, but this is multiplied by the learning rate η. For an higher learning rate, the step will be bigger, and viceversa. Thus, the updated weights are given by

$$w_{j,i}^{(k)}(t+1) = w_{j,i}^{(k)}(t) - \eta \frac{\partial E}{\partial w_{j,i}^{(k)}(t)} \tag{4.15}$$

< (4.15)

It's important to set a convenient value for the learning rate. Indeed, if η is too high, the steps will be too big, and the minimum can be lost, whereas if it is too low, the steps will be too small, and more time will be needed to find the minimum. Sometimes it can be useful to start with an high learning rate, and then decrease it as the training iterations proceed.

In each dimension $w_{j,i}$ of the weights space, the gradient gives the component of the slope in that dimension. This component is expressed as the partial derivative of the loss function with respect to the weights, which is computed by means of the backpropagation algorithm.

Backpropagation algorithm

The partial derivative of the loss can be computed [27] applying two times the chain rule

$$\frac{\partial E}{\partial w_{j,i}} = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial a_j}\frac{\partial a_j}{\partial w_{j,i}} \tag{4.16}$$

where we temporarily omit the index of the layer k and the index of the iteration t.

Let the backpropagation error be

$$\delta_j = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial a_j} \tag{4.17}$$

The partial derivative of the activation value can be computed as

$$\frac{\partial a_j}{\partial w_{j,i}} = \frac{\partial}{\partial w_{j,i}}\left(\sum_i w_{j,i} z_i\right) = z_i \tag{4.18}$$

In fact the activation value a_j is simply the sum of all the outputs from the nodes of the previous layer i multiplied by the weights.

Therefore we have

$$\frac{\partial E}{\partial w_{j,i}} = \delta_j z_i \tag{4.19}$$

Now we can proceed to compute δ_j. The partial derivative of z_j with respect to a_j depends on the activation function of the node j. The partial derivative of the loss function with respect to z_j depends on the chosen loss function. It can be rewritten as

$$\frac{\partial E}{\partial z_j} = \sum_i \left(\frac{\partial E}{\partial a_i}\frac{\partial a_i}{\partial z_j}\right) \tag{4.20}$$

Since

$$\frac{\partial E}{\partial a_i} = \frac{\partial E}{\partial z_i}\frac{\partial z_i}{\partial a_i} = \delta_i \tag{4.21}$$

we have

$$\frac{\partial E}{\partial z_j} = \sum_i \left(\delta_i \frac{\partial a_i}{\partial z_j}\right) = \sum_i (\delta_i w_{j,i}) \tag{4.22}$$

Putting (4.17) and (4.22) together, we obtain

$$\delta_j = \sum_i (\delta_i w_{j,i}) \frac{\partial z_j}{\partial a_j} \tag{4.23}$$

We can notice that it is sufficient to have the errors δ_i of the nodes of the layer $L+1$ (where the layer 0 is the input layer, and the numbers grow towards the output) in order to compute δ_j for a node of layer L. Indeed, the aim of the backpropagation algorithm is the computation of the updated weights by propagating backward the error of the output layer through the entire network.

Thus, the function (4.15) to update the weights becomes

$$w_{j,i}(t+1) = w_{j,i}(t) + \Delta w_{j,i}(t+1) \qquad < (4.24)$$

and

$$\Delta w_{j,i}(t+1) = -\eta(t)\delta_j(t)z_i(t) \qquad < (4.25)$$

Backpropagation for a linear neuron

We consider the simplest example of learning: a linear neuron with squared error measure. The output z of the node is the weighted sum of the inputs x_i.

$$y = \sum_i w_i x_i \qquad < (4.26)$$

To learn the weights of the model we have to minimize the error summed over all training cases. To measure the loss we use the squared difference between the target output and the estimated output and we sum these residuals over all the n training cases:

$$E = \frac{1}{2}\sum_n (t^n - y^n)^2 \qquad < (4.27)$$

We can differentiate to get the error derivatives with respect to the weights. To do this, we use the chain rule:

$$\frac{\partial E}{\partial w_i} = \frac{1}{2}\sum_n \frac{\partial y^n}{\partial w_i}\frac{\partial E^n}{\partial y^n} = -\sum_n x_i^n (t^n - y^n) \qquad < (4.28)$$

Thus we can update a weight w_i in proportion to the partial derivatives of the error, summed over all training cases. We obtain the batch delta rule:

$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i} = \sum_n \eta x_i^n (t^n - y^n) \qquad < (4.29)$$

Backpropagation for a sigmoid neuron

Now we consider a sigmoid neuron. The output of the node is computed as it follows:

$$z = \sum_i w_i x_i \qquad < (4.30)$$

$$y = \frac{1}{1+e^{-z}} \qquad < (4.31)$$

Similarly to the computation that we have done previously, we compute the derivative of the error with respect to the weights by means of the chain rule:

$$\frac{\partial E}{\partial w_i} = \frac{1}{2}\sum_n \frac{\partial y^n}{\partial w_i}\frac{\partial E^n}{\partial y^n} = -\sum_n \frac{\partial y^n}{\partial w_i}(t^n - y^n) \quad < (4.32)$$

Even to compute the partial derivative of the output we can use the chain rule in the following way

$$\frac{\partial y}{\partial w_i} = \frac{\partial z}{\partial w_i}\frac{\partial y}{\partial z} = x_i y(1-y) \quad < (4.33)$$

where

$$\frac{\partial z}{\partial w_i} = x_i \quad < (4.34)$$

and

$$\frac{\partial y}{\partial z} = \frac{-\left(-e^{-z}\right)}{\left(1+e^{-z}\right)^2} = \frac{1}{1+e^{-z}}\frac{e^{-z}}{1+e^{-z}} = y(1-y) \quad < (4.35)$$

because

$$\frac{e^{-z}}{1+e^{-z}} = \frac{(1+-e^{-z})-1}{(1+e^{-z})} = \frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}} = 1-y \quad < (4.36)$$

Thus we obtain

$$\frac{\partial y}{\partial w_i} = x_i y(1-y) \quad < (4.37)$$

and

$$\frac{\partial E}{\partial w_i} = \frac{1}{2}\sum_n \frac{\partial y^n}{\partial w_i}\frac{\partial E^n}{\partial y^n} = -\sum_n x_i^n y^n\left(1-y^n\right)\left(t^n - y^n\right) \quad < (4.38)$$

The quantity to update the weights for a sigmoid neuron is

$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i} = \sum_n \eta x_i^n y^n\left(1-y^n\right)\left(t^n - y^n\right) \quad < (4.39)$$

4.4.3 Optimizer

The backpropagation algorithm is an efficient way to compute the error derivative with respect to each weight for a single training case. For a whole learning procedure, we still need to specify how often and how much to update the weights. Since the neural networks may have a large number of weights, and for training we need to process a large number of data, it is essential to use a fast optimization algorithm which updates the weights efficiently.

In general we can update the weights choosing one of the following strategies:

- Full batch: all the error derivatives computed for all the training cases of the dataset are averaged together to get the updating step. Let $\{(X^1, T^1), ..., (X^P, T^P)\}$ be the training set of the input-output pairs. The loss function E^p gives the distance between the output computed by the model and the expected output. The average cost function is E_{train}, and it can be evaluated like the average of all the errors E^p computed on the training set [28]:

$$E_{train} = \frac{1}{P}\sum_{p=1}^{P} E^p \qquad < (4.40)$$

  The problem with this approach is that if we have very bad initial weights, it takes a lot of time to adjust them because we have to process the entire dataset at every iteration.
- Online: the weights are updated after each training case. Thus the derivative of the error is computed on E^p. This strategy can cause the weights values to oscillate rather than taking a good direction toward convergence.
- Mini-batch: we take a small random set of the training samples, we compute the error derivatives, and we sum them to get the update quantity for the weights.

Mini-batch learning is the strategy typically used for training big neural networks on large dataset, and is the one chosen in this work.

Furthermore, it is important to set a good learning rate. Also in this case different strategies are available:

- Fixed learning rate: its value is chosen by hand, and it remains the same for all the training procedure.
- Adapting learning rate: its value can be adapted automatically by evaluating the trend of the loss score: if it is oscillating, then the learning rate is reduced; whereas if the progress is steady, the learning rate can be increased.
- Separate adapting learning rate: it might be reasonable to have different learning rates for each edge of the network to update faster some weights, and slower some other ones.
- Alternative strategy to steepest descent: sometimes the direction of the current steepest descent in the error surface is not on the way to the minimum.

Now we introduce three optimizers that are widely used, which are known to be stable and reliable. These implement different strategies to update the weights during training.

Stochastic Gradient Descent

The stochastic gradient descent (SGD) algorithm [28][29] is an algorithm that apply backpropagation with an online strategy: the adjustment of the weights is done after each training example. In particular a single example $\{X^t, T^t\}$ is randomly (hence the name stochastic) chosen from the training set at every iteration t and the error E^t is computed based on the distance between the output of the network and the expected target value T^t. Then the weights are updated using the derivative of E^t evaluated by means of the backpropagation algorithm:

$$w_{j,i}(t+1) = w_{j,i}(t) + \eta \frac{\partial E^t}{\partial w_{j,i}} \tag{4.41}$$

Since the error is not averaged over a set of training cases, it's a noisy estimate, and the updates of the weights don't proceed smoothly down the error surface, but there are many oscillations (Figure 4.15). Furthermore the reached minimum may not be the global minimum, because of the oscillations. However, usually a point good enough for practical applications is reached, and SGD is a technique much faster than batch gradient descent. Another advantage of SGD is that it can handle more efficiently sets with redundant data. Moreover, the oscillations can be a way to avoid being stuck in local minima: indeed usually a local minimum for an entire dataset isn't a local minimum for a single training sample.

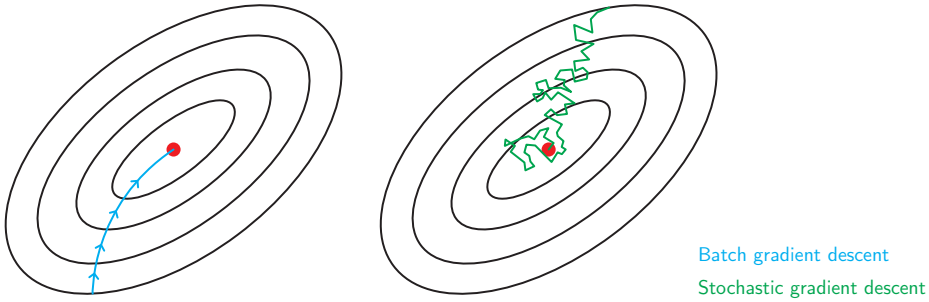


Figure 4.15: Full batch gradient descent and stochastic gradient descent comparison

Mini-batch gradient descent

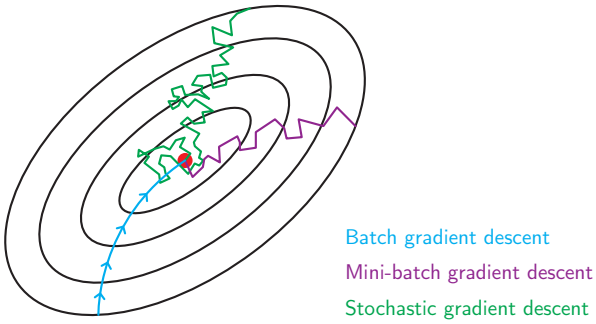
Mini-batch gradient descent [30] is a good compromise between full batch and online learning. Indeed the training samples are divided in batches. For each iteration the error is averaged over a batch, and the weights updated. Thus the rule to adjust the weights becomes

$$w_{j,i}(t+1) = w_{j,i}(t) + \eta \sum_m \frac{\partial E^m}{\partial w_{j,i}} \tag{4.42}$$

where m is the index of training sample in the current batch.

This method reduces the oscillations of the weights' updates leading to a more stable convergence. Furthermore, the forward step to compute the output of the model over all the samples of the batch can be performed with

vectorization techniques: this allows increasing the efficiency of the gradient evaluation. A common size for a mini-batch is between 50 and 256, and it depends on the application.



< Figure 4.16: Full batch gradient descent, stochastic gradient descent and mini-batch gradient descent comparison

Gradient descent with momentum

Gradient descent doesn't work very well if the error surface has an elongated shape, like the one in Figure 4.17. In a case like this one, it would be better to have a slower learning on the vertical axis and a faster learning on the horizontal axis. One way to smooth out the oscillations is to use the momentum technique [27][31], which evaluates the update of the weights $\Delta w_{j,i}$ considering also the previous one. Thus the update rule is

$$w_{j,i}(t+1) = w_{j,i}(t) + \Delta w_{j,i}(t+1) \tag{4.43}$$

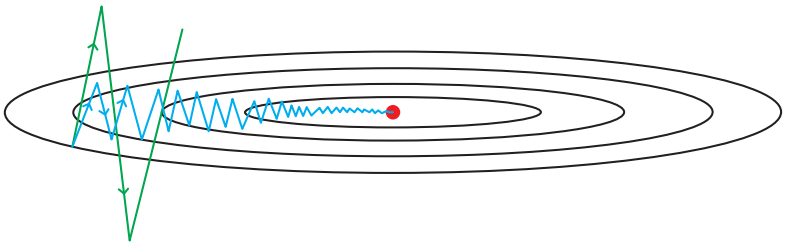
< (4.43)

and

$$\Delta w_{j,i}(t+1) = \gamma \Delta w_{j,i}(t) - \eta \frac{\partial E}{\partial w_{j,i}} \tag{4.44}$$

< (4.44)

where γ is a parameter called momentum factor with a range between 0 and 1. This factor weights the contribution of the previous update to the current one. We can think of the gradient term as an acceleration and $\Delta w_{j,i}(t)$ as a velocity multiplied by a friction term γ. Thus, this rule determines how the point is moving down the error surface.



< Figure 4.17: The green line represents the gradient descent without momentum and the blue one with momentum

In presence of many up and down oscillations, these will be averaged to a low value, whereas if the updates proceed in the same direction, the average value will remain high.

RMSprop

Another problem of the gradient descent is to choose a convenient learning rate, because an high learning rate may cause to overshoot the minimum, and a low one may cause a very slow training. Furthermore, this problem is emphasized by the fact that the magnitude of the gradients can be highly variable, and as a consequence their oscillations, making difficult to find the minimum. One solution is proposed by the Root Mean Square Propagation (RMSprop) algorithm [32].

For each weight, a running average MS of the magnitudes of recent gradients is computed as

$$MS(w_{j,i}, t) = \mu MS(w_{j,i}, t-1) + (1-\mu)\left(\frac{\partial E}{\partial w_{j,i}(t)}\right)^2 \quad < (4.45)$$

where μ is a parameter typically chosen equal to 0.9. In this case the update rule becomes

$$w_{j,i}(t+1) = w_{j,i}(t) - \eta \frac{\partial E}{\partial w_{j,i}} \frac{1}{\sqrt{MS(w_{j,i}, t) + \varepsilon}} \quad < (4.46)$$

Sometimes the parameter $\varepsilon = 10^{-8}$ is added to avoid division by 0. Dividing by the root of the running average helps to smooth out the oscillations enhancing the learning quality and speed. Indeed, an higher learning rate can be used to get faster training without diverging.

Adam

An algorithm which puts together the advantages of momentum and RMSprop is the Adaptive Moment Estimation (shortened as Adam) algorithm [33]. This methods keeps not only the average of past squared gradients, like the RMSprop method, but also the average of the past gradients, similarly to the momentum technique. To perform Adam algorithm, the following equations can be applied:

$$m(t) = \beta_1 m(t-1) + (1-\beta_1)\frac{\partial E}{\partial w} \quad < (4.47)$$

$$v(t) = \beta_2 v(t-1) + (1-\beta_2)\left(\frac{\partial E}{\partial w}\right)^2 \quad < (4.48)$$

where $m(t)$ is the first moment estimate and $v(t)$ is the second moment estimate. Then the bias-corrected estimates are computed as

$$\hat{m}(t) = \frac{m(t)}{1-\beta_1^t} \quad < (4.49)$$

$$\hat{v}(t) = \frac{v(t)}{1-\beta_2^t} \quad < (4.50)$$

with β_1^t and β_2^t being β_1 and β_2 to the power t.

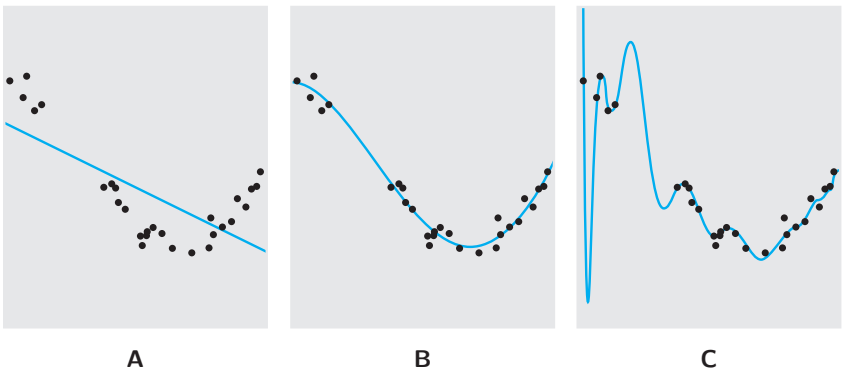
Finally the weight update rule is given by

$$w_{j,i}(t+1) = w_{j,i}(t) - \alpha \frac{\hat{m}(t)}{\sqrt{\hat{v}(t)} + \varepsilon} \qquad < (4.51)$$

The parameters α, β_1 and β_2 are respectively the step size and the exponential decay rates for the moment estimates (that have to be in the range $[0, 1)$). The default settings proposed are $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$ and $\alpha = 0.001$.

4.5 | Regularization

In addition to an effective and optimized training algorithm, we have also to ensure that the trained model generalizes well, that is it will make good predictions for data not seen during training. At the beginning of the training, the loss on both training and test data decreases, relevant patterns can still be learned and the model is underfitting (Figure 4.18). After some iterations, the number of which depends from case to case, the loss on the training decreases, but the loss on the test increases: this means that the model is learning patterns which are specific of the training samples, but not relevant as general regularities. In this case the model is overfitting (Figure 4.18). The training data contains not only the characteristics that the model should learn but also accidental characteristics caused by sampling errors (i.e. characteristics present in the dataset just because of the particular chosen samples). An overfitting model is learning both kinds of characteristics. Therefore, when it will be used to make new predictions on unseen data, it will also search for the accidental characteristics and thus it will make mistakes. A good model should not be too complex, and at the same time, it should fit well a wide set of cases.

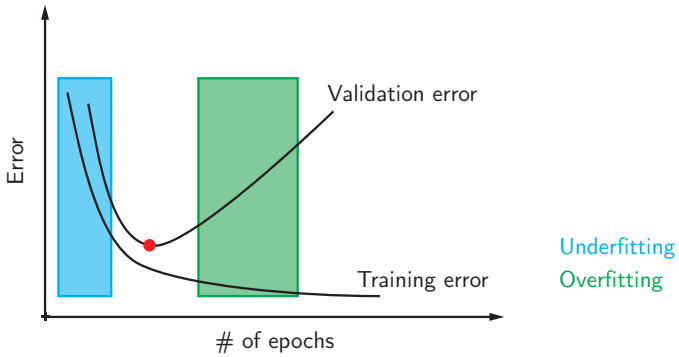


< Figure 4.18:
A) Underfitting
B) Proper generalization
C) Overfitting

Different ways to reduce overfitting are the followings:

- To get more data: increasing the number of samples, the model will naturally learn only the patterns that are present in most data
- To reduce the model's capacity: limiting the number of layers, or nodes per layer, the information that the model can learn is reduced. The more parameters the model has, the more it will be able to learn accidental characteristics. However, the capacity has to be high enough to avoid underfitting. Therefore, it is necessary to do some trials to find the right capacity, starting from a low number of parameters, and then gradually increasing it. This method works best with cross-validation, where the dataset is divided in three subsets, training, validation and test, and the number of parameters are adjusted using the validation data.
- Early stopping: the performances during the iterations are supervised, and training is stopped before the model overfits (Figure 4.19). Indeed, if we don't let the weights become high, the capacity of the model will be limited.
- Weight penalties: different types of penalties are used to scale down the weights to limit the capacity of the model.
- Weight constraints: some constraints are imposed to prevent the weights from growing beyond some predefined thresholds.
- Dropout: a given percentage of nodes are randomly selected. Their activations are not forwarded, and weight updates are not applied in the backward step. The network becomes less sensitive to the specific weights of neurons. This enhances the network generalization, by learning more internal representations.
- Noise: adding noise to the weights or to the activation values, the model is made less sensitive to small variations of the inputs, thus its robustness is increased.

Typically there isn't a single correct method, but a combination of different methods is used. In the following sections we introduce the main three methods that have been used for this work.



< Figure 4.19: Early stopping. A good situation is to stop where there is the red point

4.5.1 Weight penalties

The capacity of the model can be controlled by limiting the size of the weights. This is done by adding a penalty to the loss function E which prevents the weights from getting too big. The regularized loss function is [12]

$$\tilde{E}(w; X, T) = E(w; X, T) + \lambda R(w) \tag{4.52}$$

where λ is an hyperparameter whose value determines the contribution of the regularization term $R(w)$: if λ equals to 0, no regularization is applied, whereas the larger λ is, the more regularization is applied, and thus the more the model will be kept simple.

For different forms of $R(w)$ term we have different types of regularization:

- L1 regularization: $R(w)$ is proportional to the absolute value of the weights;
- L2 regularization: $R(w)$ is proportional to the square value of the weights. This method is sometimes called weight decay.

L2 regularization

The regularized loss function is given by

$$\tilde{E} = E + \frac{\lambda}{2} \sum_i w_i^2 \tag{4.53}$$

Then, the derivative of the error becomes

$$\frac{\partial \tilde{E}}{\partial w_{j,i}} = \frac{\partial E}{\partial w_{j,i}} + \lambda w_{j,i} \tag{4.54}$$

This derivative is equal to 0 when

$$w_{j,i} = -\frac{1}{\lambda} \frac{\partial E}{\partial w_{j,i}} \tag{4.55}$$

It is noticeable that at the minimum of the loss function, we have large weights if we have also big error derivatives. The network model will not use weights which are not necessary. In other words, only the parameters that make a significant contribution to the reduction of the loss function are preserved, the others are limited [12].

This type of regularization improves generalization because it helps preventing that the network fits the accidental characteristics. Furthermore, the regularized model will have less variations in the outputs when the inputs change.

L1 regularization

The regularized loss function is given by

$$\tilde{E} = E + \lambda \sum_i |w_i| \tag{4.56}$$

The derivative of the error becomes

$$\frac{\partial \tilde{E}}{\partial w_{j,i}} = \frac{\partial E}{\partial w_{j,i}} + \lambda \text{sign}(w_{j,i}) \tag{4.57}$$

Unlike L2 technique, the regularization term does not depend on the magnitude of each weight, but only on the sign of it. It can be demonstrated that L1 regularization may cause the weights to become sparse for large enough λ [12]. This property is useful for feature selection, where a subset of the available features has to be chosen.

4.5.2 Dropout

Dropout [34][35] is a regularization technique that consists of randomly setting to zero a percentage of the nodes of the network at each iteration of the gradient descent. During a training iteration, each remaining node should perform well even without the dropped nodes. This forces the nodes to represent the same patterns with different representations.

The percentage of dropped nodes has to be set to obtain a good compromise between an overfitting model and a model that can generalize properly. Usually it is set between 0.2 and 0.5, but it may vary from case to case.

Dropout can be thought as a method of training the ensemble of different subnetworks created by removing input or hidden nodes from the original network (Figure 4.20) [12]. In particular the subnetworks are not trained independently, but each one inherits some parameters from a parent model. This dependence makes possible to represent a huge amount of networks with a feasible computational memory.

To implement dropout, only the random selection of nodes has to be developed. Indeed, a dropped node has an activation set to 0, thus it does not contribute to the loss function. Instead, during test the predictions are computed with an average network of all the possible subnetworks. This is implemented by scaling the weights, modified by the dropout, by 1 minus the percentage of dropped nodes. This model is then used as a normal network without dropout [36]. A more efficient way to train with dropout has also been introduced in [37]: it is called fast dropout.

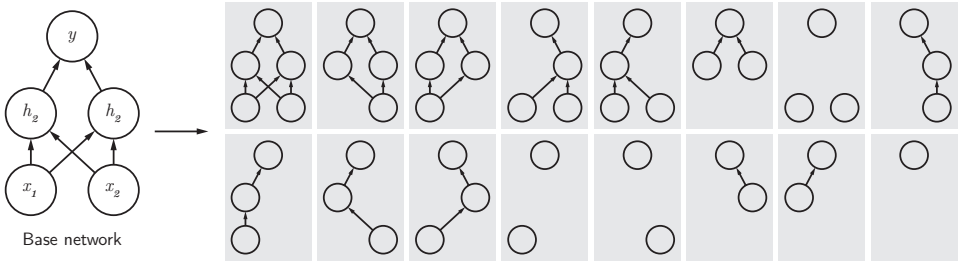
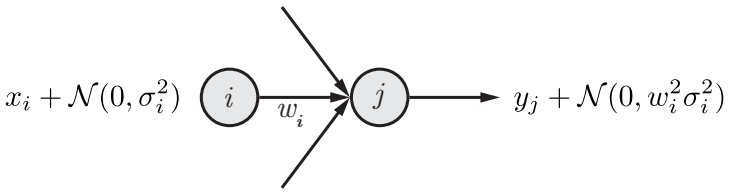


Figure 4.20: Dropout

4.5.3 Noise

Adding noise to the inputs of a network is equivalent to performing a regularization imposing weight penalties [38][39]. Indeed, if we add Gaussian noise to the inputs, we will have a noise added to the outputs whose variance is amplified by the squared weight (Figure 4.21). This additive noise makes a contribution to the loss function which can be considered as a regularization term added to the loss of the system without noise.

< Figure 4.21: Noise propagating in a node



Furthermore, adding noise to the weights is another way of restricting the capacity of a network. However, this is not equivalent to imposing weight penalties, and in some type of networks it may also work better. Adding noise to the weights is a method for improving the stability of the learned function [12]. Indeed, the weights are encouraged to move to regions of the weight space where small variations of the weights themselves result in small changes of the output.

Chapter 5
Language recognition

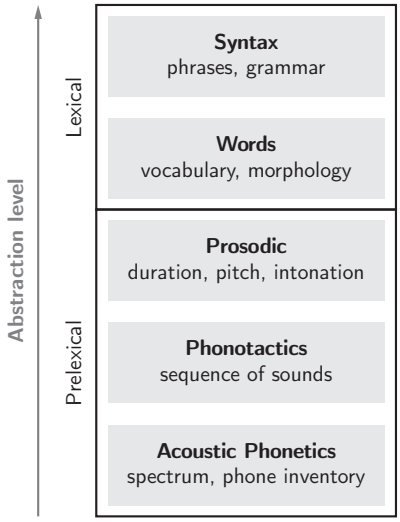
Language Recognition is the automatic process that tries to identify the language spoken in a given utterance. This technology is used in a wide range of applications: spoken language translation, multilingual speech recognition, emergency call routing, surveillance and security information distillation, or as front-end for language-dependent speech recognizers.

This chapter will present the main components of a whole language recognition system. First we will introduce the possible approaches to language recognition based on the intrinsic characteristics of languages. The next section will present data and metrics provided by the National Institute of Standards and Technology (NIST), which encourages the scientific research, and offers competitions to evaluate the current state of technology. Then, we will show an overview of the whole system, and a deeper analysis of its main components: the feature extractor and the classifier. Finally, we will present the experiments performed in this work, based on a neural network classifier.

5.1 | Language characterization

As it often happens, the human ability to accomplish a certain task is the primary source of inspiration. Humans can distinguish different language by means of two broad classes of cues: prelexical information and lexical semantic knowledge [40]. Among the prelexical information we have phonetic repertoire, phonotactics, rhythm, and intonation. As part of the lexical semantic knowledge we have the vocabulary, the meaning of the words, and the grammar. These two classes are both useful to the determination of the spoken language of a given content, however the lexical one is the most important only if the person knows the language of the test utterance. On the contrary, if the person doesn't know it, he will rely more on the prelexical class to understand at least the broad language group (e.g., tonal versus non-tonal languages).

The several language cues that are part of these two classes can be ordered according to their level of knowledge abstraction, as shown in Figure 5.1 [41].



< Figure 5.1: Levels of cues used for language recognition

Depending on the cues which are considered, it is possible to implement different language recognition systems based on: acoustic-phonetic, phonotactic, prosodic and lexical approaches. We now briefly introduce these approaches, but in this work we will focus on the acoustic-phonetic approach.

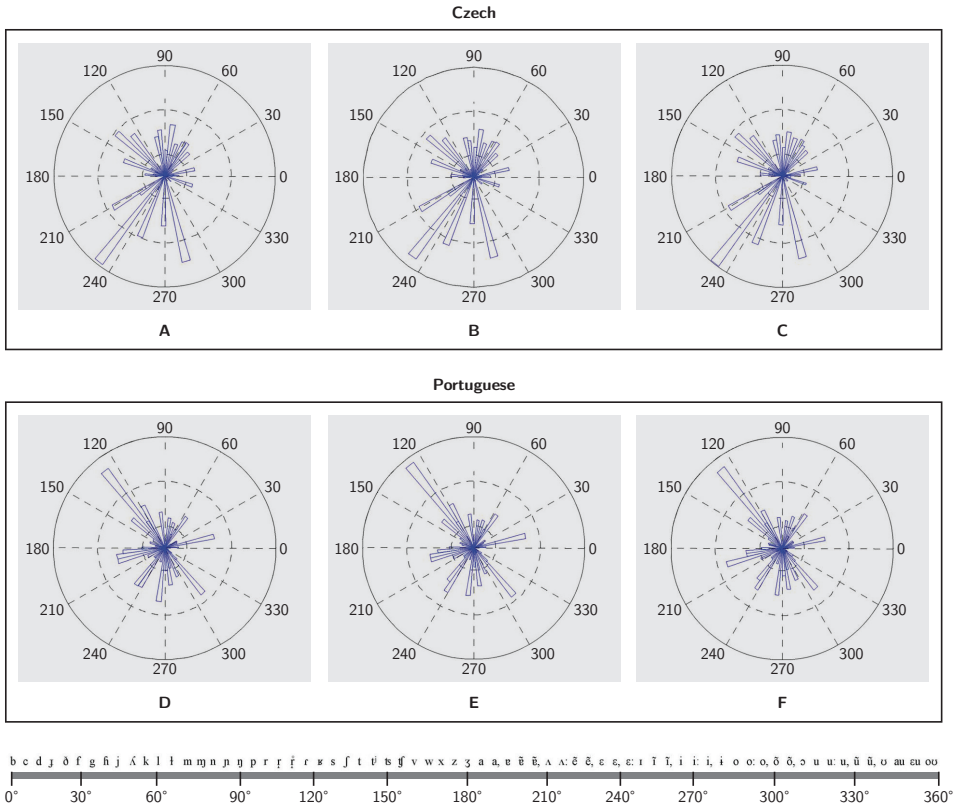
Acoustic-phonetic approach

Acoustic phonetics science studies the physical properties of speech. The sound signal is analyzed in time and frequency domain, measuring for example the amplitude, the duration, the fundamental frequency and the formants. These properties can be related to the linguistic concepts of phone and phoneme. The phone is the smallest elementary sound in a speech signal and the phoneme is the smallest unit sound of speech which is semantically significant [3]. A phone can be interpreted as a phoneme realization. Different phonemes can distinguish one word from another in a given language. For example in English the words “pat” and “bat” sound similar, but the different phonemes /p/ and /b/ differentiate the meaning of the two words.

Although the human speech production system can generate a wide range of sounds, in each language there is a limited number of sounds that recur. In particular the majority of languages has approximately 30 phonemes. The set of used phonemes changes between different languages, even if some phonemes may appear in more than one language. Thus, different languages can be distinguished based on the fact that they have different acoustic-phonetic distributions.

For example, in [41] Czech and Portuguese languages were considered for an experiment on their phone distributions. Different speakers utterances from the two languages were mapped to a shared set of symbols from the

International Phonetic Alphabet (IPA). Figure 5.2 shows the histograms of the phones' occurrences with polar plots. The three diagrams of the first row represent the utterance mapping of three Czech speakers, while the three of the second row represent mappings for Portuguese speakers. The three utterances have a different content, but the plots have a similar distribution, while there are evident differences among the plots of the two languages. So we can visually verify that different languages have a different phonetic repertoire.



< Figure 5.2: Polar histograms showing the phone distributions of Czech (A-C) and Portuguese (D-F) utterances for three different native speakers. The same language utterances have different contents

Phonotactic approach

Phonotactics is the study of the phonological rules that establish the permitted combinations of phonemes. For example, it defines the possible consonants and vowels sequences, and also some constraints, like having some phonemes at the beginning or the end of a word.

The phonotactic rules differ from one language to another, while the phonetic repertoire can be more similar between different languages.

For the purpose of language recognition, the phone sequences of different languages are predicted and compared by means of the n-gram model [26]. This approach computes the probability of a phoneme given the sequence of the previously spoken phonemes. However, instead of considering the entire history, the sequence is approximated by the last phonemes. In particular, a n-gram is a sequence which takes into account the last n phonemes.

For example, in [41] the bigram models of seven languages were built using the GlobalPhone database. Then, the accuracy of the models to predict phone sequences belonging to the seven languages was computed. The metrics adopted was the perplexity [26]. The perplexity is the inverse probability of the test set, normalized by the number of words. For a test set $W = w_1 w_2 ... w_N$ the perplexity of a bigram model is

$$PP(W) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i|w_{i-1})}} \quad (5.1)$$

For low perplexity the bigram model matches better the phone sequence and viceversa. As it is shown in Table 5.1, the bigram model of a language can better predict phone sequences of the same language.

Table 5.1: Perplexity measured between 7 languages arbitrarily selected from the GlobalPhone database based on bigram models

Test languages	Bigram model						
	CZ	FR	GE	KO	PO	SW	TU
Czech (CZ)	16	318	356	491	383	273	555
French (FR)	456	15	115	452	155	200	335
German (GE)	586	163	15	451	447	190	370
Korean (KO)	605	582	549	16	509	548	554
Portuguese (PO)	717	321	382	490	17	450	565
Swedish (SW)	416	424	162	506	584	16	670
Turkish (TU)	985	212	214	362	429	310	13

Prosodic approach

The prosodic aspects of speech don't concern characteristics at the phonemes level, but at larger time units level. The prosodic features are pitch, rhythm, stress, duration and intonation [3].

Different languages have different prosody constraints. Pitch variations can be led by lexical and syntactic conventions in some languages. For example, some words in Japanese have a fall of the pitch in correspondence of some phonemes. Rhythm can depend on syllables as in Italian (syllable-timed languages), stress as in English (stress-timed languages) or mora as in Japanese (mora-timed languages). Tone variations can distinguish different words in tonal languages, such as Mandarin Chinese [41].

Even if prosodic features are important characteristics of the languages, they are less informative than the acoustic-phonetic or phonotactic features in the distinction of different languages. They are mostly reliable to differentiate broad classes of languages, but not specific ones. Therefore, this approach has not yet become successful.

Lexical approach

Morphology is the study of the way words are formed in a certain language and syntax is the study of the structure of a sentence, so the ways in which words are connected.

Obviously a list of words or a model of the possible sequences of words (n-gram) can be useful to identify a language, because each one has a specific vocabulary and set of syntactic rules. Indeed lexical approach was implemented by Large Vocabulary Continuous Speech Recognition (LVCSR) systems. However, by considering again the analogy of the human capacity to recognize languages, we can point out that it is excessive to learn an entire language with the only purpose of distinguishing it from the others [41]. So the cost of these systems is not entirely justified. Furthermore it has been shown that LVCSR systems are sensitive to background noise, channel effects, accents, and more [42]. As a consequence, these systems are less used than others.

5.2 | NIST LRE17 dataset and performance metrics

The National Institute of Standards and Technology (NIST) is a physical sciences laboratory whose mission is to promote innovation and competitiveness between the industries. Its interests embrace a wide field of science and technology, from nanomaterials to global communication networks. Since 1996 NIST offers a series of competitions in the language recognition field to evaluate the current state of technology, and to search for new promising ideas. For this reason it provides a dataset of utterances from different languages, and imposes standard experimental protocols and performance metrics, which are both been used in this work. In particular we refer to the NIST 2017 Language Recognition Evaluation Plan (NIST LRE17) [43].

5.2.1 Dataset

A part of the dataset used in this work consists of the data provided by NIST LRE17, and another part comes from a variety of sources that we will briefly mention.

The data provided for NIST LRE17 are the following sets [43]:

- LRE data from previous year evaluations
- Fisher corpus
- Switchboard corpora
- LRE17 development set

Furthermore, other data are added with the aim to have many diverse channels, and a larger amount of data [44]. Thus part of the KALAKA-3 database, part of the Al Jazeera Dialectal Speech Corpus and IARPA Babel Program data were used. Furthermore, part of the data were also augmented. Reverberated speech and noise were mixed to the clean data to have an imposed SNR. Artificial room impulse responses were generated. Finally different noise sources were added with various SNR values: real stationary noises (like fans) and real transient noises (like ambient of city, library, office, etc.), babbling noises created mixing various speech segments and other ones artificially generated with transformations of white noise.

Since in the dataset provided by NIST many segments were considerably long, two versions of dataset were assembled. The first one, referred to as *full*, contains the segments without alterations. For the second one, speech segments lasting more than 40 seconds were cut into shorter segments of various lengths. This second version of the dataset is called *cuts*.

5.2.2 Performance metrics

In this work, the scores to evaluate the performance of the system are the metrics required for NIST LRE17 [43]. The cost function is calculated considering pairs of target-language (L_T) and non-target-language (L_N). It is needed to compute false rejects (i.e. missed detections) on the former and false alarms on the latter. These quantities are then combined using the following linear function

$$C(L_T, L_N) = C_{Miss} \times P_{Target} \times P_{Miss}(L_T) + C_{FA} \times (1 - P_{Target}) \times P_{FA}(L_T, L_N) \quad < (5.2)$$

where C_{Miss} is the cost of a missed detection, C_{FA} is the cost of a false alarm and P_{Target} is the a priori probability of a certain target language. The values of these parameters are fixed, and defined in Table 5.2.

< Table 5.2: Cost parameters

Parameter ID	CMiss	CFA	PTarget
1	1	1	0.5
2	1	1	0.1

Since $C(L_T, L_N)$ is not easy to interpret, it is normalized by the default cost $C_{Default}$. This is the cost which could be obtained if every segment would be declared to match the target language.

$$C_{Norm}(L_T, L_N) = \frac{C(L_T, L_N)}{C_{Default}} \quad < (5.3)$$

and

$$C_{Default} = C_{Miss} \times P_{Target} \quad < (5.4)$$

Thus combining (5.2), (5.3) and (5.4) we obtain the following cost function

$$C_{Norm}(L_T, L_N) = P_{Miss}(L_T) + \beta \times P_{FA}(L_T, L_N) \quad < (5.5)$$

where β is given by

$$\beta = \frac{C_{FA} \times (1 - P_{Target})}{C_{Miss} \times P_{Target}} \quad < (5.6)$$

In addition to the cost computed for the pairs, we consider also the average cost for the system which can be computed as

$$C_{avg}(\beta) = \frac{1}{N_L} \left\{ \sum_{L_T} P_{Miss}(L_T) + \frac{1}{N_L - 1} \left[\beta \times \sum_{L_T} \sum_{L_N} P_{FA}(L_T, L_N) \right] \right\} \quad < (5.7)$$

where N_L is the number of target languages. Using the two sets of cost parameters defined in Table 5.2 to compute β we obtain two values of C_{avg}, and the final value for the average cost function is computed as their average:

$$C_{primary} = \frac{C_{avg}(\beta_1) + C_{avg}(\beta_2)}{2} \quad < (5.8)$$

5.3 | System description

Given a speech segment, the task of the system considered in this work is to determine which is the most likely language spoken in that segment. The candidate languages are limited to the closed set defined in Table 5.3.

< Table 5.3: LRE17 target languages

Language cluster	Target language	Language code
Arabic	Egyptian Arabic, Iraqi Arabic, Levantine Arabic, Maghrebi Arabic	ara-arz, ara-acm, ara-apc, ara-ary
Chinese	Mandarin, Min Nan	zho-cmn, zho-nan
English	British English, General American English	eng-gbr, eng-usg
Slavic	Polish, Russian	qsl-pol, qsl-rus
Iberian	Caribbean Spanish, European Spanish, Latin American Continental Spanish, Brazilian Portuguese	spa-car, spa-eur, spa-lac, por-brz

The output of the system is a vector containing 14 values, one for each language, which represent the log-likelihood scores for the corresponding languages.

Given a target language model L_i, we can compute the log-likelihood score for an observation $\mathcal{O}$ as

$$l_i = \log\left(P(\mathcal{O}|L_i)\right) \tag{5.9}$$

and obtain the posterior probability by means of the Bayes' theorem

$$P(L_i|\mathcal{O}) = \frac{P(L_i)\exp(l_i)}{\sum_{j=1}^{N_L} P(L_j)\exp(l_j)} \tag{5.10}$$

where N_L is the number of target languages and $P(L_i)$ is the a priori probability for the i-th language.

To accomplish this task we used a system based on the acoustic-phonetic approach. Thus, the aim is to extract the acoustic characteristics from the audio sample, and then to classify them in one of 14 language classes. In particular, the system used in this work is composed by different subsystems:

- Stacked Bottleneck Neural network (SBN) feature extractor. These features substitute the MFCCs and derivatives (described in Section 2.3.2) because they take better into account the context of the samples. SBN features are introduced in the following Section 5.4.
- I-vector modelling (Section 3.4).
- One or more classifiers used in parallel. The main classifiers are a Gaussian linear classifier and a neural network, whose details are briefly introduced in Section 5.5.
- Calibration and fusion of the scores returned by the classifiers. The first stage is used to adjust the individual scores making sure that they are consistently meaningful across the input data. The second stage is used to enhance the capability of the different subsystems to recognize complementary features.

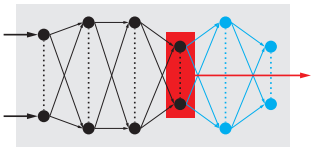
5.4 | Stacked Bottleneck features

The Stacked Bottleneck Neural network (SBN) framework to extract features has been proposed by the colleagues of Brno, with whom we participated to the NIST LRE 2017 [44].

A bottleneck layer of a neural network is an hidden layer which has a

number of nodes significantly lower than the near layers. The information that comes from larger layers is compressed in the bottleneck layer, and then expanded to proceed to the next larger layer. Using a bottleneck layer, the noise of the information is lowered and the system is more robust. Furthermore the risk of overfitting is attenuated.

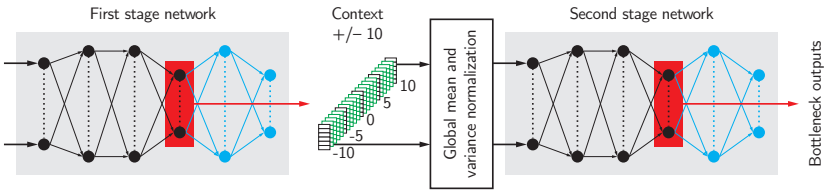
A bottleneck (BN) feature vector is the representation of the information extracted from the output of the bottleneck layer [45]. In other words, instead of considering the output of the entire NN, we take an intermediate transformation of the input vector. The NN is still trained for its final purpose, but then we will ignore all the layers following the bottleneck and consider the latter as the output of the NN. The features thus obtained are a representation of the information which has been compressed and transformed in a non-linear way.



< Figure 5.3: Network with a bottleneck layer

A stacked bottleneck (SBN) feature vector is obtained by a cascade of two such neural networks [45]. The BN vectors outgoing from the first NN are stacked in time and then sampled at 5 different time frames. These are the input for the following NN, which accomplishes an operation similar to the first NN. The final SBN feature vector is the output of the bottleneck layer of the second NN.

By stacking the BN vectors of the first NN, it is possible to retain the information about the context of a feature vector.

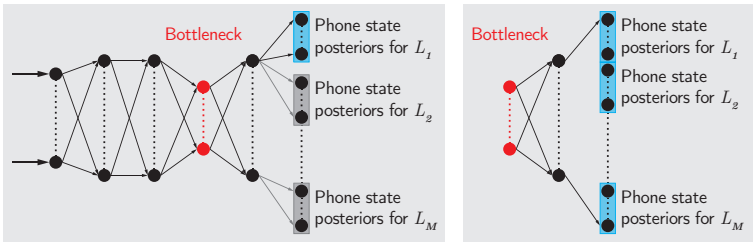


< Figure 5.4: Stacked bottleneck network

It has been proved [46] that training the NNs with data coming from one language at a time, the SBN feature vector obtained is monolingual, and the features are particularly tied to the single language. Whereas training the NNs on several languages simultaneously, the features extracted are more generic and perform better in the multilingual recognition task. Indeed in this case the NN is able to extract features in a more independent way from the single language and the acoustic space covered from the features is wider.

The block softmax technique is used for training the last layer to map the inputs to the different languages. The output layer is divided into parts corresponding to the set of target languages. During training, only the part which corresponds to the language of the input vector data is activated.

On the contrary, the technique using a single softmax maps the input to a large layer which is composed by the concatenation of all the possible language-phoneme couples. A bigger effort is employed to distinguish similar phonemes that belong to different languages, and the performance is affected. For this reason the block softmax approach works better.



< Figure 5.5: Schema of different output layer for multilingual training A) Block softmax B) One softmax

We report the specifications of the feature extractors developed in [44] for the data vectors that we will use in this work.

Input features

To compute the input features for the NN the audio signal is processed with a filter bank having 24 bands, which are defined according to the Mel scale. Then the logarithm is applied to the outputs of the filter bank, and 2 fundamental frequency coefficients are added to these 24 values. The resulting feature vector is subtracted by the mean. The coefficients are signals in the time domain. Hamming window followed by DCT is performed to extract 6 parameters for each value. The result is a feature vector of size $26 \cdot 6 = 156$.

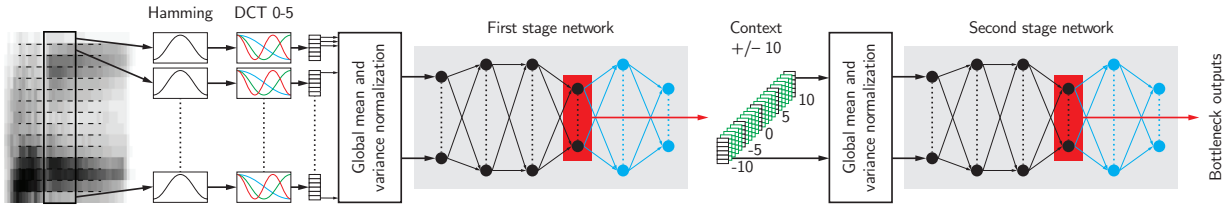
Neural network architecture

Both NN are feed-forward neural networks with hidden layers of size 1500 and bottleneck layer of size 30 or 80. The outputs of the bottleneck layer of the first NN are sampled at times $t-10$, $t-5$, t, $t+5$, $t+10$, where t is the current frame index. These 5 frames are stacked together, and are used as input for the second NN, which has the same architecture of the first (if not specified differently in the following). The final features are the outputs of the bottleneck layer of the second NN.

Three architectures are used to extract different types of SBN features:

- FSH-30: the network is trained on the Fisher English corpus. The first NN has a layer configuration given by 3 hidden layers, and a bottleneck with size 80 directly connected to the output layer. The second NN has the same configuration, but the bottleneck has size 30.
- FSH-80: the network is trained on the Fisher English corpus. The first NN has a layer configuration given by 2 hidden layers, a bottleneck of size 80, and 1 hidden layer before the output.
- BabelML17-80: the network is trained on 17 languages from BABEL project. The architecture is the same as FSH-80, but the network is trained in multilingual mode with block softmax.

Figure 5.6 reports a scheme for the entire feature extractor.



∧ Figure 5.6: Schema of SBN feature extractor

5.5 | Classifiers

Since the aim of this work is to classify feature vectors of various languages, we considered a simple Gaussian linear classifier as reference system, and we tested different architectures of neural networks.

5.5.1 Gaussian linear classifier

We can assume that the i-vector point estimates for each language l are generated by a random variable

$$X_i \sim \mathcal{N}(\mu_l, \Lambda^{-1}) \qquad < (5.11)$$

where μ_l is the mean vector which depends on the language l and Λ^{-1} is the precision matrix. Instead of working with the covariance matrix, it is more convenient to use the precision matrix:

$$\Lambda = \Sigma^{-1} \qquad < (5.12)$$

Furthermore, we assume that the precision matrix is shared among all the language distributions.

Applying Maximum-Likelihood estimation on the dataset, we can get the parameters of the model which are the ones that maximize the log-likelihood. In other words, we can find the model under which the dataset is more likely to be generated. Thus given the language l, the class-conditional log-likelihood of X_l is written as

$$\log\left(P(X_i|l)\right) = \frac{1}{2}\log|\Lambda| - \frac{1}{2}(X_i - \mu_l)^T \Lambda (X_i - \mu_l) + k \qquad < (5.13)$$

where k is a constant which depends on the data.

5.5.2 Neural network

The basic architecture for the neural networks with which we experimented is a feed-forward network. As input it receives an i-vector, and outputs a vector with the probabilities that the input belongs to each target language. Different number of layers and nodes have been tested to find the best network. Furthermore, various regularization approaches have been tested.

5.6 | Experimental results

We started from the most basic neural network, composed by an input layer, a single hidden layer of 400 nodes with ReLU activation function, and an output layer with softmax activation function. We chose categorical crossentropy as loss function, and Adam as optimizer. We performed a set of experiments changing some parameters, we picked up the network with the best results, and then we did another set of experiments using the best network. Using this protocol, we were able to progressively enhance the network without losing the meaning of the variations.

All the scores that will be reported were obtained training the networks with data from the *cuts* set, in which the maximum length of the audio segments is 40s. We also experimented with the *full* set, but the results were always worse.

The results for the first set of experiments are shown in Table 5.4. In particular, the reported values are the minimum error rate and the corresponding epoch. Both performance metrics for development data and for evaluation data are presented.

< Table 5.4: Results for networks with one layer and different number of nodes

Architecture	Development		Evaluation	
	C_{avg} [%]	epoch	C_{avg} [%]	epoch
1 layer with 400 nodes	6.07	7	22.64	26
1 layer with 800 nodes	5.68	48	**22.38**	41
1 layer with 1600 nodes	5.90	4	22.48	47
1 layer with 2400 nodes	5.74	5	22.77	48
1 layer with 3200 nodes	6.34	4	23.41	3

The best network is the one with one hidden layer and 800 nodes. Thus small gradual variations in the parameters are applied to this network to improve it. In particular the number of layers is increased by 1 at each test. Furthermore, dropout with different amounts is applied on each one of the

hidden layers to regularize the network. In fact, we can notice from Table 5.4 that the error rates on the evaluation data are higher than the ones on the development data, which means that the network is overfitting.

Architecture	Dropout	Development		Evaluation	
		C_{avg} [%]	epoch	C_{avg} [%]	epoch
2 layers with 800 nodes	0	6.23	29	24.25	35
2 layers with 800 nodes	0.2	6.12	39	22.84	6
2 layers with 800 nodes	0.5	5.55	17	**21.73**	21
3 layers with 800 nodes	0	6.50	18	24.23	1
3 layers with 800 nodes	0.2	6.12	42	23.64	35
3 layers with 800 nodes	0.5	5.66	42	21.97	21
4 layers with 800 nodes	0	6.61	33	24.80	24
4 layers with 800 nodes	0.2	6.31	18	23.52	3
4 layers with 800 nodes	0.5	6.12	44	22.63	6
5 layers with 800 nodes	0.5	7.46	49	22.97	6
6 layers with 800 nodes	0.5	9.18	36	23.08	8
7 layers with 800 nodes	0.5	10.63	16	23.73	16
8 layers with 800 nodes	0.5	12.08	28	24.21	12

< Table 5.5: Results for different percentages of dropout in networks with different number of layers

From the results of Table 5.5 we can notice that a dropout of 0.5 is improving the performances of the network, thus in the next test we trained networks with this percentage of dropout, and different number of layers and nodes. The results are shown in Table 5.6.

Furthermore, we did another test on the network with 2 layers and 800 nodes adding dropout also on the input layer. The error rates are presented in Table 5.7.

Architecture	Dropout	Development		Evaluation	
		C_{avg} [%]	epoch	C_{avg} [%]	epoch
2 layers with 400 nodes	0.5	5.74	20	**21.51**	4
3 layers with 400 nodes	0.5	5.57	46	21.65	5
4 layers with 400 nodes	0.5	6.04	34	21.72	20
5 layers with 400 nodes	0.5	6.53	50	22.17	33
2 layers with 1600 nodes	0.5	5.44	32	22.23	2
3 layers with 1600 nodes	0.5	5.60	34	22.32	34
4 layers with 1600 nodes	0.5	7.13	32	22.54	6
5 layers with 1600 nodes	0.5	8.85	22	23.45	7

< Table 5.6: Results for a fixed percentage of dropout on networks with different number of layers and nodes

Architecture	Dropout	Initial dropout	Development		Evaluation	
			C_{avg} [%]	epoch	C_{avg} [%]	epoch
2 layers with 800 nodes	0.5	0.1	5.27	48	**20.99**	4
2 layers with 800 nodes	0.5	0.2	5.79	50	**20.99**	9
2 layers with 800 nodes	0.5	0.3	7.70	47	21.47	7
2 layers with 800 nodes	0.5	0.4	10.49	44	22.31	34

< Table 5.7: Results for different percentages of dropout on the input layer

Decreasing the number of nodes from 800 to 400 the error rate also decreases from 21.73 to 21.51. Moreover, adding the initial dropout decreases the error rate from 21.73 to 20.99. Putting these configurations together, and varying some parameters we did the tests shown in the following Table 5.8.

Architecture	Dropout	Initial dropout	Development		Evaluation	
			C_{avg} [%]	epoch	C_{avg} [%]	epoch
2 layers with 400 nodes	0.5	0.1	5.44	48	20.82	18
2 layers with 400 nodes	0.5	0.2	6.23	44	**20.81**	16
3 layers with 400 nodes	0.5	0.1	5.87	44	21.30	26
3 layers with 400 nodes	0.5	0.2	7.24	50	20.94	23
4 layers with 400 nodes	0.5	0.1	6.45	44	21.38	24
4 layers with 400 nodes	0.5	0.2	8.63	50	21.57	23

< Table 5.8: Results for different percentages of initial dropout in networks with various numbers of layers

We also tried to train networks with layers having different number of nodes. The results are presented in Table 5.9, and the architecture is indicated by the number of nodes of each layer in square brackets. For example, [800] [400] represents a network with one layer of 800 nodes followed by one layer of 400 nodes.

Architecture	Dropout	Initial dropout	Development		Evaluation	
			C_{avg} [%]	epoch	C_{avg} [%]	epoch
[800] [400]	0.5	0	5.63	21	21.76	8
[400] [800]	0.5	0	5.68	35	21.94	24
[800] [400] [800]	0.5	0	5.79	47	22.22	3
[400] [800] [400]	0.5	0	5.49	50	21.86	6
[400] [800] [400]	0.5	0.2	7.49	45	**21.12**	22

< Table 5.9: Results for networks having layers with different number of nodes

Mixing various number of nodes did not perform well, thus we discarded this test.

So far the activation function used for the hidden nodes was the ReLU function. For the following set of experiments (Table 5.10) different activa-

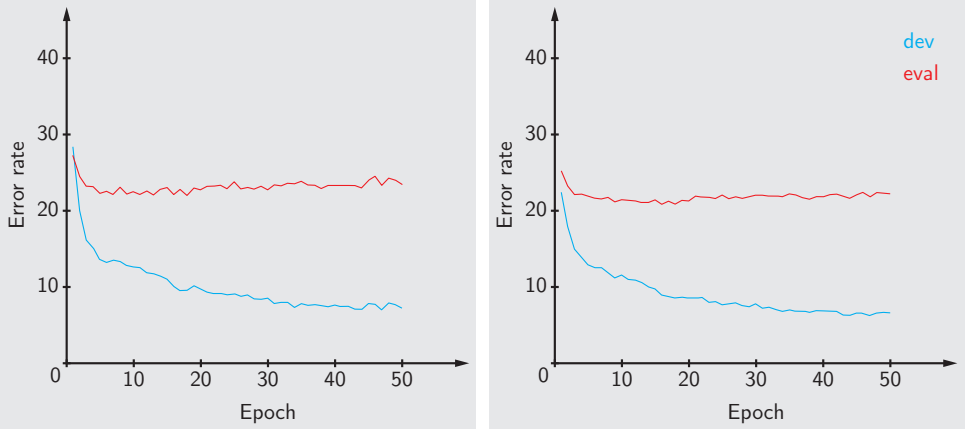
tion functions were tested. Furthermore, we tried also to apply L2 regularization because the network was still overfitting, and we obtained the results shown in Table 5.11. For the experiment results reported in Table 5.10 and Table 5.11, the network had a dropout of 0.5 on the hidden layers, and an initial dropout of 0.2.

Architecture	Activation function	Development		Evaluation	
		C_{avg} [%]	epoch	C_{avg} [%]	epoch
2 layers with 400 nodes	ReLU	6.23	44	20.81	16
2 layers with 400 nodes	tanh	9.70	50	21.18	39
2 layers with 400 nodes	sigmoid	8.01	50	**20.58**	18

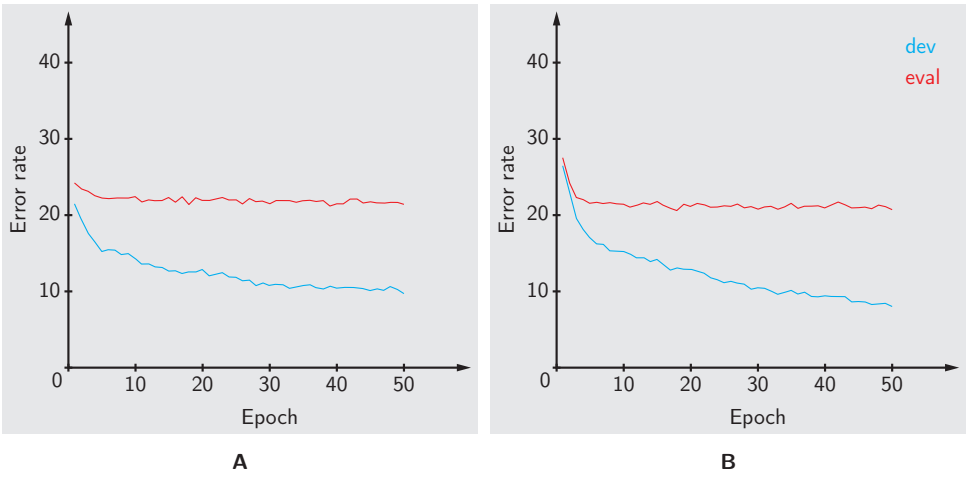
< Table 5.10: Results for networks with different activation functions

Architecture	L2 regularization	Development		Evaluation	
		C_{avg} [%]	epoch	C_{avg} [%]	epoch
2 layers with 400 nodes	0	6.23	44	**20.81**	16
2 layers with 400 nodes	1E-04	6.61	48	20.87	12
2 layers with 400 nodes	1E-03	11.48	49	21.30	11

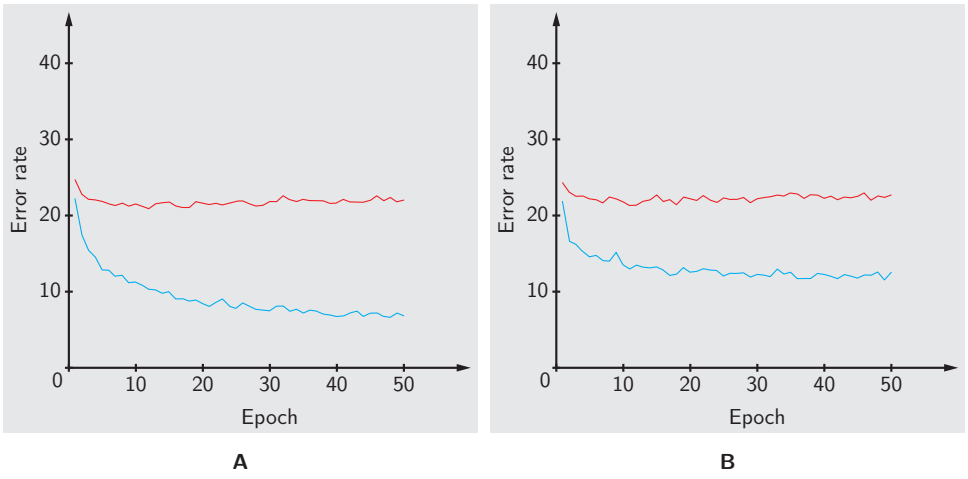
< Table 5.11: Results for networks with different amounts of L2 regularization



< Figure 5.7: Network with ReLU activation function (first row in Table 5.10) A) Raw data B) Calibrated data



< Figure 5.8:
A) Network with tanh activation function (second row in Table 5.10)
B) Network with sigmoid activation function (third row in Table 5.10)
Both graphs report calibrated data



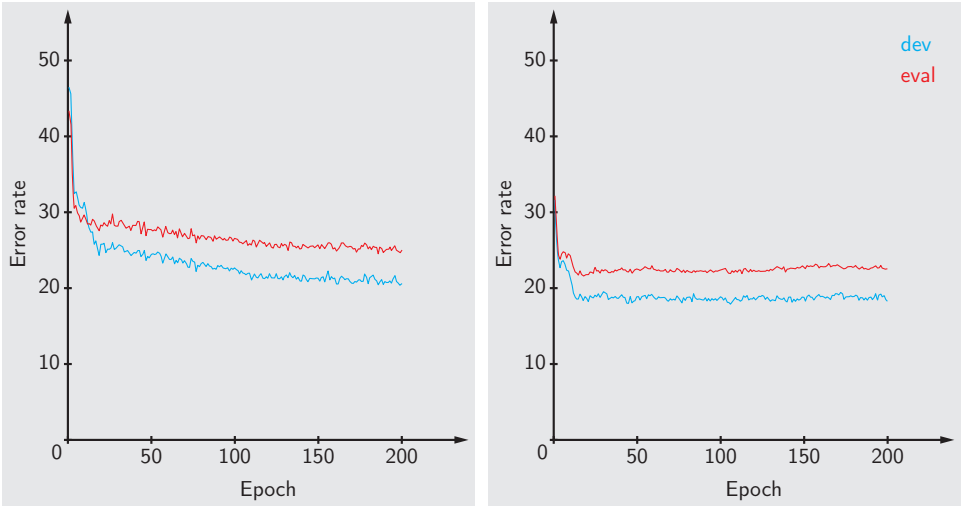
< Figure 5.9:
A) Network with L2 regularization equals to 0.0001 (second row in Table 5.11)
B) network with L2 regularization equals to 0.001 (third row in Table 5.11)
Both graphs report calibrated data

So far the best network has 2 hidden layers with 400 sigmoidal nodes, a dropout of 0.5 on the hidden layers, and a dropout of 0.2 on the input layer. The error rate of such network is 20.58 %. We then tried different number of layers, and various percentages of dropout and regularization. Since no significant improvement was measured, we don't report all these results for brevity, but only the two most relevant.

In the first test (Table 5.12) we added a random noise with normal distribution to the input vector, that changed at each epoch. The aim of adding the noise was to perform a simple data augmentation, and to regularize the network. However, the network took many more epochs to converge, in fact the error rate had many small oscillations (as it is shown in Figure 5.10), and the final result wasn't remarkable.

Architecture	Regularization	Development		Evaluation	
		C_{avg} [%]	epoch	C_{avg} [%]	epoch
2 layers with 400 nodes	noise	17.87	106	**21.58**	18
4 layers with 400 nodes	noise	24.34	196	25.58	196
6 layers with 400 nodes	noise	25.74	198	26.62	198

< Table 5.12: Results for different networks with noisy inputs



< Figure 5.10: Network with 2 layers of 400 nodes and noisy inputs (first row in Table 5.12) A) Raw data B) Calibrated data

In the second test (Table 5.13) we setup an architecture with a bottleneck in the center. In particular, we had four layers with 400 (or 800) nodes, one small layer with 40 nodes, and other four layers with 400 (or 800) nodes. Thus, the information from the previous larger layers was compressed to pass through the bottleneck, and then expanded to the following layers. In this way, the unnecessary features are discarded, and the risk of overfitting is attenuated. However, also in this case the result was disappointing.

Architecture	Development		Evaluation	
	C_{avg} [%]	epoch	C_{avg} [%]	epoch
4x[400] [40] 4x[400]	6.86	26	**24.41**	34
4x[800] [40] 4x[800]	6.80	45	24.45	14

< Table 5.13: Results for networks with a bottleneck layer

For the next phase of experiments, a preprocessing was applied to the input data. We started again with a basic network, and we made small changes on the resulting best configurations. During the first set of tests we used a simple architecture with one layer just changing the number of nodes (Table 5.14).

Architecture	Development		Evaluation	
	C_{avg} [%]	epoch	C_{avg} [%]	epoch
1 layers with 400 nodes	13.17	47	19.51	9
1 layers with 800 nodes	11.86	50	**19.48**	11
1 layers with 1600 nodes	10.60	49	19.74	3
1 layers with 2400 nodes	9.64	48	19.64	3
1 layers with 3200 nodes	8.96	49	19.78	4

< Table 5.14: Results for networks with different number of nodes (preprocessed data)

As we can notice from the results, the error rates are very close. Therefore, for the following tests we considered a wide range of configurations with different number of layers and nodes, different activation functions and different techniques of regularization. The results shown below are the most meaningful ones. They were obtained with layer having 400 nodes. For brevity we omit the tests with networks having layers with a larger number of nodes.

We report below the results for the test with different number of layers (Table 5.15).

Architecture	Development		Evaluation	
	C_{avg} [%]	epoch	C_{avg} [%]	epoch
2 layers with 400 nodes	7.13	43	**19.48**	7
4 layers with 400 nodes	6.99	39	19.94	3
6 layers with 400 nodes	7.24	50	20.32	2
8 layers with 400 nodes	7.60	46	20.40	8
10 layers with 400 nodes	8.44	50	20.71	3

< Table 5.15: Results for networks with different number of layers (preprocessed data)

Also in this case we can notice that the network is overfitting, thus in the following test we applied some dropout to regularize the network (Table 5.16).

Architecture	Dropout	Initial dropout	Development		Evaluation	
			C_{avg} [%]	epoch	C_{avg} [%]	epoch
2 layers with 400 nodes	0.2	0	8.69	49	19.23	7
2 layers with 400 nodes	0.2	0.1	13.93	47	20.15	3
2 layers with 400 nodes	0.2	0.2	16.53	49	20.57	1
2 layers with 400 nodes	0.5	0	13.58	49	**19.04**	7
2 layers with 400 nodes	0.5	0.1	15.85	25	19.87	7
2 layers with 400 nodes	0.5	0.2	17.24	25	20.46	2

< Table 5.16: Results for networks with different percentages of dropout applied to the hidden layers and to the input layer (preprocessed data)

We can notice that dropout is regularizing the network, and the evaluation scores are closer to the development ones. Compared to the case without preprocessing of the data (Table 5.8), the initial dropout is worsening the recognition. We hypothesize that the preprocessing is returning input vectors with more meaningful information and less noise. Thus, randomly dropping out input nodes is no more helpful.

Then we experimented with different activation functions and different regularizations. The results are summarized in Table 5.17 and Table 5.18.

Architecture	Dropout	Activation function	Development		Evaluation	
			C_{avg} [%]	epoch	C_{avg} [%]	epoch
2 layers with 400 nodes	0.5	ReLU	13.58	49	**19.04**	7
2 layers with 400 nodes	0.5	tanh	15.19	41	19.23	7
2 layers with 400 nodes	0.5	sigmoid	15.33	45	19.46	7

< Table 5.17: Results for networks with different activation functions (preprocessed data)

In Table 5.18, all the networks have nodes with ReLU activation, and a dropout on the hidden layers equal to 0.5.

Architecture	Regularization	Development		Evaluation	
		C_{avg} [%]	epoch	C_{avg} [%]	epoch
2 layers with 400 nodes	L2 1e-4	15.60	38	**18.95**	7
2 layers with 400 nodes	L2 1e-3	16.91	7	19.41	7
2 layers with 400 nodes	L2 1e-4 + noise	17.54	1	20.83	2

< Table 5.18: Results for networks with different types and amounts of regularization (preprocessed data)

Therefore the best network we obtained has 2 hidden layers with 400 nodes and ReLU activation, dropout of 0.5 applied to the hidden layers, and L2 regularization with amount 0.0001.

Finally, we compare this result with the ones reported in [44] for a Gaussian linear classifier, and for a neural network.

Classifier	Dataset	C_{avg} [%]
Gaussian linear classifier	full	19.55
Previous neural network	cuts	19.90
Actual neural network	cuts	18.95

< Table 5.19: Results obtained in [44] for two different classifiers and result for the network proposed in this work

We can conclude that the neural network proposed in this work perform slightly better than the previous classifiers.

Chapter 6
Speaker recognition

Speaker recognition is the automatic process aiming at assessing the identity of the speaker of a given utterance. This technology can be adopted in a wide range of applications: authentication procedures, forensic activities, audio indexing, telephone-based services, and speaker diarization.

This chapter will present the main components of a whole system of speaker recognition. We first introduce different approaches to speaker recognition. We then describe specifically the components of the system on which we focused in this work. Next section will introduce state-of-the-art classifiers, and a Siamese architecture for neural network that we will use for classification of pairs of i-vectors. Then, we will describe the dataset composition, and the error rate measure that has been used for the experiments. Finally, we will present a set of experiments performed with the aim of improving the speaker recognition performance using a Siamese architecture.

6.1 | Introduction

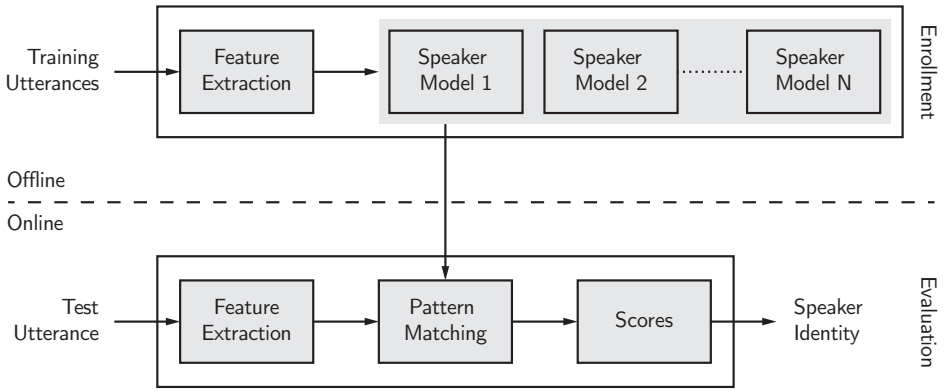
Just like fingerprint, face and retina, human voice has physiological characteristics closely linked to an individual. The uniqueness of a speaker voice is due to physical characteristics, but also to acquired characteristics. The physical characteristics are differences in shape and size of the voice production organs (for example vocal tract or larynx). The acquired characteristics are attributable to the different manner of speaking of each person, such as a particular accent, rhythm, intonation or vocabulary. Part of these characteristics are used by state-of-the-art speaker recognition systems.

Speaker recognition can be categorized in two main branches: speaker identification and speaker verification [47].

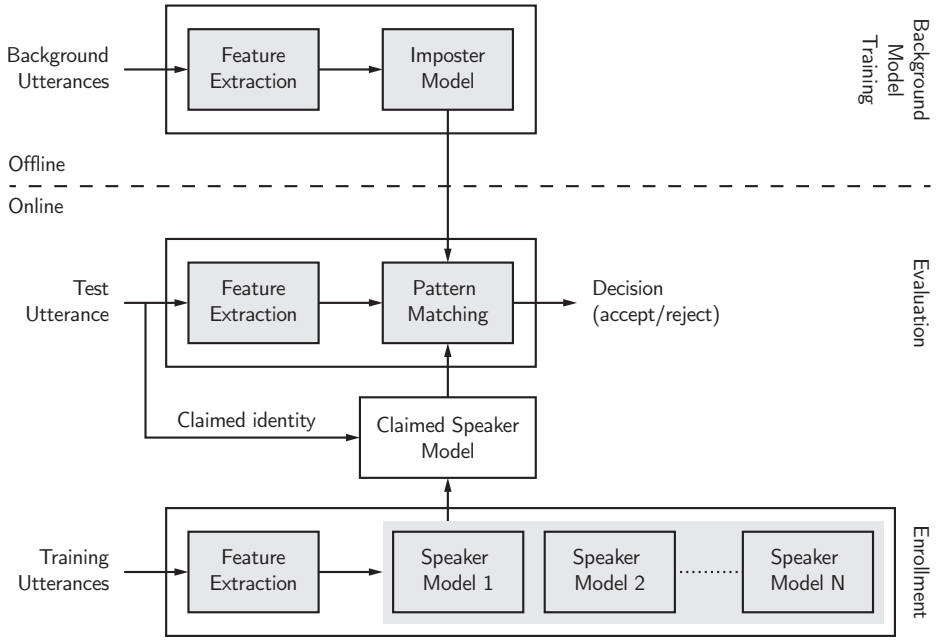
The task of a speaker identification system is to identify if a given utterance belongs to one among a given set of pre-enrolled speakers. This is a multiclass classification problem, where each member of the enrolled set represents a class. If the speaker of the test utterance is known to belong

to the enrolled group, it's a closed-set problem, otherwise it's an open-set problem, which is more difficult to deal with. A speaker identification process includes two main steps: enrollment and recognition (Figure 6.1). In the former step, training utterances are given, and an acoustic model for each speaker is estimated. This process is performed offline. For recognition, a test utterance is given, which is compared with all the enrolled models, and an estimate of the speaker's identity is returned.

< Figure 6.1: Speaker identification system



In speaker verification, the speaker declares his identity, and the task of the system is to validate the claimed identity. This is a binary classification problem, where the claimed speaker model is compared to an impostor model (or background model), and the returned response is an acceptance or rejection of the claim with a given confidence measure. While in speaker identification a 1-to-N comparison is performed, in speaker verification the comparison is 1-to-1. Usually, a speaker verification process includes three main phases: impostor modelling, speaker enrollment, and evaluation (Figure 6.2). The impostor model is trained offline with utterances from many different speakers. The enrolled speaker models are obtained by adapting the UBM. During evaluation, the claimed speaker model and the impostor model are compared and a score is returned. The final decision is obtained using a threshold: if the score exceeds the threshold, the speaker is accepted and viceversa.



< Figure 6.2: Speaker verification system

A speaker recognition system can be text-dependent or text-independent. In the first case, the speaker utterance is known in advance. For enrollment, a verification system asks the speaker to pronounce, one or more times, a specific phrase. The user has to pronounce this "passphrase" for future authentications. For a text-independent system, instead, the content of the speaker utterance is unknown.

In this work we will focus on text-independent speaker verification.

6.2 | System description

The whole speaker recognition system is composed by the following parts:

- Feature extraction: the speech signal is digitized and reduced to a set of feature vectors, in which the relevant information is emphasized and the redundant one discarded (Chapter 2).
- I-vector modelling: statistical models are used to estimate the distribution of the features (Section 3.4). For each enrolled speaker, we aim at building a unique "voiceprint".
- Classifier: the model obtained from the test utterance and a target model are compared to give a score. In particular, in Section 6.3 we will consider different pairwise classification approaches.

The enrollment dataset of NIST 2012 SRE has been used for training the neural networks. Since we adopt the pairwise approach, given a set of n i-vectors, the number of pairs in the training set would be n^2. However to make the training computationally feasible we use a strategy described in Section 6.4.

Feature extraction was performed as follows [48]: each segment of the dataset was submitted to Voice Activity Detection, then was framed in intervals of 10 ms, and 19 MFCC coefficients were extracted using a 25 ms sliding Hamming window. Short time mean and variance normalization was performed on the resulting vector, with a sliding window of 3s. Then, by stacking 18 cepstral coefficients, their 19 first derivatives, and their 8 second derivatives, a final 45 dimension feature vector was obtained.

The hyper-parameters of the UBM, and the i-vector model were trained using the NIST SRE 2004-2010 datasets, the data from the Switchboard II (Phases 2 and 3), and the data from the Switchboard Cellular (Parts 1 and 2). Thus in total the used utterances were 66140. The i-vector dimension was set to 400.

6.3 | Classifiers

Given a test utterance, the classifier has to determine whether this utterance belongs to a target speaker. This task can be solved with two different approaches. The first one consists in training the classifier in a one-versus-all scheme, where the different speakers are considered as different classes. For example, to discriminate between segments spoken by the target and segments spoken by different speakers we can train a different SVM for each target [49]. This technique, however, requires many utterances of each speaker to effectively train the classifier, but usually we only have a few of them. Indeed, we could only have the utterance spoken during the first user's enrollment.

For this reason, a more effective approach is training the classifier in a pairwise mode, i.e., using as input pairs of speech segments rather than a single one. The classifier will then determine whether two speech segments belong to the same speaker or to different speakers. Thus, a multiclass classification problem is transformed to a binary classification problem where the two classes are *same-speaker* and *different-speaker*.

In this section we present four different pairwise classifiers.

6.3.1 Probabilistic Linear Discriminant Analysis

Probabilistic Linear Discriminant Analysis (PLDA) [50][51] is a generative model that can be used for i-vectors comparison. In particular, it assumes that an i-vector can be generated by the following model

$$\phi_r = m + U_1 x_1 + U_2 x_{2_r} + \varepsilon_r \tag{6.1}$$

where ϕ_r is the i-vector for the utterance r, x_1 and x_{2_r} are latent variables which represent the speaker identity and the channel effects, respectively, and ε_r is the residual noise. As in the JFA model, matrices U_1 and U_2 restrict the speaker and channel factors to lie in small subspaces. It is also assumed that segments of the same speaker have in common the same latent variable x_1. Latent variables x_1 and x_{2_r} are assumed to obey prior distributions. U_1, U_2 and the hyper-parameters of these distributions are estimated maximizing the likelihood of the observed i-vectors.

The Gaussian PLDA (GPLDA) assumes that the latent variables have Gaussian distribution

$$x_1 \sim \mathcal{N}(0, I) \tag{6.2}$$

$$x_{2_r} \sim \mathcal{N}(0, I) \tag{6.3}$$

$$\varepsilon_r \sim \mathcal{N}(0, \Lambda^{-1}) \tag{6.4}$$

This is the simplest PLDA technique and thus it can fails to accurately model i-vectors (as shown in [51]). A more complex model, called heavy-tailed PLDA (HTPLDA), has been proposed, which assumes that the priors have Student's t-distribution

$$x_1 \sim \mathcal{N}(0, u_1^{-1} I) \qquad u_1 \sim \mathcal{G}\left(\frac{n_1}{2}, \frac{n_1}{2}\right) \tag{6.5}$$

$$x_{2_r} \sim \mathcal{N}(0, u_{2_r}^{-1} I) \qquad u_{2_r} \sim \mathcal{G}\left(\frac{n_2}{2}, \frac{n_2}{2}\right) \tag{6.6}$$

$$\varepsilon_r \sim \mathcal{N}(0, v_r^{-1} \Lambda^{-1}) \qquad v_r \sim \mathcal{G}\left(\frac{\nu}{2}, \frac{\nu}{2}\right) \tag{6.7}$$

However this approach is computationally expensive. It was shown in [52] that GPLDA and HTPLDA achieve comparable performances with an appropriate pre-processing of the i-vectors by a simple length normalization. Being GPLDA faster, it is usually preferred.

Two-covariance model

The two-covariance model [53] is a simplification of GPLDA in which the speaker and channel subspaces extend to the entire i-vector space. In particular an i-vector is modelled as

$$\phi = \boldsymbol{y} + \boldsymbol{z}_r \tag{6.8}$$

where $\boldsymbol{y}$ is the speaker factor and $\boldsymbol{z}_r$ is the factor which takes into account the channel effects and the residual noise. Furthermore, the speaker's prior and the distribution of ϕ given the speaker are assumed to be Gaussians

$$P(\boldsymbol{y}|\mathcal{M}) = \mathcal{N}(\boldsymbol{y}|\boldsymbol{\mu}, \boldsymbol{B}^{-1}) \tag{6.9}$$

$$P(\phi|\boldsymbol{y}, \mathcal{M}) = \mathcal{N}(\phi|\boldsymbol{y}, \boldsymbol{W}^{-1}) \tag{6.10}$$

where $\boldsymbol{B}^{-1}$ is the between-speaker covariance matrix, $\boldsymbol{W}^{-1}$ is the within-speaker covariance matrix and $\mathcal{M}$ is the model which generates (6.8). Let $\boldsymbol{S} = \{\phi_1, ..., \phi_n\}$ be a set of i-vectors of a speaker. The posterior of $\boldsymbol{y}$ given $\boldsymbol{S}$ is also Gaussian

$$P(\boldsymbol{y}|\boldsymbol{S}, \mathcal{M}) = \mathcal{N}(\boldsymbol{y}|\boldsymbol{L}^{-1}\boldsymbol{\gamma}, \boldsymbol{L}^{-1}) \tag{6.11}$$

with parameters

$$\boldsymbol{L} = \boldsymbol{B} + n\boldsymbol{W} \tag{6.12}$$

$$\boldsymbol{\gamma} = \boldsymbol{B}\boldsymbol{\mu} + \boldsymbol{W}\sum_{\phi \in \boldsymbol{S}} \phi \tag{6.13}$$

Given an enrollment i-vector ϕ_1 and a test i-vector ϕ_2 (whose roles are interchangeable), we want to verify if they are from the same speaker, i.e., we want to compute the log-likelihood ratio between the *same-speaker* hypothesis (H_s) and the *different-speaker* hypothesis (H_d)

$$\lambda = \log \frac{P(\phi_1, \phi_2|H_s)}{P(\phi_1, \phi_2|H_d)} \tag{6.14}$$

With various steps illustrated in [54], the likelihood can be obtained as

$$\begin{aligned}\lambda = \frac{1}{2}\Big(&\log|\tilde{\boldsymbol{\Gamma}}| - \boldsymbol{\gamma}_1^T\tilde{\boldsymbol{\Gamma}}\boldsymbol{\gamma}_1 + \log|\tilde{\boldsymbol{\Gamma}}| - \boldsymbol{\gamma}_2^T\tilde{\boldsymbol{\Gamma}}\boldsymbol{\gamma}_2 \\ &- \log|\boldsymbol{B}| + \boldsymbol{\mu}^T\boldsymbol{B}\boldsymbol{\mu} - \log|\tilde{\boldsymbol{\Lambda}}| + \boldsymbol{\gamma}_{1,2}^T\tilde{\boldsymbol{\Lambda}}\boldsymbol{\gamma}_{1,2}\Big)\end{aligned} \tag{6.15}$$

with

$$\tilde{\boldsymbol{\Lambda}} = (\boldsymbol{B} + 2\boldsymbol{W})^{-1} \qquad \tilde{\boldsymbol{\Gamma}} = (\boldsymbol{B} + \boldsymbol{W})^{-1} \tag{6.16}$$

$$\boldsymbol{\gamma}_{1,2} = \boldsymbol{B}\boldsymbol{\mu} + \boldsymbol{W}(\phi_1 + \phi_2) \qquad \boldsymbol{\gamma}_i = \boldsymbol{B}\boldsymbol{\mu} + \boldsymbol{W}\phi_i \tag{6.17}$$

Furthermore to make evident the role of the i-vectors, (6.15) can be rewritten as

$$\begin{aligned}s(\phi_1, \phi_2) = &\phi_1^T\boldsymbol{\Lambda}\phi_2 + \phi_2^T\boldsymbol{\Lambda}\phi_1 + \phi_1^T\boldsymbol{\Gamma}\phi_1 + \phi_2^T\boldsymbol{\Gamma}\phi_2 \\ &+ (\phi_1 + \phi_2)^T\boldsymbol{c} + k\end{aligned} \tag{6.18}$$

where

$$\boldsymbol{\Lambda} = \frac{1}{2}\boldsymbol{W}^T\tilde{\boldsymbol{\Lambda}}\boldsymbol{W} \qquad \boldsymbol{\Gamma} = \frac{1}{2}\boldsymbol{W}^T\left(\tilde{\boldsymbol{\Lambda}} - \tilde{\boldsymbol{\Gamma}}\right)\boldsymbol{W} \tag{6.19}$$

$$\boldsymbol{c} = \boldsymbol{W}^T\left(\tilde{\boldsymbol{\Lambda}} - \tilde{\boldsymbol{\Gamma}}\right)\boldsymbol{B}\boldsymbol{\mu} \qquad k = \tilde{k} + \frac{1}{2}\left((\boldsymbol{B}\boldsymbol{\mu})^T\left(\tilde{\boldsymbol{\Lambda}} - 2\tilde{\boldsymbol{\Gamma}}\right)\boldsymbol{B}\boldsymbol{\mu}\right) \tag{6.20}$$

Since the two-covariance model is a particular case of PLDA, the model parameters $\boldsymbol{B}$, $\boldsymbol{W}$ and $\boldsymbol{\mu}$ can be obtained using the EM algorithm [51].

6.3.2 Pairwise SVM

To introduce the pairwise SVM approach [55][56] we consider (6.18) and we describe how it is possible to discriminatively train the model parameters $\boldsymbol{\Lambda}$, $\boldsymbol{\Gamma}$, $\boldsymbol{c}$ and k without explicitly modelling the i-vector distributions. In particular, since (6.18) is non-linear, we need a transformation $\varphi()$ which allows to write this function as the dot-product of the model parameters and the expanded i-vectors pairs $\varphi(\phi_1, \phi_2)$.

By means of the Frobenius inner product, we can express a bilinear form as $\boldsymbol{x}^T A \boldsymbol{y} = \langle A, \boldsymbol{x}\boldsymbol{y}^T\rangle = vec(A)^T vec(\boldsymbol{x}\boldsymbol{y}^T)$, where $vec(A)$ is a column vector composed by the stacked columns of A. Therefore, (6.18) becomes

$$\begin{aligned} s\left(\phi_1, \phi_2\right) = {} & \langle \boldsymbol{\Lambda}, \phi_1\phi_2^T + \phi_2\phi_1^T\rangle + \langle \boldsymbol{\Gamma}, \phi_1\phi_1^T + \phi_2\phi_2^T\rangle \\ & + \boldsymbol{c}^T\left(\phi_1 + \phi_2\right) + k \end{aligned} \tag{6.21}$$

Then we can introduce

$$\boldsymbol{w} = \begin{bmatrix} vec(\boldsymbol{\Lambda}) \\ vec(\boldsymbol{\Gamma}) \\ \boldsymbol{c} \\ k \end{bmatrix} = \begin{bmatrix} \boldsymbol{w}_{\boldsymbol{\Lambda}} \\ \boldsymbol{w}_{\boldsymbol{\Gamma}} \\ \boldsymbol{w}_{\boldsymbol{c}} \\ \boldsymbol{w}_k \end{bmatrix} \tag{6.22}$$

and expand the i-vector pairs

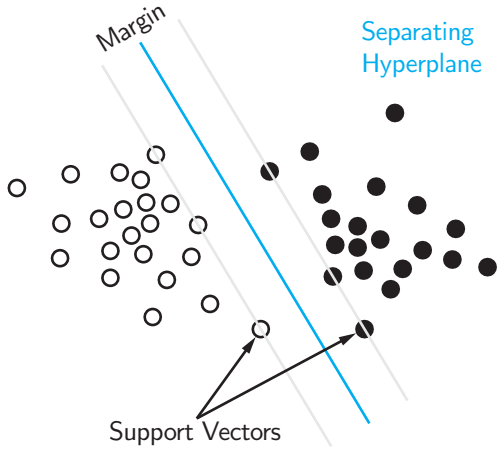
$$\varphi(\phi_1, \phi_2) = \begin{bmatrix} vec(\phi_1\phi_2^T + \phi_2\phi_1^T) \\ vec(\phi_1\phi_1^T + \phi_2\phi_2^T) \\ \phi_1 + \phi_2 \\ 1 \end{bmatrix} = \begin{bmatrix} \varphi_{\boldsymbol{\Lambda}}(\phi_1, \phi_2) \\ \varphi_{\boldsymbol{\Gamma}}(\phi_1, \phi_2) \\ \varphi_{\boldsymbol{c}}(\phi_1, \phi_2) \\ \varphi_k(\phi_1, \phi_2) \end{bmatrix} \tag{6.23}$$

Thus we can rewrite the scoring function as the following dot-product

$$\begin{aligned} s\left(\phi_1, \phi_2\right) &= S_{\boldsymbol{\Lambda}}(\phi_1, \phi_2) + S_{\boldsymbol{\Gamma}}(\phi_1, \phi_2) \\ &\quad + S_{\boldsymbol{c}}(\phi_1, \phi_2) + S_k(\phi_1, \phi_2) \\ &= \boldsymbol{w}_{\boldsymbol{\Lambda}}^T\varphi_{\boldsymbol{\Lambda}}(\phi_1, \phi_2) + \boldsymbol{w}_{\boldsymbol{\Gamma}}^T\varphi_{\boldsymbol{\Gamma}}(\phi_1, \phi_2) + \\ &\quad + \boldsymbol{w}_{\boldsymbol{c}}^T\varphi_{\boldsymbol{c}}(\phi_1, \phi_2) + \boldsymbol{w}_k^T\varphi_k(\phi_1, \phi_2) \\ &= \boldsymbol{w}^T\varphi(\phi_1, \phi_2) \end{aligned} \tag{6.24}$$

where $S_{\mathbf{\Lambda}}$, $S_{\mathbf{\Gamma}}$, $S_{\boldsymbol{c}}$ and S_k are the contribution to the final score given by the different components of $\boldsymbol{w}$.

Representing a trial with the expanded vector $\varphi(\phi_1, \phi_2)$, we can estimate $\boldsymbol{w}$ with a Support Vector Machine (SVM). This is a linear discriminative classifier, which estimates the hyperplane that best separates two given classes. In particular, the best hyperplane is the one that has the largest distance from the nearest data points of each class (Figure 6.3). Even though SVM is a linear classifier, it can perform a non-linear classification by means of the *kernel trick*, which allows computing the hyperplane without explicitly expand the features.



< Figure 6.3: The best hyperplane is the one that maximizes the margin between the classes

In our case, the two classes that we want to discriminate are *same-speaker* (target) and *different-speaker* (non-target) and the data point are pairs of i-vectors. The hyperplane can be obtained by

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \frac{1}{2}\lambda||\boldsymbol{w}||^2 + \frac{1}{n}\sum_{i=1}^{n} \max\left(0, 1 - \zeta_i \boldsymbol{w}^T \boldsymbol{x}_i\right) \quad < (6.25)$$

where n is the number of training patterns $\boldsymbol{x}_i$, which have labels $\zeta_i \in \{-1, +1\}$, λ is the regularization factor and the term $\max\left(0, 1 - \zeta_i \boldsymbol{w}^T \boldsymbol{x}_i\right)$ is the hinge loss function.

Since the training patterns are given by all possible i-vector pairs, their number can grow to the order of hundred of millions, making the training impossible. However in [55][56] a fast scoring technique has been introduces, which uses a primal solver and allows evaluating the loss function and its gradient without expanding the i-vectors, making training feasible even for large training datasets.

6.3.3 Cosine distance

A simple scoring technique that directly uses the i-vectors has been proposed in [22]. It consists in performing the cosine distance in the pairwise i-vector space. For two i-vectors ϕ_1 and ϕ_2, the cosine distance score is

$$s(\phi_1, \phi_2) = \frac{\phi_1^T \phi_2}{||\phi_1||\ ||\phi_2||} \quad < (6.26)$$

A value of c close to 1 means that the i-vectors are from the same speaker, whereas a value close to -1 means they are from different speakers. The advantage of this technique is that the adapted supervector does not need to be estimated in an enrollment step. In this case, the factor analysis is used just for feature extraction, rather than for modeling speaker and channel factors.

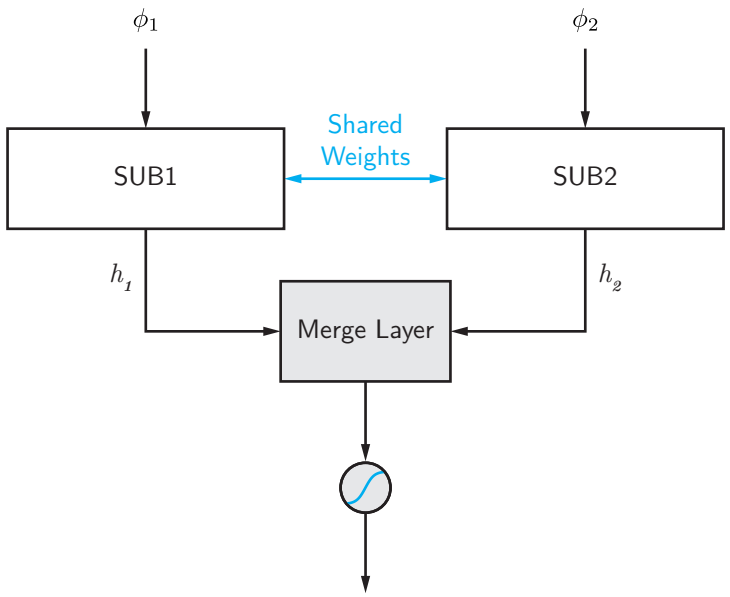
As shown in [52], the classification performance increases using normalized i-vectors. In particular, we can normalize an i-vector by dividing it by the L2-norm

$$||\phi|| = \sqrt{\sum_{i=1}^{M} \phi_i^2} \quad < (6.27)$$

6.3.4 Siamese neural network

The problem of comparing two patterns and evaluating their similarity is solved in [57] with a Siamese neural network. This architecture receives two input patterns, and returns a value that represents the similarity of the inputs. Thus, given two i-vectors as inputs, the network should learn a metric to evaluate the probability that the utterances belong to the same speaker [58][59][60]. Furthermore, during training the network can learn a representation of the i-vectors that makes more effective the computation of the distance metric. In particular, it must learn to assign a distance value close to 0 to i-vector pairs of the same speaker, and close to 1 otherwise.

Considering for example the approach of the previous Section 6.3.3, a Siamese network could learn the cosine distance of two i-vectors. Indeed, the score in (6.26) is the dot-product of two normalized i-vectors. This dot-product can be performed with this sequence of operations: an element-wise product of the i-vectors followed by the sum of the product's elements. Thus, we consider a Siamese network which has only one layer to evaluate the combination of the inputs, and we provide two normalized i-vectors as inputs. If the learnt combination was reduced to a simple sum, the metric computed by the network would be the cosine distance. If the network, instead, computed a weighted sum of the components of the product vector, it would potentially learn a better metric.



< Figure 6.4: Siamese neural network

In particular, the architecture of a Siamese neural network consists of two identical sub-networks, whose outputs are combined by a loss module (Figure 6.4). The sub-networks share the weights w. They extract the hidden representations h_1 and h_2. These representations are merged by the loss module, which computes the distance $D(h_1, h_2)$. Finally, the elements of the distance are combined together by a single node with sigmoid activation, which returns a value between 0 and 1 representing the similarity of the input i-vectors.

Different functions can be used for computing $D(h_1, h_2)$. As explained previously, we are interested in the cosine distance. Thus the operation performed in the merge layer is the element-wise product of the hidden representations

$$m_j = h_{1_j} h_{2_j} \tag{6.28}$$

where $j = 1, ..., N$ for hidden representations with size N. The obtained distance approximates the cosine distance.

The architecture of the two sub-networks can have different configurations. The basic one is composed by an input layer, having the size of the i-vector, and a layer with N nodes, having ReLU activation function. The output of the sub-network is the hidden representation, with size N. More complex sub-networks can be devised with different number of layers, with various number of nodes, and other activation functions.

Since we consider a binary classification problem, and the output node has a sigmoid activation function, the loss function that we use is binary cross-entropy [10]. The output of the sigmoid node can be interpreted as the probability that the input belongs to a certain class. Thus, the binary cross-entropy can be computed as

$$E(p) = -t\log(y) - (1-t)\log(1-y) \qquad \text{with} \quad t \in \{0,1\} \ \text{ and } \ 0 < y < 1 \tag{6.29}$$

where t is a categorical label that represents the target, and y is the output of the sigmoid node.

In order to obtain good accuracy, it's important that the input feature vectors are normalized [61]. In particular, L2-normalization of the i-vectors (6.27) is relevant for the Siamese network to approximate the cosine distance, and for making the learning process stable.

6.4 | Dataset

In a pairwise classification approach, the training samples are all the possible combinations of i-vectors in the dataset. This means that given a set of n i-vectors, the number of pairs in the training set would be n^2. A solution to this problem is to reduce this set by selecting the pairs that are most significant for training. These can be selected by considering the scores returned by a PLDA model [48]. Indeed, we can reasonably assume that the probabilities returned by the Siamese network and the PLDA scores are correlated. Thus, we can select a subset of i-vector pairs, depending on how much they are easy or difficult to classify by the PLDA, in order to create a balanced dataset.

In particular, we first score all the possible pairs with the PLDA, then we divide them in sets. Let T be the number of pairs of i-vectors from the same speaker (or true pairs). The pairs of i-vectors from different speakers (or negative pairs) can be split in two groups: one with the $M \cdot T$ pairs having the highest PLDA score, and one with all the remaining pairs. The elements in the first group are the different-speaker pairs difficult to classify by the PLDA, whereas the elements of the second group are different-speaker pairs easily detected. In our experiments, the constant M has been set to 40.

Since we use mini-batch optimization, it is important creating balanced batches. The best strategy that we have found is to randomly select the samples with the following proportions: $1/2$ same-speaker, $1/4$ difficult different-speaker and $1/4$ easy different-speaker [61]. In this way the neural network is able to learn how to discriminate accurately pairs belonging to any group: same-speaker, difficult different-speaker and easy different-speaker pairs. Furthermore, the overfitting is reduced.

6.5 | Evaluation scores

The scores produced by a classifier allow to assign labels to samples. However depending on the output of the classifier, the decision is not immediate. This is the case of a binary classifier which returns continuous values between 0 and 1. The output can be interpreted as the probability p that the input sample belongs to the positive or to the negative class. Thus a threshold θ is needed to decide if the sample is positive, i.e. $p > \theta$, or negative, i.e. $p < \theta$. It may happen that a positive sample is classified as negative and viceversa: this is a case of misclassification. To have as few misclassifications as possible is important to perform an accurate calibration, which is the process of choosing the threshold. Depending on the value of the threshold some kind of mistakes can be made more likely than others. Furthermore a type of error can be more costly in a certain application. Therefore the calibration is an essential process in a recognition system.

Given a threshold and a sample score, there are four possible classification outcomes [62]:

- True positive: the sample is positive and is classified as positive
- False positive: the sample is negative and is classified as positive
- True negative: the sample is negative and is classified as negative
- False negative: the sample is positive and is classified as negative

< Table 6.1: Confusion matrix

		True class	
		P	**N**
Predicted class	**P**	True Positive	False Positives
	N	False Negatives	True Negatives

For a good tradeoff of the classification errors, we are interested in evaluating the probability of false negatives, i.e., when a target is misclassified as impostor, and the probability of false positives, i.e., when an impostor is misclassified as target. These probabilities are called False Reject Rate (FRR) and False Accept Rate (FAR) and can be calculated as

$$FRR = \frac{\#false\ negatives}{\#total\ positive\ samples} = \frac{\#target\ speaker\ scores < \theta}{\#total\ target\ speaker\ samples} \quad < (6.30)$$

$$FAR = \frac{\#false\ positives}{\#total\ negative\ samples} = \frac{\#impostor\ scores > \theta}{\#total\ impostor\ samples} \quad < (6.31)$$

Using different values of the threshold θ, we have different tradeoffs between FAR and FRR. For example, if θ increases, also FRR increases and FAR decreases. In this case, the classifier will less likely accept an impostor,

but it will also more likely reject a true speaker.

The tradeoff point that we will consider in this work is the Equal Error Rate (EER), which is given by

$$\theta \mid FRR = FAR \tag{6.32}$$

This is a relevant value because it is shown [63] that optimizing the classifier using the EER as objective, we have an improvement of all error rates at different thresholds. Furthermore, EER can also be used for comparing different classifiers.

6.6 | Experimental results

For the experiments on the neural network, we started with the architecture proposed in [61]: a Siamese neural network with two sub-networks having an input of size 400 (like the dimension of the i-vectors), and a dense hidden layer with 2048 nodes and ReLU activation function. A dropout of 0.4 is applied during training. Each sub-network returns a representation of size 2048. Then, the element-wise product of these two vectors is computed in the merge layer. Finally, a sigmoid node produces the output score. The loss function that has been used is the binary cross-entropy, and the optimizer is RMSprop, with a learning rate of 0.001. The batches size for the gradient update have been set to 1024 samples, including 512 same-speaker pairs, 256 different-speaker pairs with high PLDA scores, and 256 different-speaker pairs with low PLDA scores. 51376 batches are processed to complete an epoch. The input samples are L2-normalized because it allows the neural network to approximate the cosine distance. With a network of this type, we obtained an EER of 4.22 %. The aim of the following experiments is to improve this result.

For the first set of experiments, we incremented the number of layers of the sub-networks to understand whether a more complex architecture could improve the hidden representations.

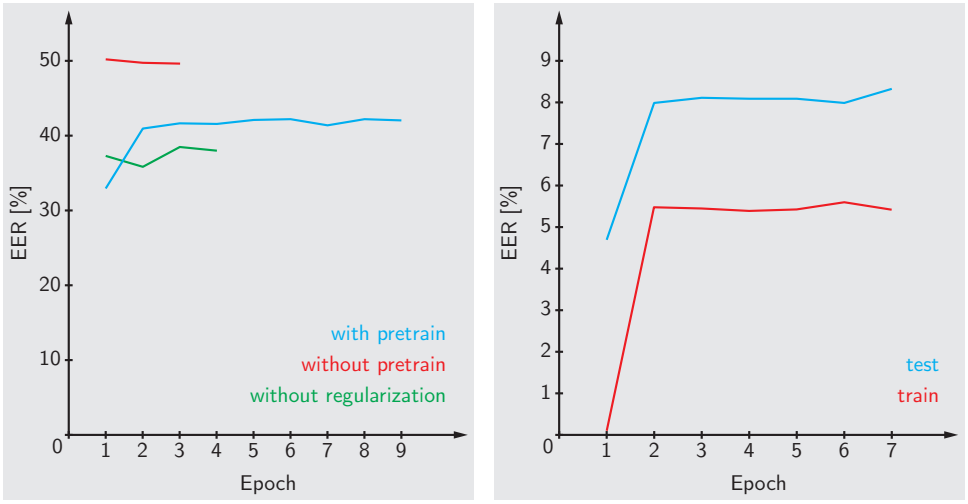
Table 6.2: Results for networks with different numbers of layers in the sub-network

Sub-network architecture	L2 regularization	Training		Test	
		EER	epoch	EER	epoch
2 layers	0	0.06%	4	35.82%	2
3 layers	0	2.21%	2	**27.99%**	1
4 layers	0	4.37%	1	30.18%	1
2 layers	1E-03	49.69%	2	49.60%	3

As Table 6.2 shows, adding layers caused overfitting, thus we applied L2 regularization, but the network was not able to learn. Therefore, we started the training without L2 regularization, and we applied the regularization after one epoch. In Table 6.3 are presented the results for networks with 2 layers in the sub-network.

L2 regularization (1st epoch)	L2 regularization (next epochs)	Activation	Dropout	Training		Test	
				EER	epoch	EER	epoch
0	1E-03	ReLU	0.4	0.12%	1	32.94%	1
0	1E-04	ReLU	0.4	0.12%	1	8.47%	8
0	1E-05	ReLU	0.4	0.10%	1	6.75%	5
0	1E-04	tanh	0.4	0.10%	1	**4.69%**	1
0	1E-04	tanh	0	0.13%	1	5.10%	1

< Table 6.3: Results for networks trained without regularization during the first epoch and with different rates of regularization for the following epochs



< Figure 6.5: A) EER for the network with 2 layers in the sub-network. Comparison for different strategies of regularization Blue: first row in Table 6.3 Red: fourth row in Table 6.2 Green: first row in Table 6.2 B) EER for the network of fourth row in Table 6.3

As we can notice from Figure 6.5 the L2 regularization applied from the first epoch onwards increase a lot the EER, but it also helps to stabilize the learning. Applying the regulation starting from the second epoch, the EER decreases, the learning is more stable with respect to learning without regularization, but the learning process gets stuck and the EER does not decrease. Changing the activation function to tanh, the EER does not grow significantly. Indeed when the argument is very positive or very negative, the tanh function tends to saturate to -1 or +1. Although we had improvements with tanh, the performance is not satisfactory if compared with the 4.22% reference EER.

In another experiment, we applied L2-normalization to the output representations of the sub-networks. In order to preserve the network weights to assume negligible values, we used tanh as activation function of the nodes.

Sub-network architecture	h1 and h2 normalization	Activation	Training		Test	
			EER	epoch	EER	epoch
1 layer	l2-norm	tanh	0.04%	9	3.84%	18

< Table 6.4: Results for a network with L2-normalized representations

As shown in Table 6.4 this normalization improved the baseline EER.

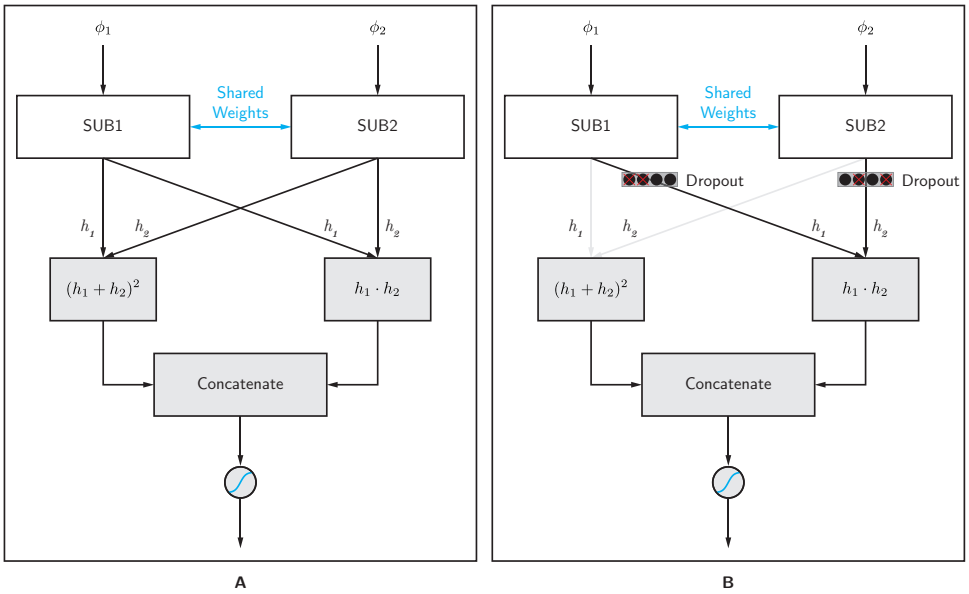
Adding a layer with 128 nodes between the merge layer and the output node improved the performance as reported in Table 6.5.

Sub-network architecture	Layer after merge layer	Activation	Training		Test	
			EER	epoch	EER	epoch
1 layer	128 nodes	ReLU	0.02%	4	3.97%	11

< Table 6.5: Results for a network with a layer of 128 nodes added before the output node

For the next set of experiments, we changed the computation performed by the merge layer. Rather than performing an element-wise product, the squared sum and the product of the sub-network representations were computed, and then concatenated before going to the sigmoid output node (Figure 6.6A), allowing the final node to evaluate more information.

However, dropout applied to this architecture was not feasible because the learning was unstable causing the training EER to be higher than the EER of the test. We solved this problem applying the dropout only to the representations used to compute the product (Figure 6.6B).



< Figure 6.6: A) Siamese network with a merge layer which concatenates the squared sum and the product of the sub-network's representations. Dropout is applied directly on the single layer of the sub-network B) Dropout is applied only on the representations that are multiplied

Dropout before product	Activation	L2 regularization	Optimizer	Training		Test	
				EER	epoch	EER	epoch
0.4	ReLU	0	RMSprop	0.02%	10	5.95%	14
0.4	tanh	0	RMSprop	0.08%	5	**3.48%**	2
0.4	tanh	1E-05	RMSprop	0.65%	7	4.63%	9
0.4	tanh	0	Adam	0.02%	17	3.61%	7

< Table 6.6: Results for networks with dropout applied only to the representations that are multiplied. Different activation functions, regularizations and optimizers are used

From the results of Table 6.6 we can notice a good improvement of the EER. Thus, we performed other tests with this merge configuration, tanh as activation function of the nodes, and no L2 regularization, obtaining an EER equal to 3.30%, the best result so far (Table 6.7).

Dropout before product	Optimizer	h1 and h2 normalization	other specs	Training		Test	
				EER	epoch	EER	epoch
0.4	RMSprop	l2-norm	none	0.00%	13	**3.30%**	5
0.4	Adam	l2-norm	none	0.01%	13	3.43%	6
0.4	RMSprop	l2-norm	layer with 128 nodes after merge	0.03%	5	3.91%	1
0.4	RMSprop	l2-norm	sum of squares	0.02%	9	3.66%	7

< Table 6.7: Results for networks with L2-normalization applied to the sub-network's representations

Table 6.8 presents the results that have been obtained by changing the architecture of the sub-network while maintaining this merging technique. In Table 6.9, instead, we show the results obtained adding layers of different size after the merge.

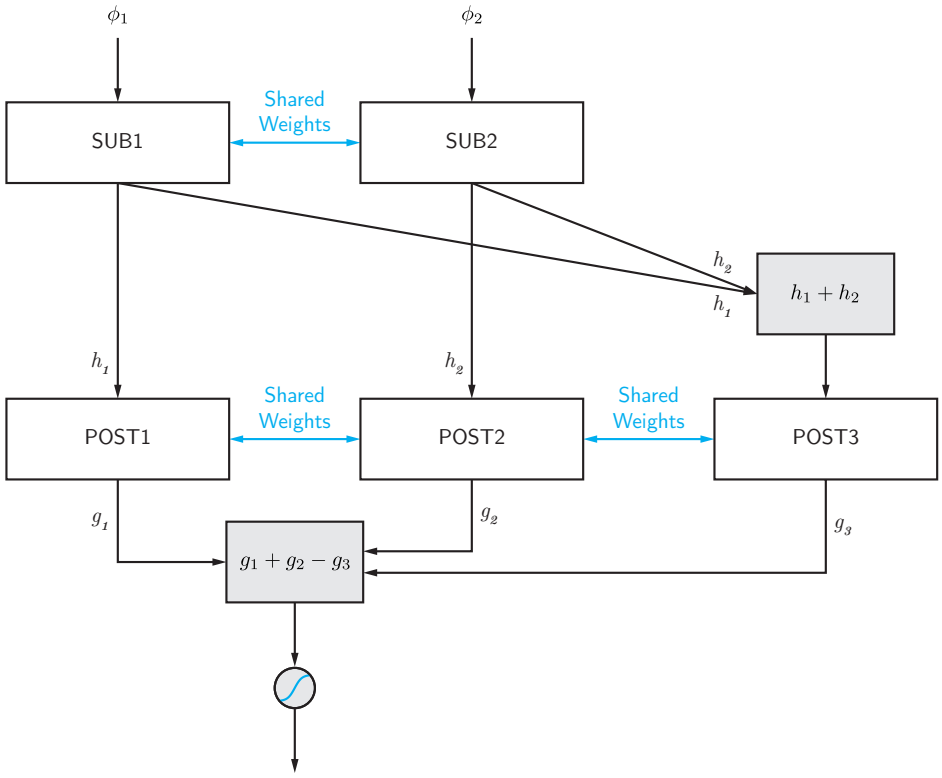
Sub-network architecture	Training		Test	
	EER	epoch	EER	epoch
1 layer with 512 nodes	0.00%	24	3.45%	3
1 layer with 1024 nodes	0.00%	17	3.35%	6
1 layer with 4096 nodes	0.02%	5	**3.30%**	6
2 layers with 512 nodes	0.02%	15	5.36%	2
2 layers with 1024 nodes	0.01%	15	4.37%	7
3 layers with 1024 nodes	0.03%	5	5.70%	7

< Table 6.8: Results for networks with different sub-network architectures

Layer after merge layer	Training		Test	
	EER	epoch	EER	epoch
1024 nodes	0.03%	5	3.73%	1
512 nodes	0.03%	5	3.81%	1
256 nodes	0.03%	8	3.76%	1
64 nodes	0.03%	5	3.89%	1
32 nodes	0.03%	5	**3.63%**	1

< Table 6.9: Results for networks with a layer positioned after the merge layer

Since the results were similar to the previous ones, we did not continue experimenting on this line. We built another architecture adding a different sub-network to process the representations of the first sub-network (Figure 6.7). We will refer to this architecture as *post-network*. In particular, we use three post-networks which share their weights. The representation h_1 is the input of the first post-network, the representation h_2 is the input of the second post-network and the sum of both representations is the input of the third post-network. Then we compute the sum and difference of the output representations of the post-networks. Our aim is to avoid overfitting. Indeed, without the third post-network, which uses $h_1 + h_2$ as input, the post-network could learn h_1 and h_2 focused on the training data. In Table 6.10 we present the results for these experiments. The configuration of the sub-network is: a single layer with 2048 nodes, tanh activation function, and 0.4 dropout rate.



< Figure 6.7: Neural network with two different sub-networks which share the weights: SUB and POST. The representations returned by the POST sub-networks are merged with sum and difference

Post-network architecture	Post-network dropout	Training		Test	
		EER	epoch	EER	epoch
1 layer with 2048 nodes	0.4	0.06%	16	13.21%	22
1 layer with 1024 nodes	0.4	0.55%	6	15.19%	6
1 layer with 512 nodes	0.4	1.70%	9	14.99%	9
2 layers with 1024 nodes	0.4	1.01%	7	8.62%	6
2 layers with 512 nodes	0.4	1.60%	7	10.64%	5
3 layers with 1024 nodes	0.4	0.91%	5	7.16%	5
3 layers with 512 nodes	0.4	3.02%	6	9.22%	3
4 layers with 1024 nodes	0.4	5.38%	2	9.63%	2
4 layers with 512 nodes	0.4	8.36%	13	12.28%	3
2 layers with 512 nodes	0.2	0.26%	11	6.18%	7
3 layers with 1024 nodes	0.2	0.09%	6	**5.46%**	10

< Table 6.10: Results for networks with different post-network architectures

From Table 6.10 we can notice that we improved the EER using more layers, but we did not reach a result better than 3.30%.

Finally, we removed the top sub-network, and we used as inputs of the post-networks the i-vectors. For example, the input of the first post-network is i-vector ϕ_1, rather than its representation h_1. However, as shown in Table 6.11, the results were disappointing.

Post-network architecture	Post-network dropout	Training		Test	
		EER	epoch	EER	epoch
3 layers with 1024 nodes	0	0.24%	5	17.07%	1
3 layers with 512 nodes	0	0.15%	9	12.36%	1
3 layers with 1024 nodes	0.2	0.03%	21	15.27%	13
3 layers with 1024 nodes	0.4	0.05%	39	**9.01%**	31

< Table 6.11: Results for networks with different post-network architectures and dropout rates

In conclusion, the best network obtained has a sub-network with 1 layer of 2048 nodes and tanh activation function. The merge is done by concatenating the squared sum and the product of the sub-network representations. A dropout of 0.4 is applied only on the representations used to compute the product. Furthermore, L2-normalization is performed on the representations before the merge layer. The optimizer is RMSprop and the activation of the output node is the sigmoid function. Using this architecture we obtained an EER equal to 3.30%. Comparing this result with the one reached by other techniques, the proposed architecture improve the cosine distance, and the previous Siamese network, but it does not improve the performance of the state-of-the-art PLDA and PSVM.

Technique	EER
Cosine distance	5.30%
Previous Siamese network	4.22%
Actual Siamese network	3.30%
PLDA	3.22%
PSVM	3.01%

< Table 6.12: Comparison with EER obtained with state-of-the-art techniques and the Siamese network proposed in this work

Chapter 7
Conclusion

In this thesis, we presented an overview of the language and speaker recognition systems. In particular, we were interested in understanding the entire process for having more awareness about the characteristics of the input vectors that are provided to the classifier. Since the main objective was to achieve improvements for neural network classifiers, we also analyzed the working principles of neural networks, their training algorithms, and different techniques for their regularization.

As far as language recognition is concerned, we proposed a neural network classifier as part of a system submitted to the NIST 2017 Language Recognition Evaluation. We started from a basic architecture, and we experimented many different architectures on both the original and augmented versions of the dataset. Since we had to deal with overfitting problems, even with simple and small architectures, we applied different methods of regularization. The best results were obtained with pre-processed data, leading to an 18.95% equal error rate. One of the most important sources of errors that we observed was the mismatch between the training and test data. Indeed, splitting the train dataset in two parts, and performing training and testing with these two subsets, we obtained an equal error rate of 5.7%. This was confirmed by performing the same experiment on the test dataset, for which we obtained an equal error rate of 14.9% using the same network architecture. These characteristics of the data caused the network to not perform well on the test set.

In the context of speaker recognition, we presented the Siamese neural network architecture to perform binary classification of i-vector pairs. The aim was to reach an error rate closer to the one achieved by state-of-the-art PLDA and PSVM techniques. We experimented different sub-network configurations, types of regularization, and activation functions. We proposed a merging technique that consisted in computing the concatenation of squared sum and product of the sub-network representations (similar to Euclidean distance). Although a good EER of 3.30% has been obtained, the performance was not better than PLDA. Adding a second sub-network and by merging the representations by means of sum and difference did not provide better results.

Since in the current system, feature extraction, i-vector modelling, and classification are independent steps, future work could be devoted to a system that is trained to joint optimize these components.

References

[1] S. Davis, and P. Mermelstein, "Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences" *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 28, pp. 357–366 (1980).

[2] T. Kinnunen, and H. Li, "An overview of text-independent speaker recognition: From features to supervectors" *Speech communication*, vol. 52, no. 1, pp. 12–40 (2010).

[3] X. Huang, A. Acero, and H.-W. Hon, "Spoken language processing: A guide to theory, algorithm, and system development" *Upper Saddle River, NJ: Prentice Hall PTR* (2001).

[4] L. Rabiner, and B. H. Juang, "Fundamentals of Speech Recognition" *Upper Saddle River, NJ, USA: Prentice-Hall, Inc.* (1993).

[5] J. Ramirez, J. Segura, C. Benítez, A. de la Torre, and A. Rubio, "Efficient voice activity detection algorithms using long-term speech information" *Speech Comm.* 42 (3–4), 271–287 (2004).

[6] T. Kinnunen, and P. Rajan, "A practical, self-adaptive voice activity detector for speaker verification with noisy telephone and microphone data" *IEEE International Conference on Acoustics, Speech and Signal Processing*, Vancouver, pp. 7229-7233 (2013).

[7] L. R. Rabiner, and R. W. Schafer, "Introduction to Digital Speech Processing" *Foundations and Trends in Signal Processing*, vol 1, no 1–2, pp 1–194 (2007).

[8] P. A. Torres-Carrasquillo, D. A. Reynolds, E. S. M. A. Kohler, R. J. Greene, and J. J. R. Deller, "Approaches to language identification using Gaussian Mixture Models and shifted delta cepstral features" *Proceedings of ICSLP 2002*, pp. 89–92 (2002).

[9] W. Campbell, P. A. Torres-Carrasquillo, and D. Reynolds, "Language recognition with support vector machines" *Proceedings of Odyssey: The Speaker and Language Recognition Workshop*, pp. 41–44, ISCA (2004).

[10] C. M. Bishop, "Pattern Recognition and Machine Learning" *Springer* (2007).

[11] D. Reynolds and R. Rose, "Robust text-independent speaker identification using gaussian mixture speaker models" *Speech and Audio Processing, IEEE Transactions on*, vol. 3, pp. 72–83 (1995).

[12] I. Goodfellow, Y. Bengio and A. Courville, "Deep Learning" *MIT Press* (2016).

[13] S. Cumani, "Speaker and Language Recognition Techniques", *PhD thesis, Politecnico di Torino* (2012).
[14] D. A. Reynolds, T. F. Quatieri, and R. B. Dunn, "Speaker verification using adapted gaussian mixture models," *Digital Signal Processing*, vol. 10, pp. 19–41 (2000).
[15] J. L. Gauvain, and C. H. Lee, "Maximum a posteriori estimation for multivariate gaussian mixture observations of markov chains," *Speech and Audio Processing, IEEE Transactions on*, vol. 2, pp. 291–298 (1994).
[16] P. Kenny, G. Boulianne, P. Ouellet, and P. Dumouchel, "Speaker and session variability in gmm-based speaker verification" *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 15, pp. 1448–1460 (2007).
[17] P. Kenny, "Joint factor analysis of speaker and session variability: Theory and algorithms" *Tech. Rep. CRIM-06/08-13, CRIM* (2005).
[18] P. Kenny, G. Boulianne, and P. Dumouchel, "Eigenvoice modeling with sparse training data," *IEEE Transactions on Speech and Audio Processing*, vol. 13, no. 3, pp. 345–354 (2005).
[19] P. Kenny, M. Mihoubi, and P. Dumouchel, "New map estimators for speaker recognition" *Proceedings of EUROSPEECH 2003*, pp. 2964–2967 (2003).
[20] P. Kenny, G. Boulianne, P. Ouellet, and P. Dumouchel, "Joint factor analysis versus eigenchannels in speaker recognition," *IEEE Trans. Audio, Speech, Lang. Process*, pp. 1435–1447 (2007).
[21] N. Dehak, "Discriminative and generative approches for long- and short-term speaker characteristics modeling: Application to speaker verification," *Ph.D. dissertation, École de Technologie Supérieure, Montreal*, QC, Canada (2009).
[22] N. Dehak, P. Kenny, R. Dehak, P. Dumouchel, and P. Ouellet, "Front–end factor analysis for speaker verification," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 19, no. 4, pp. 788–798 (2011).
[23] F. Chollet, "Deep Learning with Python" *Manning Publications Co.* (2018).
[24] M. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, and G. Hinton, "On rectified linear units for speech processing" *38th International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Vancouver (2013).
[25] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models" *International Conference on Machine Learning (ICML)* (2013).
[26] D. Jurafsky, and J. H. Martin, "Speech and Language Processing" *Englewood Cliffs, NJ, USA: Prentice-Hall* (2000).
[27] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Neurocomputing: Foundations of research," *ch. Learning Representations by Back-propagating Errors*, pp. 696–699, Cambridge, MA, USA: MIT Press (1988).
[28] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W.

Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition" *Neural Computation 1(4)*, 541–551 (1989).
[29] Y. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop" *In Neural networks: Tricks of the trade*, pp. 9–48. Springer (2012).
[30] S. Ruder, "An overview of gradient descent optimization algorithms" *arXiv:1609.04747* (2016).
[31] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning" *Proceedings of the 30th International Conference on Machine Learning (ICML-13)* (S. Dasgupta and D. Mcallester, eds.), vol. 28, pp. 1139–1147, JMLR Workshop and Conference Proceedings (2013).
[32] T. Tieleman, and G. Hinton, "Lecture 6.5 — RmsProp: Divide the gradient by a running average of its recent magnitude" *Coursera: Neural Networks for Machine Learning* (2012).
[33] D. P. Kingma, and J. Ba, "Adam: a Method for Stochastic Optimization" *International Conference on Learning Representations*, pp. 1–13 (2015).
[34] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958 (2014).
[35] S. Wager, S. Wang, and P. S. Liang, "Dropout training as adaptive regularization" *Advances in Neural Information Processing Systems 26* (C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, eds.), pp. 351–359, Curran Associates, Inc. (2013).
[36] D. Yu, and L. Deng, "Automatic speech recognition – A deep learning approach" *Springer* (2015).
[37] S. Wang, and C. Manning, "Fast dropout training" *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pp. 118–126 (2013).
[38] C. M. Bishop, "Regularization and complexity control in feed-forward networks" *Proceedings International Conference on Artificial Neural Networks ICANN'95*, vol. 1, pp. 141–148 (1995).
[39] C. M. Bishop, "Training with noise is equivalent to Tikhonov regularization" *Neural Computation*, 7(1), 108–116 (1995).
[40] J. Zhao, H. Shu, L. Zhang, X. Wang, Q. Gong, and P. Li, ''Cortical competition during language discrimination,'' *NeuroImage*, vol. 43, pp. 624–633 (2008).
[41] H. Li, B. Ma, and K. A. Lee, "Spoken language recognition: from fundamentals to practice" *Proceedings of the IEEE*, vol. 101, no. 5, pp. 1136-1159 (2013).
[42] G. Saon, and J.-T. Chien, "Large-Vocabulary Continuous Speech Recognition Systems: A Look at Some Recent Advances" *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 18-33 (2012).
[43] NIST 2017 Language Recognition Evaluation Plan.
[44] P. Matějka, O. Plchot, O. Novotný, S. Cumani, A. Lozano-Diez, J. Slavicek, M. Diez, F. Grézl, O. Glembek, K. V. Mounika, A. Silnova,

L. Burget, L. Ondel, S. Kesiraju, and J. Rohdin, "BUT-PT System description for NIST LRE 2017" BUT Speech Processing Group (2017).
[45] P. Matějka, L. Zhang, T. Ng, S. H. Mallidi, O. Glembek, J. Ma, and B. Zhang, "Neural network bottleneck features for language identification" *Proceedings of the IEEE Odyssey: The Speaker and Language Recognition Workshop*, Joensu, Finland (2014).
[46] R. Fér, P. Matějka, F. Grézl, O. Plchot, K. Veselý, and J. H. Černocký, "Multilingually trained bottleneck features in spoken language recognition" *Computer Speech & Language*, vol. 46, no. Supplement C, pp. 252 – 267 (2017).
[47] K. S. Rao, and S. Sarkar, "Robust Speaker Recognition in Noisy Environments" *Springer* (2014).
[48] S. Cumani, and P. Laface, "Large scale training of pairwise support vector machines for speaker recognition," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 22, no. 11, pp. 1590–1600 (2014).
[49] W. M. Campbell, D. E. Sturim, and D. A. Reynolds, "Support vector machines using GMM supervectors for speaker verification" *IEEE Signal Processing Letters*, vol. 13, pp. 308–311 (2006).
[50] S. J. D. Prince, and J. H. Elder, "Probabilistic linear discriminant analysis for inferences about identity" *11th International Conference on Computer Vision*, pp. 1–8 (2007).
[51] P. Kenny, "Bayesian speaker verification with heavy–tailed priors" *Keynote presentation, Odyssey 2010, The Speaker and Language Recognition Workshop* (2010). Available at http://www.crim.ca/perso/patrick.kenny/kenny_Odyssey2010.pdf
[52] D. Garcia-Romero, and C. Y. Espy-Wilson, "Analysis of i–vector length normalization in speaker recognition systems" Proc. Of Interspeech 2011, pp. 249–252 (2011).
[53] N. Brümmer and E. de Villiers, "The speaker partitioning problem" *Proc. Odyssey 2010*, pp. 194–201 (2010).
[54] S. Cumani, N. Brümmer, L. Burget, P. Laface, O. Plchot, and V. Vasilakakis, "Pairwise discriminative speaker verification in the i-vector space" *IEEE transactions on audio, speech, and language processing*, vol. 21 n. 6, pp. 1217-1227 (2013).
[55] S. Cumani, N. Brümmer, L. Burget, and P. Laface, "Fast discriminative speaker verification in the i–vector space" *Proceedings of ICASSP 2011* (2011).
[56] L. Burget, O. Plchot, S. Cumani, O. G. P. Matějka, and N. Brümmer, "Discriminatively trained Probabilistic Linear Discriminant Analysis for speaker verification" *Proceedings of ICASSP 2011* (2011).
[57] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature Verification using a "Siamese" Time Delay Neural Network" *Proceedings of the 6th International Conference on Neural Information Processing Systems*, pp. 737-744 (1994).
[58] S. Chopra, R. Hadsell, and Y. LeCun, "Learning a similarity metric

discriminatively, with application to face verification" *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 539–546, IEEE Computer Society (2005).
[59] R. Hadsell, S. Chopra, and Y. Lecun, "Dimensionality reduction by learning an invariant mapping" *Proc. Computer Vision and Pattern Recognition Conference*, IEEE Press (2006).
[60] G. Koch, R. Zemel, and R. Salakhutdinov, "Siamese neural networks for one–shot image recognition" (2015).
[61] F. Tuveri, "Factor analysis and neural networks for speaker recognition" *Thesis, Politecnico di Torino* (2016).
[62] T. Fawcett, "An introduction to roc analysis" *Pattern Recogn. Lett.*, vol. 27, pp. 861–874 (2006).
[63] N. Brümmer, "Measuring, refining and calibrating speaker and language information extracted from speech" *PhD thesis, University of Stellenbosch* (2010).