Understanding and Improving Deep Learning Source Separation Algorithms

Grau Noël, Joan

Curs 2017-2018

Director: PRITISH CHANDNA

GRAU EN ENGINYERIA EN SISTEMES AUDIOVISUALS

Universitat Pompeu Fabra Barcelona
Escola Superior Politècnica

Treball de Fi de Grau

Understanding and Improving Deep Learning Source Separation Algorithms

Grau Noël, Joan

TREBALL FI DE GRAU

ENGINYERIA EN SISTEMES AUDIOVISUALS

ESCOLA SUPERIOR POLITÈCNICA UPF

CURS 2017-2018

PRITISH CHANDNA

Contents

Acknowledgments

First and foremost I would like to thank my thesis tutor, Pritish Chandna, for his continuous assistance and dedication throughout the making of this thesis.

Furthermore, I also would like to thanks to my family and friends for their continuous support and encouragement.

List of Tables

List of Figures

ABSTRACT

In this thesis we are going to explore source separation and it's meaning, from both a theoretical and practical point of view, since it is one of the most discussed topics in signal processing. As well, a increasing interest in deep learning has overtaken this discussion and changed the classical points of view on source separation and opened the door into better and more reliable methods. We will start by explaining this set of historical methods typically used to solve this problem, with various levels of success, some which have acted the baseline for new algorithms and applications. We are going so explain how deep learning is related with the current BSS (Blind Source Separation) problems, and we will end up with the explanation of one of these state-of-the-art algorithms proposed to solve source separation problems, with quite success.

The algorithm we will study performs a low-latency monaural audio separation, and we will later take this model and propose various improvements: we will start by porting this state-of-the-art algorithm to a new framework, we will then upgrade it's architecture to be able to fully work with stereophonic audio signals, which are the standard for this days in musical mixtures; finally we will also propose data augmentation techniques and a denoising deep-learning framework to further improve the model result.

ABSTRACTE

En aquesta tesis explorarem què és el problema de la separació de senyals, una dels problemes típics del processament de senyals, que ens els últims anys també ha sigut tema de discussió a d'altres àrees com l'aprenentatge profund. Començarem explicant quins han sigut els mètodes emprats històricament per solucionar aquest problema, alguns dels quals han tingut més repercussió que d'altres, i que han actuat com a la base en la qual mètodes i algoritmes més moderns s'han aplicat per separar senyals. També explicarem com l'aprenantatge profund es relaciona amb la separació cega de senyals (BSS - Blind Source Separation en anglès), i finalitzarem aquesta explicació centrant-nos amb un model en l'estat de l'art proposat per resoldre aquest problema, i que ha obtingut bons resultats.

L'algoritme en questió es centra a realitzar una separació monaural a baixa latència, i agafarem aquest mateix model per intentar de millorar-lo proposant una sèrie de millores: començarem per actualizar el marc tecnològic originalment utilitzar per construïr aquest model, millorarem després la seva arquitectura per tal de que treballi amb senyals estereofòniques, les quals són la norma actualment; seguidament proposarem tècniques d'augmentació de dades, i finalitzarem proposant un model d'aprenantatge profund que apliqui reducció de soroll a les sortides de l'algoritme, per tal de millorar-ne els resultats.

1 INTRODUCTION

Source separation has been one of the classic problems in audio signal processing amongst the years. Although this problem has been extended through other disciplines, such as image processing, digital communications, medical imagining, among many others, with similar goals, in this thesis we will focus on the ones related to audio processing. More specifically this thesis will focus on the applications related to the separation of sources present in musical mixtures, first starting by doing a chronological explanation of the historical algorithms used to solve this problem. Special emphasis on will be made on the current state of the art algorithms, which as many other disciplines in this recent years has taken deep learning as it's default method of research and evolution.

Even though separating signals is a simple and natural task for a human, i.e. being able to differentiate and recognize different instruments in a musical mixture, or for example recognize different voice pitches and relate them to singular sources, etcetera. This, which is not by any means an easy task to replicate technologically, has been a subject of interest historically. It was first introduced and formulated with the *cocktail party problem* [7]. As explored by this theory, despite the problem being a very easy task to resolve by humans, modeling and solving it has proven to be a very complex task for a machine [14]. The first solution for the problem is to try to replicate the process of humans (solving the cocktail problem) directly by a machine step by step, from the perceptual task of receiving the mixture signal to decomposing it's sources and finally recognizing each one by relating it to a physical source; of course, this would imply that this is a pure neurological task to resolve, and even though this area has made great advancements at studying the way our brain is shaped, there is still discussion on about how brain processes this information in order to extract a solution. Of course, this is not a task it is not going to be addressed in this thesis, but it is going to work as an starting point for the modeling of the problem. In the current state of the technology, the only possible solutions for this problem is the simplification of the problem by creating simpler, smaller models based on what the human brain does [37]. With the creation and rising popularity of neural networks and more specifically it's deep learning variations, whose purpose is to attempt to replicate the way a human brain behaves and performs tasks starting by it's smallest component, the neuron, multiple variants of this theory have been spawned, such as networks which try to replicate the visual cortex system [16][21], usually known as CNNs (Convolutional Neural Networks), whose applications are not limited to image recognition or processing alone [19], since they have been proven to be a solid method for training neural networks across many applications [1] and disciplines, like in the case we will describe in this thesis. We will describe in this thesis, as state-of-the-art, one algorithm [5] which uses an evolution of the CNNs, namely an autoencoder, and it's the one which will be subject to the improvements and analysis of it's yielded results.

Finally even though this thesis is not going to enter in any of the following points, it is important to express the possible implications source separation could have in audio analysis and applications. Even though the applications could be endless, the most classic ones which are now the focus point of BSS are usually aiming at demixing tracks for audio engineering or remastering, speech enhancement, automatic speech recognition, pitch or beat tracking, real-time audio processing, among many others; but in a more general manner, the most interesting outcomes would be the ones related to the information gain it could be extracted from existing audio signals and it's possible applications, such as musical/speech analysis and synthesis.

1.1 Source separation problem

Source separation problem can be introduced as as the combination of different factors or environmental variables that shape the way in how the whole cocktail-party-inspired model in constructed and understood. First of all, the terminology that describes how a separation problem behaves must be defined: first of all Source separation models can be divided in two big categories when taking into account which is the prior information we have about the initial problem setup:

- Blind Source Separation: refers to the model where little to no information about the original signal or the original sources is known, and the only information known is the mixture signal. This is the most common case in a real world scenario, since usually the only information known in advance it's just the mixed signal; for example, if we have a mixture composed by two different sources, one being a real world instrument and the other being pure white noise, a human can, differentiate and mentally separate this two sources, even though one it's just a random noise that he previously couldn't know; the same could be said if two instruments that one person didn't know would appear on the same mixture: a human is still able to differentiate them and perceive it as two different sources, even though it can't relate them to a previously known physical instrument; this is what we are trying to model. This also implies that this is an under-determined problem, which means that there are fewer equations (the mixed signal) than unknowns (the original sources), and thus further constraints need to be imposed in order to get to a correct solution. There is also some cases where actually *some* information about the original mixture it is given, but it's not enough to consider it an ISS (Informed Source Separation), which would be the state-of-the-art case we will treat later on, where we actually have information about the number of sources in the mixture, and will take advantage of it to create the model. That being the case, we will refer our problem from now on as a BSS problem.

- Informed Source Separation: These are the cases where usually some musical information about the signal treated is given. Some methods use for example score-informed source separation for music [26][12], spatial information [30] or mixing information [31]; all of this extra information being constraints added to further simplify the model. Of course, this models usually yield better results than BSS modeled ones, but with the expense of needing a previous and specific work to generate this extra information, which usually it isn't possible to get in a real-world scenario. As this is not the main objective of the thesis it won't go into further detail.

It can also taken into account more specific constraints for the model, even though the following are usually assumed at the generation of the model, they are worth mentioning.

- Music vs speech signals: even though they share the low level structure like the periodicity, harmonic partials, transitivity, and wideband noise, there are higher level descriptors to consider between them, such as the phonemes on voice vs the musical structure of instruments (chords), the dependence between sources and the fundamental frequencies, pitch, spectral envelope, etcetera. [46]

- Underdetermined vs overdetermined: usually refers to the way the mixture was recorded channel-wise; for example a recording made with more microphones than sources is an overdetermined mixture, but more often we have recordings with just two channels but more than two sources, therefore making it an underdetermined model.

- Instantaneous vs convolutive: this refers to the reverb present in the mixture, which is usually the hardest factor to treat in a mixture when performing source separation. In the dataset that will be later described and tested, all of the mixtures are professionally mixed in studio, so we assume they are instantaneous.

- Time-varying vs static: this is a niche case, where a time-varying signal exists if either the source or the receiver (for example a microphone) are moving during a recording. We assume staticity in our recordings.

A good way to picture the last three points are imagining the following two cases: a live recording is usually over-determined (various microphones, one or more for source), convolutive (high presence of reverb) and possibly time-varying (the performer may be moving); against the second case, where we have a studio recording with an usually underdetermined signal (just one microphone), very little reverb (instantaneous) and static sources. [46]

Having introduced most of the theoretical elements to consider for a new audio mixture model, now a more mathematical definition will be formulated. On an ideal world, a mixture signal could be interpreted with the following equation:

$$x_i(t) = \sum_{k=1}^{K} s_{ik}(t)$$

Where $x_i(t)$ is the mixture signal, with $i = 1, 2, ...N$ being N the number of channels in the mixture, can be decomposed as the sum of signals $s_{ik}(t)$ where $k = 1, 2, ...K$ are the number of sources on each channel i , with the final amplitude of the summation of $s_{ik}(t)$ being the total amplitude of $x_i(t)$. Of course, this would mean that the final mixture has the sum of the *ideal* signals from the sources, but in the most common scenarios this ideal case is rarely given. A way to model this, is apply a time-invariant filter to each source, modeling what it would be the frequency masks for each source and unit of time. The result would be the following:

$$x_i(t) = \sum_{k=1}^{K} s_{ik}(t) * h_{ik}(t)$$

With $h_{ik}(t)$ being a LTI (Linear and time-invariant) filter for each source. Then, by the following convolutional properties, $f * g = \int_{-\infty}^{\infty} f(\tau) \cdot g(t - \tau)\delta\tau$, we obtain the following formula, still in the time domain of the signal:

$$x_i(t) = \sum_{k=1}^{K} \sum_{\tau=-\infty}^{\infty} s_{ik}(\tau) \cdot h_{ik}(t - \tau)$$

It can be seen where the difficulty of the problem lays: it is needed to recover from an audio signal, what is for each bin of time the contribution from each source, plus also needing for each source, what is the filtering mask that modifies it's contribution at the same time. This, with just the information that is given, seems to be a very difficult or close to impossible task, unless we somehow can get to know how each source contributes to the final signal by looking at the signal patterns for example, or by some other method.

Of course, there should be an easier way to deal with this problem, and since we are working with audio signals, it is possible to shift any signal from the time domain to the frequency domain by using the Fourier Transform, therefore, with the previous equation, and knowing that multiplication in time domain equals to convolution in frequency domain and viceversa, the following transformation can be formulated:

$$X_i(f,t) = \sum_{k=1}^{K} \sum_{\tau=-\infty}^{\infty} S_{ik}(f,\tau) * H_{ik}(f,t-\tau)$$

And with the previous property we talked before, we can simplify this equation back for each signal, for each unit of time t:

$$X_i(f,t) = \sum_{k=1}^{K} S_{ik}(f,t) \cdot H_{ik}(f,t)$$

One advantage that this transformation gives, is that now, apart from having much more information about the signal and it's frequencies, is that we can take them apart depending on what is the contribution from each source, therefore we can filter the sources from the audio mixture if we know the mask for each source for each bin of time. This is called time-frequency masking. To do so, we need to transform to the time-frequency domain, and it can be done by performing the short time Fourier transform (STFT).

Finally, the masks applied to each signal must be modeled. Some of the methods for the mask filtering are the following:

- Mask binarization:

Having the STFT of the original mixture $X_i(n,f)$, and the mask for each source $S_{i,k}(n,f)$

$$S_{ik}(n,f) = \begin{cases} X_i(n,f) \;\; if \;\; |S_{ik}(n,f)| = max_k \, |S_{ik}(n,f)| \\ 0 \qquad\qquad\qquad\qquad otherwise \end{cases}$$

Which means that for each mask $S_{i,k}(n,f)$, the original value of the mixture mask $X_i(n,f)$ is kept if the source $S_{i,k}(n,f)$ is the one contributing the most on that bin, therefore no other source will have any value at that frequency and time bin. Since this is an abrupt modeling, a solution to that is to use an ideal binary mask for each source, where each bin has the original value of magnitude or zero, but independently for each source, so they can share bins on the same frequencies if two sources contribution in that bin go over a threshold. Still, it's a pretty extreme solution although it's the easiest one to implement.

- Wiener filtering:

Of course, the previous mask can be tweaked and improved, because if we think about it, it's very normal that in a mixture, more than one signals might share frequencies (wave superposition principle, plus taking into account that we are working with harmonics, so superpositions may be more frequent) . Weiner filtering fixes this problem by instead giving a maximum value or zero to each source mask on each time and frequency bin, it gives a weight ranging from 0 to 1 multiplied by the mixture mask $X_i(n, f)$. The formula is the following:

$$S_{ik}(n,f) = \frac{|S_{ik}(n,f)|^2}{\sum_{k=1}^{K} |S_{ik}(n,f)|^2} X_{ik}(n,f)$$

Therefore, what we need to do is find each mask for each source, and apply it to the mixture STFT to get the contribution of each source. Even though the problem remains underdetermined, it opens the door to a fair share of methods that can get to resolve it.

1.2 Objectives

The objective can be divided in two blocks: the first one consists in naming and briefly explaining the methods and algorithms, with the necessary accuracy relative to its importance, that have evolved up until the current state-of-the-art algorithms, or on the contrary are still being used. Other than that, the basis for the current state-of-the-art algorithms will also be needed to be formulated, as they are based in deep learning, it is needed to understand the basis for neural networks and it's variations. The second part is a more technical part where we will analyze the current state-of-the-art algorithm, for which it has been chosen DeepConvSep [6][4], which is working with the old DSD100 dataset. The main objective is to recreate this algorithm in a new deep learning framework (we have chosen Pytorch), try a new dataset (we have chosen musDB [35]), and try to implement various improvements on the original algorithm architecture.

2 HISTORICAL METHODS

Across the years many methods have been studied and modeled with the main objective of resolving the source separation problem. These methods have been spawned independently across the years by researchers in different disciplines, such as audio signal processing, stochastic signal processing, physics or cognitive psychology. [46] This is due the fact each of those algorithms focuses on resolving a distinct variation problem, by creating different model constraints that allow a simpler modeling of the problem. Using this fact gives advantages and disadvantages, the later being for example low flexibility, as some methods only work in specific scenarios, or implementation issues, since some methods and theories are based in ideal cases.

2.1 Principal Component Analysis

PCA is a statistical method mostly used to treat data by reducing it's dimensionality based on the correlation between the variables within any set of variables in a dataset, with the objective of finding an axis of projection for the data that retains as much as possible original information of the original data but in a much smaller dimensionality, by using the variability of the data as a tool to find this new data space. This is achieved by performing a transformation with the original set of variables, to a new ones called principal components (PCS), which is usually made with a linear decomposition method such as Singular-Value Decomposition (SVD).

The final goal is to obtain a linear transformation of the original data that retains the variance and weight for each component, so having an original matrix X with dimensionality $n\ x\ p$, being n the number of samples and p being the number variables, we can find the covariance matrix for each variable to the rest of the dataset by using:

$$C = \frac{X \cdot X^T}{(n-1)}$$

This matrix can be diagonalized giving a dimensionality of $p\ x\ p$. Such a result is mostly gotten by a method such as SVD (as a matrix decomposition method). What SVD does is decompose this matrix in two sets of matrices, one containing eigenvectors, which are the principal directions of the data, and related to each vector a value (called eigenvalue) that represents the weight of each eigenvector to the data. Since this matrices are built from the covariance matrix, this is a representation of the directions of covariance from the original data for each variable, with the weight for each one on the original dataset. This allows for reduction of the number of number of components on the data, as we can only keep the $m \ll p$ first components of the data, considering they are in a descending order. This allows for the data to be represented in less dimensions, taking only the most important m components. It is also important to know that in order to do this whole procedure, it is assumed that there is non-zero values across all the matrices, as this linear transformation requires doing the inverse of the matrix (although there are some tricks to work around this constraint, such as the matrix *pseudoinverse*).

In source separation, some ideas have been explored to separate sources by finding a different projection for each one on independent axis, for example by finding a projection that increases the signal-to-noise ratio of a singular source [27]. These systems are an evolution from the Fourier transform methods, as these assume the components are independent, whereas in PCA the components are found in their projection on the basis vectors, and are not assumed, but need to be found. Therefore, here the aim is to decorrelate the source signals, by finding the most independent possible axis of projection between each source, with PCA helping also in reducing the overall noise of the data, as the small components $m \ll p$ are discarded.

2.2 Independent Component Analysis

Independent Component Analysis (ICA) it's very similar to PCA, this method instead seeks to maximize the independence between the components instead of maximizing the directions of variance as in PCA. It's also a good method of classification, as projecting the data into the axis of maximum independence, or separation, is directly done. As it's kind of an extension of PCA, and also a more suitable method for separating, this is more widespread than PCA in this task. Also, sometimes the concepts are wrongfully interchanged, but the idea of both is the same (project an original signal into different axis in order to obtain some information), although the criteria for doing it is very different, although also based on statistical patterns.

Jumping right into it's applications, ICA works by separating the original components in the mixture into additive subcomponents for each source, also based in a statistical properties of the original data, same as PCA.

Having an original mixture X composed with different independent sources S, we can create a linear mixing matrix A by using the following equality:

$$X = A^T \cdot S$$

The objective is to find a matrix similar to the source matrix S, which will be refereed as $Y \approx S$, and a weight matrix that approximates the mixing matrix $W \approx A$ in order to find the following equation:

$$Y = W^T \cdot X$$

The objective is to find the weight matrix W, assuming there is non-Gaussianity between the sources, as this would neglect their independence, because a Gaussian distribution can be seen as random and not independent from source to source. There are also measures to assure the sources are independent and non-Gaussian, such as calculating the kurtosis of the distribution. [10]

Finally, ICA usually uses gradient descent as a method to optimize the weight of W, by iterating and using a cost function to find it's minimum error, calculated by a distance method such as MSE or LE error.

Some of the algorithms based on ICA have been proposed to resolve the source separation problem, such as single channel ICA [9], using maximum likelihood [33] or using beamforming [39].

2.3 Non-negative matrix factorization

NMF is a process whose objective is to factorize a matrix into two or more matrices, with the condition of not having any negative elements in them. This constraint greatly benefits its treatment (part-based models), for example by simplifying the operations we can do in them, or making it's combination with neural networks easier [22] (which only have positive firing rates, for example). Likewise, in signal processing, spectrograms have no negative values in them, thus making this property convenient and inherently fulfilled. In this point will only explain the case where we decompose the matrix in two matrices, which is the most common case when combining NMF and signal processing.

First of all, the NMF is not a precise process, and it can't -usually- be ideally solved, so the problem tends to be approximated numerically. Now, the two components that we factorize the original matrix into, have two separate meanings in audio processing: the first matrix, B, is called the basis, and has all the possible elements of the original matrix for each index; while the other matrix, W, has all of the weights, or coefficients, for each value of B in the original matrix, for each index as well. So then, being the matrix V the original matrix:

$$V \approx W \cdot B$$

In signal processing, the matrix V refers to the spectrogram of a signal, with dimensions $n\ x\ m$, n being the time index and m being the frequency bins. In the basic case we just want to approximate the same original matrix with this factorization, the matrix W will have a dimensionality of $1\ x\ m$, n being each one of the possible frequencies of V; while the matrix B will have a dimensionality of $n\ x\ 1$, n being each index of time in the spectrogram.

For exemple, if we want to reconstruct the original vector at the time index i we just need to multiply W by the first column of B:

$$V_i \approx W \cdot B_i$$

Expanding this theory to source separation, let us have a mixture with N sources all put together. The final factorization would be a linear combination of various basis and coefficients:

$$V \approx \sum_{n=1}^{N} W_n \cdot B_n$$

Or extending it by adding a singular time bin i and frequency bin j:

$$V_{i,j} \approx \sum_{n=1}^{N} W_{i,n} \cdot B_{n,j}$$

It can be seen how this equation has extended the basis matrices to have now a dimensionality incorporating each source as a new row or column, now the matrix B having a dimensionality of $N\ x\ m$ and the matrix W a dimensionality of $n\ x\ N$.

As it has been explained before, this result must be approximated, as an ideal solution is in most cases not possible. The descriptor used to calculate the difference between the original matrix and the basis combinations is the divergence, and this factor must be minimized. This divergence can be denoted as:

$$W, B = argmin_{W,B}(\ div(V, W \cdot B))$$

Where the divergence operator is usually the euclidean distance or the Kullback-Leibler Divergence.

The minimization of the divergence is usually achieved by using an optimization method such as gradient descent, on the first iteration initializing the matrices W and B with random values, and then updating it's values depending on the divergence operator we are using. [23] The algorithm is usually ran until a converging state has been reached or a certain number of iterations has been done.

Some advances can as well be made by adding for example regularization terms on the minimizing function, which at it's turn provide information about the system latent variables. [8] proposes adding two regularization terms in the cost function alpha and beta, so the new cost function would be:

$$W, B = argmin_{W,B}(\ div(V, W \cdot B) + \alpha_W J_W(W) + \beta_W J_B(B))$$

Where J_W and J_B are functions used to ensure smoothness on the basis and coefficients matrices, chosen depending on the application being made from the algorithm, which usually are weighted averages of data transformations from the matrices, data transformations being for example the absolute value or squared error.

Finally, multiple successful applications of NMF with source separation can be find, for example for musical transcription [41], which uses NMF to estimate the spectral profile and temporal information for each note in a source; or combining NMF with a probabilistic approach, adding Bayesian theory to perform source separation [48].

3 ARTIFICIAL NEURAL NETWORKS

Artificial Neural Networks are one of the most used tools in artificial intelligence, particularly in Machine learning, and in the later years have earned a high interest and popularity amongst many research fields. The biggest and foremost advantage ANN offer is the change of view they offer compared to previous views on artificial intelligence methods, which objective is now to replicate the way humans learn by simulating the way neural systems and connections work in our brain, by simulating the functionality of it's smallest element, the neuron. The final purpose is to create a system that can learn, from a series of inputs, and can give an output or more that it can later be used, and most importantly, has some useful meaning.

Although until the last decades we haven't seen the real potential of neural networks to solve complex problems in any situation, such as image recognition, speech recognition and synthesis or natural language processing [11], in the last closest years and decades, ANN as a concept have been around since 1950's. [36] Although it wouldn't be until many decades later when they would be fully operational, due to problems in the training of those networks. Thanks to the later invention of backpropagation, a technique used to train those networks, it is now possible to make them learn.

Neural networks can be simulated and recreated by first describing and modeling it's most basic element, which is the neuron. A neuron is a small biological cell located in our nervous system, with the particularity that can receive, process and send biological signals from other cells and to other cells. Even though neurons have other capabilities and functionalities, we will only focus on such as the way these interconnect themselves, and the way they receive stimulus, or we could call it information in the inputs, and from that information can process an output stimulus that can be connected to either another neuron or can be a final result. This response, generated by an internal potential generated by it's inputs, can be mathematically translated to a function, which is usually called an activation function, and defines what will be the output signal derived from the neuron. This activation function can turn out to be very important on the overall result of a network, as it is the mean of creating non-linearity responses from aforenamed networks.

The simplest way to understand these concepts it's to understand the perceptron [36], which we can define as an ANN with just single neuron, one of the first attempts at simulating a neural network, with the capability of being used as a linear classifier, using the linear equation formulation $y = Ax + b$.

Let us have a dataset X with N variables $X = \{x_1, x_2, ...x_N\}$, we can think of them as a series of inputs coming from N different neurons. Those inputs will directly be connected to the neuron, but with the particularity, that in the same way a real neuron behaves, each input will have a different influence in the output the neuron wants to give. Therefore, each input N will have assigned a different weight ω, therefore we will have a vector of weights, with one weight for each input $W = \{\omega_1, \omega_2, ...\omega_N\}$. Aswell, we need to add an extra parameter called bias in the model, we can think of it as the shift of the previously mentioned linear equation, and it's needed to obtain a correct result; we can add it as a new input x_0 with a weight ω_0. Finally, the neurons behave by "firing" a signal after a threshold has been surpassed, therefore it is needed to do a weighted sum of all of the variables by it's weights in order to obtain the internal value of the neuron. A function is added in this neuron in order to process this internal value, and it's this function, namely called transfer function, that calculate the final output of the perceptron. We can draw it as such:

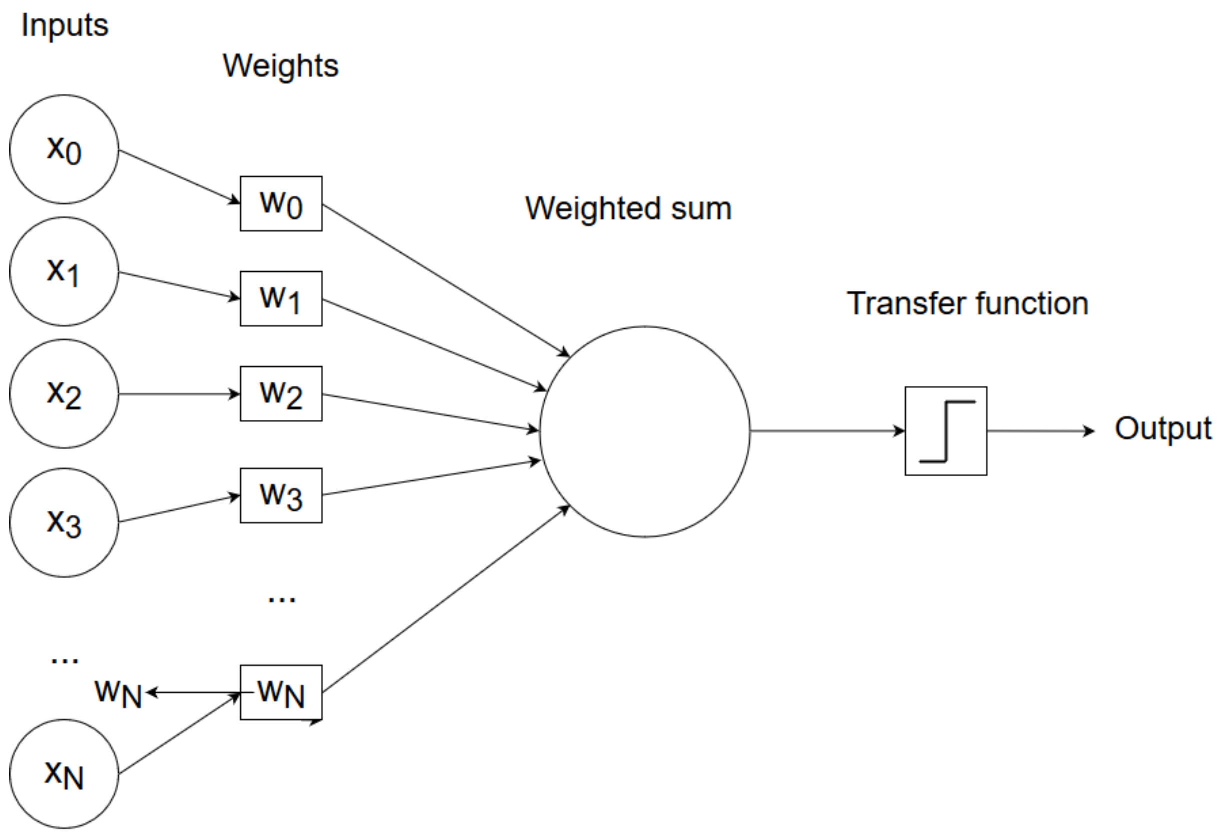


Fig. 1: Visual depiction of the perceptron. In this case we have N inputs, representing different variables, and we can see how a weight is added to each input before being sent to the neural unit. These inputs are then processed by first doing the weighted sum of all of them and sending then through a transfer function, which finally gives a single output.

The transfer function is also refereed as the activation function, and it's the part of the perceptron that gives the final output of the neural unit. Although the perceptron can only resolve linear problems, when creating higher complexity networks with multiple neurons and layers, if we use a non-linear activation function, it is possible to also get solutions to non-linear problems, which is one of the areas where deep learning shines the most, later this point will be further extended.

Going back to the neural units, each transfer function is also specific to each neuron, the same way it's weighted sum is, this means that each neuron will have the ability of having different impact on the data, by being able to potentially activate on different stimulus. This can also be understood as each cell will act differently depending on the data patterns that has as an input, therefore generating different patterns as an output.

The most used transfer functions are the following, which all happen to be non-linear transfer functions, although in some cases it may be convenient to use linear transfer functions:

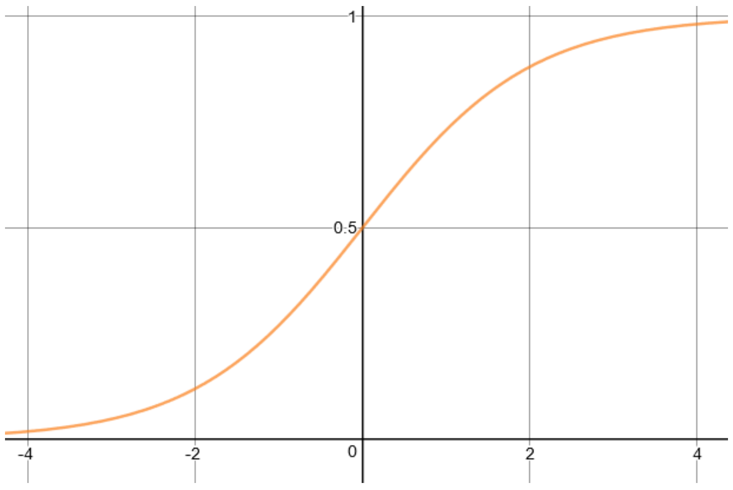


Fig. 2: Sigmoid activation function plot. Under: sigmoid function equation

$$S(x) = \frac{1}{1 + e^{-x}}$$

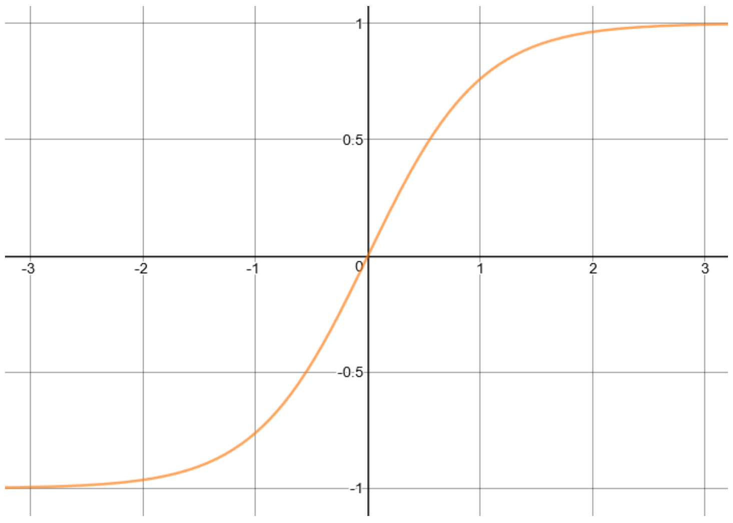


Fig. 3: $tanh$ activation function plot. Under: $tanh$ function equation

$$S(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

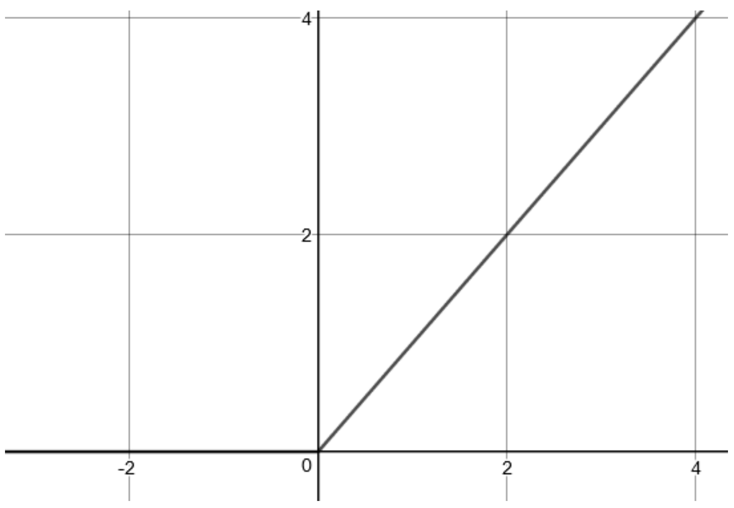


Fig. 4: Rectifier activation function plot. Under: rectifier function equation

$$max(x, 0)$$

The axes for all of them represent the same thing: the horizontal axis represents the input the transfer function receives, and the vertical axis represents the output of the unit for an input sent. The domain of these functions are important to take into account aswell, for example both the sigmoid and tanh functions can output values in the range $(-1, 1)$, non inclusive; while the rectifier function will output values in the range $[0, \infty)$. This domain needs to be taken into consideration in some cases where for example we know what kind of numerical outputs are needed in the output of an unit, for example, if we only want positive values without any limit restriction them, the rectifier function is the most convenient. Still, in most cases there is not much difference between transfer functions when the domain is the same, but some are recommended over others depending on the final application we want to make out of the network.

The output these functions give in our perceptron can be named a, and can be written as such:

$$\hat{a} = \sigma(w_0x_0 + \sum_{i=1}^{N} w_ix_i) = \sigma(b + \sum_{i=1}^{N} w_ix_i)$$

Where σ represents the activation function, which takes the summatory of each input by its weight to compute the output a. It's important to see how we have changed the first elements at the zero index, to the parameter b, which represents the bias as we have explained before.

As a note, rectifier function (ReLU) is usually the overall most used activation function especially in applications with large networks or related to image, first because it only outputs positive values, which are the ones present in image data representations, and as we said before these values aren't limited between a certain threshold; but also because the function is just a comparison between the value and zero, and the output will be either zero or the same value as it was in the output. As it's a very simple operation compared to other functions, and sometimes a network may have hundreds or thousands of different neurons each with their own transfer function, usually it's the most recommended solution just to save computational power, and at the same time does a good job of replicating the way human visual cortex works [2], so it's widely used in image, most specifically in Convolutional Neural Networks which we will later explain.

3.1 Training Neural Networks

As it has been explained before, artificial neural networks don't only take a heavy influence from biological neural networks, but they attempt to replicate its most important functionality, which is learning. As the ANN's were invented, this was the first problem encountered, because there wasn't a clear method to reliably and efficiently train those networks.

To train any network, first is needed to feed the network with information, which is the data that contains the information that will be used to make the network learn from it and later to give us the output. Therefore, the second thing we need is an output, which is the desired outcome from the information feeded, which we can compare to the actual output of the neuron. This action, of giving information to a network and observe its output is called the feed-forward step. The same rules as before apply to calculate this output, using the same output equation we demonstrated before, but for each neuron, it can be noted:

$$\hat{a}_n = \sigma_n(b_n + \sum_{i=1}^{N} w_{i,n}x_{i,n})$$

For each neuron $n = \{1, 2, ...R\}$ where R is the total number of neurons, being the final output of the network in this example the value of a_R (considering the network, not the unit, has only one final output, which isn't necessary by any means, usually multiple outputs for a network are considered.)

The next step, is to calculate what is the error between the network output, which we can call $\hat{a}$ and the desired output a. This error, whose objective of the training is to get it to be the smallest value possible, is also called the cost of the network. This cost can be calculated by any metric that can be used to numerically obtain the distances between the output $\hat{a}$ and the desired result a. This metrics in machine learning are refereed as the loss/cost functions.

Some methods used are the Mean Squared Error (MSE), Squared Error (SE), L1 loss, cross-entropy loss, etcetera. Once again like in the case of the activation functions, the final chose of the lost function it's purely based on the application being made and how we want the network to behave. In general, the error is represented as:

$$J(\theta) = \Psi(\hat{a}, a)$$

Where Ψ represents any loss function chosen, and takes the desired output a and the network output $\hat{a}$ to give a representation of its divergence as a numerical difference between them. The parameter θ in $J(\theta)$ represents the parametrization of the elements of the network, such as weights for each element, outputs, etc.

Finally, now that the error generated by the network has been defined, the learning step is the only element missing of the training, where this same error is used for the network to correct each neuron's input weights to try to minimize this error. This is done by a method called backpropagation [40], which also solved the problem of not being to reliably train neural networks.

The basic principle of backpropagation is to think of the network as a number of different elements, which all have had an impact on the output result of the network, taking into account that each one of them has a singular presence to the final error of the network. The idea is to take this final error, and start propagating back to the network in the reverse order of the feed-forward step, but doing it first by layers, and then by elements on each layer. Despite it using the final error $J(\theta)$, the network backpropagates the error by using the error gradient, with the aim of updating all of the weights of the network to a value that minimizes this same error. This makes the step highly complex and not-trivial, because the partial derivatives for each node of the network need to be taken into account and calculated. These partial derivatives can be formulated as:

$$\frac{\partial J(\theta)}{\partial \theta_{ij}^{l}} = \frac{\partial J(\theta)}{\partial s_j^l} \frac{\partial s_j^l}{\partial \theta_{ij}^l} = \delta_j^l \cdot h_i^{l-1}$$

Where $J(\theta)$ is the parametrized cost function, which is derived in respect to each activation node θ_{ij}^l. The parameters for each node represent l for the layer number, and j as an element on that layer. This decomposition has two parts, the first, δ_j^l represents the derivative $\frac{\partial J(\theta)}{\partial s_j^l}$ and the second element h_i^{l-1} the activation function in respect to the weights. This last element can be easily computed for each node, but the first element needs to be derived, as it's the element corresponding to the error gradient for each node j.[4] The contribution from each cell to the error can be calculated with:

$$\delta_i^{l-1} = \frac{\partial J(\theta)}{\partial s_j^{L-1}} = \frac{\partial J(\sigma^{L-1} s_j^{L-1}, y_j)}{\partial s_j^{L-1}}$$

And finally backpropagated to the rest of elements, to find the contribution from each layer:

$$\delta_i^{l-1} = \sigma'^{(l-1)}(s_i^{l-1}) \sum_{j=1}^{d(l)} \delta_j^l \theta_{ij}^l$$

Lastly, while explaining backpropagation two important elements have been introduced:

one being the gradient of the error, and another the partial derivatives of said gradient. The gradient is a N-dimensional space that contains the maximum slope of a function which can lead to the maximas (if the slope is followed through) and minimas (if the opposed direction of the slope is followed) of a function by iterating through it. To get to these maximas and minimas, the gradient descent algorithm is usually used, which sometimes will just be refereed as the optimization algorithm, and is an iterating method that lets us move across the gradient of a function. The updating step is done with the following equation:

$$\theta :\Rightarrow \theta - \eta \cdot \bigtriangledown_\theta J(\theta)$$

Where $\bigtriangledown_\theta J(\theta)$ is the gradient of the cost function, derived with the parameters of θ, and the η value represents the learning rate, which is a value used to tweak the rate of movement we go through the gradient. A big step will lead us to a reachable minimum faster, but a smaller step will give us a more precise minimum value. To add to this, this gradient descent algorithm will guarantee it gives us the global minimum only for convex gradient surfaces, but for non-convex surfaces it might stick to a local minimum, which would make our problem not resolved. To do so many improvements can be made to ensure we don't get stuck in a global minima, but as this is not the focus of the work no further details will be needed.

3.2 Multilayer Networks

Up to this point it has been explained how an unit on a neural network behaves, and the basic principles of training for networks. But aswell it has been explained know how such a network can only be used to solve simple linear problems for classification, therefore making it a not so useful network to solve real world scenarios, which are much more complex and often require non-linear modeling of the problem. The first step to try to be able to model problems and resolve them, is to try to create new architectures with the elements presented up to this point. Therefore the first multilayer networks appeared. What it is considered a multilayer network doesn't have a standard shape or a golden rule on its architecture to make them work better than another, but they usually have three different layers to consider when implementing them. Considering that a network must have an input and an output, we can also consider them layers in a neural network, and each unit behaves in the same way described in the previous points; in the example later shown the output layer is made out of two neural units therefore giving as the final output of the network two different values. The third layer is located between them, and acts as a mediator between the inputs and the output(s) of the network, and it's the layer that will learn from the input in order to give the first outputs, and its elements are a discrete number of trainable neural units, the same ones explained before. The easiest way is to picture this general network shape:

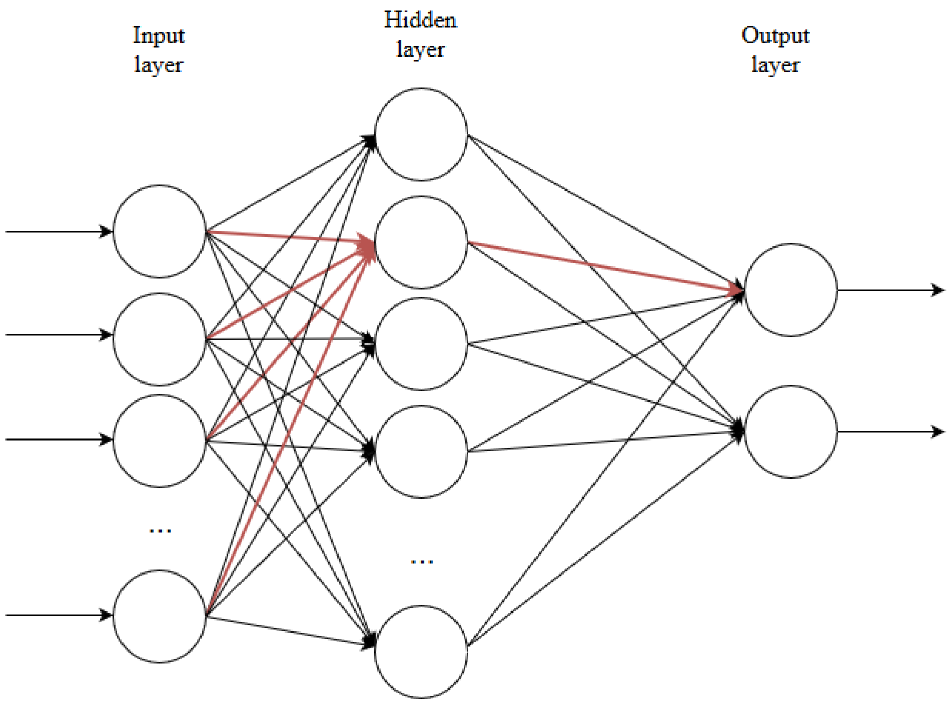


Fig. 5: Depiction of a multilayer network with 2 outputs. In red, we see how the backpropagation to a neuron affects all of its input weights later on. This network has two neural layers, the first one called hidden layer, and the final one has two units and it's called output layer, which gives the final output of the network.

The concept of hidden layer is introduced here because it's normal to reefer the neural layers in a neural network as hidden layers, since we don't directly see the outputs of the layer, as we can see, they are first send to another layer with two neural units.

As it can be seen in this network picture, the final layer gives out two different outputs. Doing so for example would be useful when combining with a softmax function for the output in order to be able to classify images, as one output will trigger when classifying an image to one class while the other will remain quiet. [42]. Also highlighted in red, there is an example of what happens when a neuron backpropagates from an output to the input in a multilayer network, it can be appreciated how a single connection to the output forces all of the input weights updated in a single neuron.

Finally, the benefits of using multilayer networks is that it can output non-linear solutions, plus the increased complexity of the network architecture may trigger the neurons to learn more specific tasks or behavior patterns that can give out increasingly complex outputs with better and more accurate results.

3.3 Deep Learning

Deep learning is the name given to multilayer networks with a multiple number of hidden layers between the input and output layers. What these layers do is generate and learn abstract data representations, which seem to mimic the task our brain does feature extraction by processing the data in different stages. That means, in a deep learning network, data patterns are generated in order to later process an output, so it can be understood as a series of multilayer networks interconnected between them, but with the higher complexity that beforehand it is assumed that it can't be known how those connections are made and learned, since they are on the hidden layer, hence the name and the abstract idea these networks offer. Despite this complexity, which makes training this networks a difficult task, has offered resources to improve algorithms based on image processing, speech recognition and most disciplines and deep learning is now the dictating state-of-the-art method at this day and age. [20]

As in the multilayer networks, for a input given to the n layer, gives the output a_n with the same principle:

$$a_n = \sigma_n(b_n + \sum_{i=1}^{N} w_{i,n}x_{i,n})$$

The beauty of deep learning networks is that they allow us to create complex and not trivial network architectures, as they extend the size of a regular network, and allow us to choose any number of inputs or outputs per layer, or using different activation functions for each element, hence the activation function is depicted as σ_n. This also means that we can now use an architecture to model a problem with a more specific manner, therefore being able to process more complex problems, hence it's why deep learning is now being used all-around.

We can depict what happens inside a deep learning network as a multilayer network with a black box inside of it:

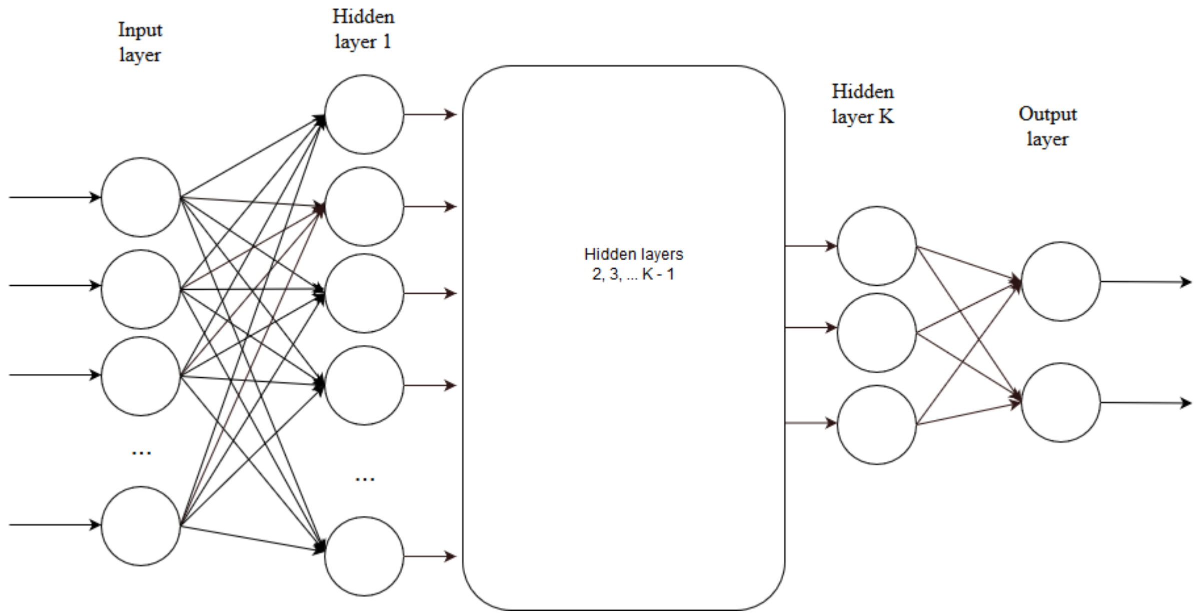


Fig. 6: Deep Learning framework with K number of hidden layers. We can imagine the middle 2 to K-1 layers are more multilayer or deep learning networks connected to the first hidden layer, which give an output to the last hidden layer, just to show the flexibility they offer.

Finally, the same rules as the previous networks are extended to deep learning networks, from the parametrization to the training of them, as in backpropagation. (Section 3.1)

3.4 Convolutional Neural Networks

Convolutional Neural Networks, by short CNN, are neural networks inspired by the way humans perceive information on the visual cortex, and attempts to recreate it by taking into account spatial information from an input of data to process it just as the brain would process a set of features from a perceived image. Once again, by specializing on a kind of a problem, this time in image-related tasks, have made these networks the rule state-of-the-art for image processing, since they have demonstrated its excellent performance an reliability on image-related tasks.

As perceiving images is a task that involves two-dimensions (or three if the time domain is taken into account, but we will focus on the simplest case of two dimensions), our data must also be presented in the same way. Now it can be imagined the data as present in a plane; back in the 50's visual cortex was proven that some visual cortex neurons have small receptive fields that they take information from, and also react to specific patterns of data. [17] This structure is replicated with CNN: instead of working similarly to the previous networks where a neuron in a layer is connected to all of the outputs of the previous layer, here each neuron is connected to only a small area in the previous layer, called a receptive field. This receptive field in a CNN has a discrete value and a defined area, and we can think of it as a sliding window on a 2D representation of data that connects its output to a neuron. We can see that in the following example:

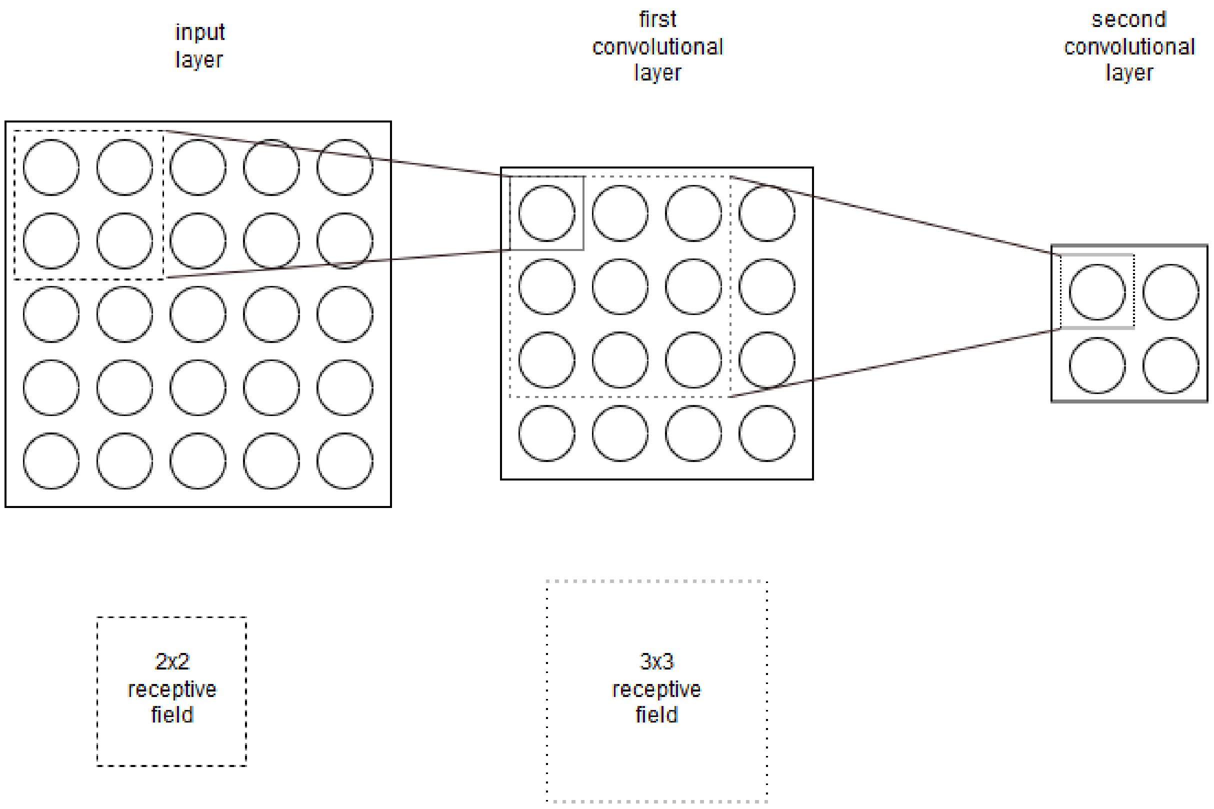


Fig. 7: Convolutional Neural Network with 3 layers. We can see how we had an original size of 5x5 inputs, and when passing a 2x2 receptive field through it, with a stride of 1, we get a resulting layer of 4x4 elements, each one holding information of the 4 elements in the receptive field on the previous layer. Finally applying a rec. field in the first convolving layer, we obtain a third layer of 2 by 2 elements, which holds information of the previous two layers.

It can be seen how the network shape and outputs haven't been built manually, but directly depending from the filter shapes and sizes we have chosen (the filter shapes also doesn't need necessarily to have a square shape). Not only these two parameters matter, but we can also use a set stride value (how many units we displace the filter each step) different than one, we can also set a dilation factor (the spacing between units in the receptive field), plus zero-padding can be applied on convenience on any of the layers.

A very important concept that this layers also offer, it's that all of the neurons that are spawned from the same receptive field (also called filters, when referring to the singular output each one gives), i.e. all of the neurons of the first or second convolutional layer in our example, share the same weights. This not only is to ease the computational requirements for those networks, but because in image processing, it is assumed that if a feature has been successfully learned in one of the filters, the rest of them in the same level should, therefore all of them learn the same features on that layer. Since we have changed how the weight works for each layer, the new output from forenamed layers is the following, being a_k the output, being the input layer of dimensions MxN, and the hidden layer having dimensions $(M-A+1)x(N-B+1)$ being AxB the receptive field:

$$a_{j,k} = \sigma\left(b + \sum_{l=0}^{l=A-1}\sum_{m=0}^{m=B-1} w_{l,m} \cdot x_{j+l,k+m}\right)$$

This brings an interesting point, with deep learning, with a big enough network, and with the same kind of problem, shouldn't we actually get to resolve the problem the same way a CNN resolves it's problem by finally learning patterns in small areas and sharing weights for the same features, amongst basically finding the same features from the input? The answer is yes. But this brings another point: if we want for example train a network that feeds off images as inputs, we imagine we use small images of 640x480 pixels: in a standard deep learning network, connected to this input, we would need just for the input 307.200 neurons. This is not only a computational problem, as the number of neurons would increase with the deep layers, but also because training complex and big deep learning layers has proven to be a very difficult task, spawning many problems such as the vanishing gradient problem or getting stuck in the gradient saddle points or local minimas. [25] Therefore, one of the solutions is to use either smaller networks or specialized networks, and CNN, when related to image processing, offer both advantages.

Having said that, there are also further optimizations that can be made on a CNN, such as adding a pooling layer, which from a small window (similar to the receptive field) just keeps the maximum value (this method it's called max pooling; which it's the most used pooling method); we can also implement dropout layers, where we can randomly remove connections of neurons between layers, which is a good method to avoid overfitting and optimizing the training of the nets [43]; or adding batch normalization layers, which what they do is to normalize the output of the neurons with values between 0 and 1, they always being positive, which is the way the neurons behave. Aswell, it's also common to add regularization terms in the cost function, see L1 and L2 regularization terms. [28]

3.5 Autoencoders

Autoencoders are a special case of ANN's, they use the same theory and basis to get features from a set of inputs, but they have the specialization having the objective to compress data efficiently by finding data correlations with a relative small number of neurons in the network compared to its inputs. It is usually an unsupervised technique, as the encoder himself needs to learn those representations without any external help. As in this thesis second part includes an autoencoder based on CNN's, the examples given in this step will be based in convolutional autoencoders.

Autoencoders are usually separated into two basic parts: the first is an encoder, and it's the first step of the network where it takes an input and convolves it with a set of N windows, or filters, to obtain the same amount of data but represented by usually a much smaller number of elements. As the name suggests, this first step of the autoencoder's objective is to compress the data as much as possible, by finding data patterns and correlations that can be used to save the same data with much less units. The output of the encoder can be written as the following equation:

$$y = \sigma_1 \left(b_1 + \sum_{i=1}^{i=N} w_{1,i} \cdot x_i \right) = \sigma_1(W_1 \cdot x + B_1)$$

Once the encoding step is done, which usually is done over a set of different encoding layers, the final decoding step is made, where the data is decoded by applying a deconvolution to the compressed output with filters of the same size used to encode it, but in a reverse order, if we want to obtain a deconvoluted output of the same size as the input, otherwise the dimensionality will be different. The main objective on the decoding stage is to finally convert the data into a representation as close to the original a possible. The reconstructed output from the decoding layer can be written as such:

$$\hat{x} = \sigma_2 \left(b_2 + \sum_{i=1}^{i=N} w_{2,i} \cdot y_i \right) = \sigma_2(W_2 \cdot y + B_2)$$

And when putting together the two functions, we get the total autoencoder output function:

$$\hat{x} = \sigma_2 \left(\, W_2(\, \sigma_1(W_1 \cdot x + B_1)) + B_2 \right)$$

Note: the previous equations explained autoencoders as the combination of CNN to perform data compression, but autoencoders don't necessarily need to be composed out of convolutional layers, for example, we could use a multilayer network where the hidden layer is used as a bottleneck layer, that is, having a relative small number of units compared to the input and output; then, the previous equations still hold and the wanted functionality is the same.

Finally, we have shown how autoencoders can be useful to compress data, and that is one of their usabilities, but an autencoder structure can be taken advantage of and be able to perform solid data denoising, which happens to be what autoencoders are most used on and they excel at. [47] For example, if we feed the network in the input with corrupted data, and we then compare the output with the same data, but without the added corruption, when calculating the total error and later backpropagating, the network will learn features that help turn this corrupted data into clear data, through using coding and decoding to find denoising data patterns.

There is also many variations for autoencoders, for example variational autoencoders (VAE), which use probabilistic theory to encode data, coding the data into normal distributions that can later be tweaked to decode into different outputs depending on the latent parameters on the net and how we project them (for example to generate new data from an input, which could be images [49]), sparse autoencoders, where we have more hidden units than inputs, but some of them are forced to remain deactivated, etcetera. In this thesis we will focus on standard autoencoders related to data denoising.

3.6 Recurrent Neural Networks

Or for short RNN, are networks where by the design of the connections between the units, we add a factor of memory on the network, thus an input in a certain time will affect the inputs feed in the future. These connections can work both ways, if there is some feedback coming back from a recurrent connection, the net is called a bidirectional RNN.

A good way to visualize this networks is think of them as a regular network, which has a connection connected to the same network but in a different time index:

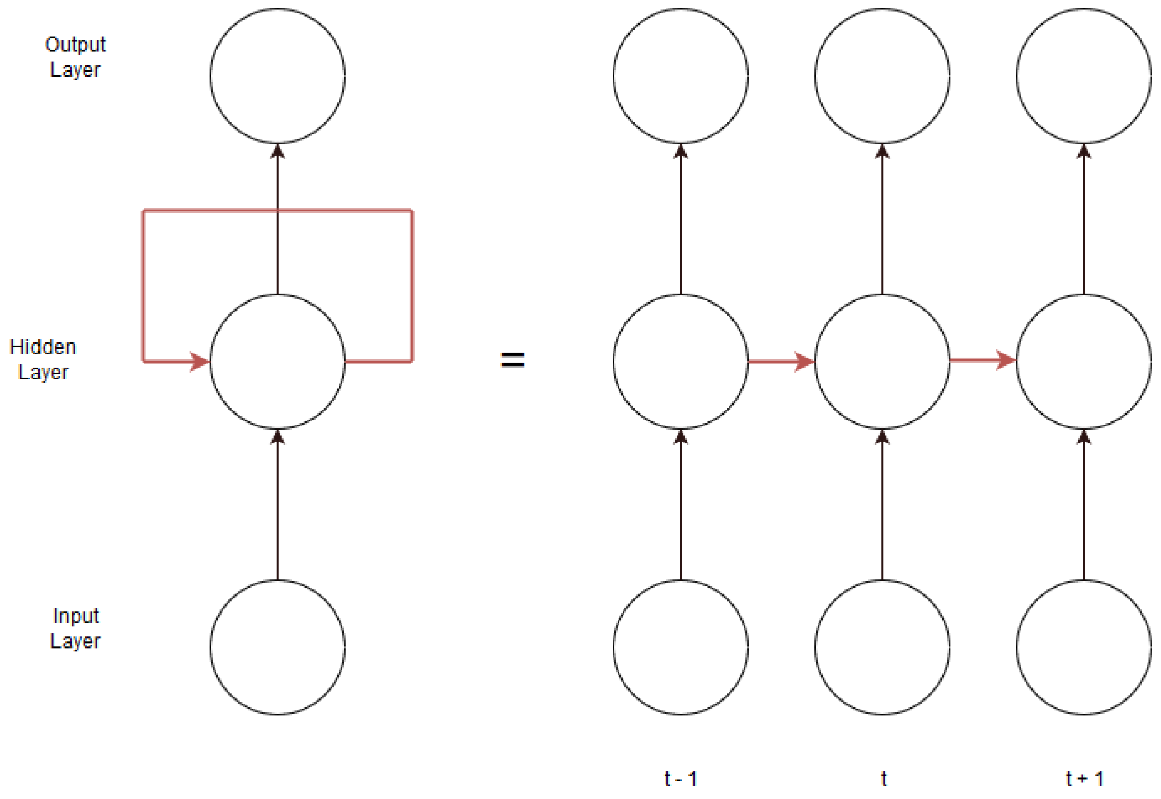


Fig. 8: On the left: RNN with one recurrent connexion on it's hidden layer. On the right: the same network, but in a time context with the time indexes $t-1$, t, $t+1$.

Since these networks take temporary information from the input, they are especially good when the input data is made of sequences, for example sequences related to speech processing, as in various phonemes sent after another; NLP (Natural Language Processing), as in sending a text tagged with grammatical categories to output a prediction on what will be the next word; or in the case of signal processing, feeding the network a stream of time frames to learn data patterns and correlations between them. For a simple recurrent network, the ouput could be written as the following function:

$$a_t = \sigma\left(\sum_{i=0}^{i=N} w_{t,i} \cdot y_{t,i} + w_{t-1,i} \cdot y_{t-1,i}\right)$$

For an output a on the time index t, where we can see how the time index $t-1$ affects the output by being added to the current time step. At this example we suggest that the output generated by this function is perpetually being added to the input on the next time step, but this would be a bad practical example, for various reasons: first of all, we are giving infinite memory to this network by doing this connection, without any regularization with context on the time index we are and up to what time index we want to hold information for. This can yield severe errors, which involve the gradient: similarly to deep networks, training RNN networks has the risk of getting to unwanted results, such as getting stuck in local minimas, saddle points or getting the vanishing gradient problem. Hence, the solution in this case, is to limit the memory RNN have, by limiting the information hold by the recurrent network each time. An optimization of RNN are the so called LSTM (Long short-term memory -units-), which are a special type of neural units that are optimized to work with information on a time-context space. LSTM units are in general made up from four different functions inside them, which are usually refereed as gates. These functions attempt to mimic the way our memory works, by imposing constraints on the data's time-context, in order to obtain a model that better imitates short-term memory in humans, including concepts like remembering and forgetting.

The first layer of the LSTM unit is the drop-out layer, where this layer will choose what information will be processed by the unit and which will be discarded. It does so by comparing the previous output of the unit in the immediately previous time index, a_{t-1} with the current input of the unit, x_t. So for this first gate, the output is:

$$f_t = \sigma_s(W_{f,t} \cdot x_t + W_{f,t-1} \cdot a_{t-1} + b_f)$$

Where the σ_s subindex reefers to the sigmoid activation function, usually used at this step, and the subindex f reefers to the parameters of the function for this ”forget” layer.

The next gate is usually the input gate, which takes the current time index inputs x_t and chooses which of those inputs will be added to be processed in the LSTM layer and which discarded, or in other words, it chooses what inputs will be used for this iteration. The output function is the following:

$$i_t = \sigma_s(W_{i,t} \cdot x_t + W_{i,t-1} \cdot a_{t-1} + b_i)$$

Once again, the σ_s is the sigmoid activation function, and the i subindex reefers to the parameters of the input function.

Next, there is the output layer, which despite it's name it's not the final output of the unit, but the final output from the inputs to be sent forward in the unit. What this gate does is to basically filter the previous gate outputs before sending them to the cell memory and output the final value of the unit. Once again, we have a sigmoid activation function and a series of parameters refereed to the output function:

$$o_t = \sigma_s(W_{o,t} \cdot x_t + W_{o,t-1} \cdot a_{t-1} + b_o)$$

As we have seen with the first three gates, what they are actually doing is behaving as a small neural networks, more specifically they are binarizing the data to feed them into the following gate just differing input-wise with each gate.

The next step in the LSTM unit it's the updating of it's memory, that is, take the now current state with the removed previous inputs and the new chosen inputs, and therefore update the old cell state c_{t-1} into the new one c_t. To do so, it does the following operation:

$$c_t = f_t \circ c_{t-1} + i_t \circ \sigma_c(W_{c,t} \cdot x_t + W_{c,t-1} \cdot a_{t-1} + b_c)$$

Where the $\circ$ operator it's the Hadamard product (each element i, j of the first matrix is multiplied by it's i, j analog on the second matrix)

Finally, the final output of the LSTM unit it's the following:

$$a_t = o_t \ \circ \ \sigma_t(c_t)$$

Where σ_t in this case is the $tanh(x)$ activation function with x being the current memory state of the unit, multiplied by the output of the first gates of the cell, with the operator being the Hadamard product once again.

These previous explanations help to see how we can solve some of the current problems in RNN by dividing the memory problem into small tasks in small hidden networks, and it's thanks to that that many applications that weren't possible due to constraints with RNN architecture, are now possibly solved by LSTM. This is especially interesting for us, as most of these advancements have been made in the signal processing discipline, and have been successfully used for speech recognition and polyphonic music modeling tasks. [13]

4 BASELINE MODEL

Up to this point we have seen a multitude of methods and tools historically used to solve the source separation problem. Now we are going to focus in a single state-of-the-art implementation, that in our case it's an algorithm developed in the last years using some of the most modern tools we have seen involving neural networks.

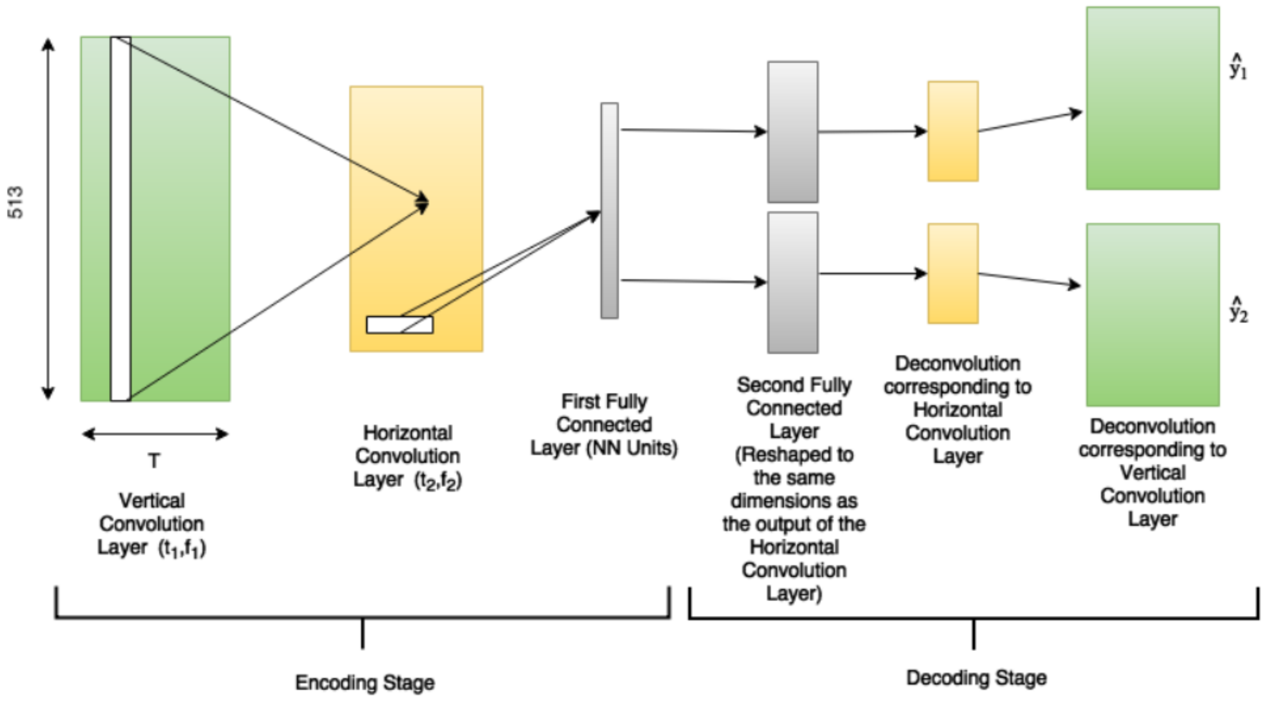
The state-of-the-art algorithm chosen is part of *DeepConvSep*, proposed by P. Chandna [*Low Latency Online Monoaural Source Separation For Drums, Bass and Vocals*] [4], which involves a combination of deep learning with Convolutional Neural Networks (CNN) as in an autoencoder and a mask-based separation algorithm, which is built to perform denoising from an audio mixture into four different -previously- known number of sources, which are drums, bass, vocals and others (this last being the rest of the mixture not including the other sources). The final intention of the network is to find and apply different denoising-masks to an input in order to obtain the final sources, by having a different mask for each one of them. In this point it is going to be explained the basic principles of this net, and how it works step by step. Aswell, from now on this implementation will be refereed as the *baseline model* instead of *state-of-the-art model.*

a) Model data input

Before entering to the explanation of the architecture of the algorithm, it's important to state what are the data as in the network inputs used, and how does them relate to CNN. Previously it has been explained how CNN's are used in image processing because of their benefits in implementation, for example its use spatial information from an input, i.e a 2-dimensional image, to perform convolutions and learn compressed representations of data. As in audio signals we don't have this clear spatial information (this is not referring to time-space information, but about location-space), the solution is to somehow create a representation of audio signals which has this location-space and it's meaningful: as explained in the Source Separation Problem (Section 1.1), transforming an audio signal from the time-space to it's time-frequency space can easily be made performing the Fourier Transform. The baseline model proposes using as an input to it's network a series of Short-time Fourier Transforms (STFT's) frames created from the original signals, in order to obtain a representation of named signals in the time-frequency space. There is a third dimensionality on any STFT representation, which is the magnitude. This magnitude represents the contribution of a frequency on a singular time bin. Therefore, the inputs here are "images" of the signal, where the x axis represents the time information of the signal, the y axis all of the possible frequencies of the signal, and each (x, y) bin has a magnitude representing the weight of the y-th frequency on the x-th time index.

b) Architecture

we previously explained how we generate the inputs from an audio signal in order to be able to work with a classical CNN architecture, which is the one of the new viewpoints compared to the classic ANN-based or method-based models blind source separation. First of all it's useful to picture the whole network architecture in order to understand how a CNN can perform denoising and separate different sources:



513
T
Vertical Convolution Layer (t_1,f_1)
Horizontal Convolution Layer (t_2,f_2)
First Fully Connected Layer (NN Units)
Second Fully Connected Layer (Reshaped to the same dimensions as the output of the Horizontal Convolution Layer)
Deconvolution corresponding to Horizontal Convolution Layer
Deconvolution corresponding to Vertical Convolution Layer
$\hat{y}_1$
$\hat{y}_2$
Encoding Stage
Decoding Stage

Fig. 9: *DeepConvSep* architecture, extracted from [6], depicts the overall architecture of the model. Note that here only two outputs are pictured, against the four outputs that are actually in the implemented model, one for each source.

The feeding-forward direction of the model goes from left, where the left-most figure represents the data input, to the right, where the last figures represent the outputs of the model. Aswell, only two outputs are described in the above picture, but in the actual implementation there are four outputs, one for each source. The order of the feed-forward step of the network is the following, layer by layer:

- Vertical Convolution Layer: This first layer performs a vertical convolution, so it gives up the typical square window in favor of using a vertical window, with the aim of getting a representation of the frequency features of the input, which since it's made of musical signals, we can name timbre features. In the baseline model, 50 different filters are learned in order to generate features from the data.

- Horizontal Convolution Layer: The second layer takes as an input the direct output of the timbre-layer, and with similar synonimity with the first layer, this one uses a horizontal window with the aim of creating a representation of data that is based in the time-varying features of the input. Since the last layer took frequency features of the initial input, this layer will now output a time-frequency encoded representation of the initial data. The baseline model uses as a standard filter size half of the time context length from the initial input (and aswell the same length from the previous output, as the time context hasn't been modified in the first CNN layer); aswell, it receives 50 filtered outputs from the previous layer as inputs, and generates a reduced number of 30 filtered outputs to feed forward to the next layer.

- First fully connected Layer: Now, this output is send to a fully connected layer. If the resulting output on the encoding stage was a series of convoluted matrices of NxM elements, this fully connected layer will have a total number of $N \times M \times K$ units, K being the total number of filtered outputs of the previous layer, which as it was said is 30 by default. This layer objective is to ease the obtainment of non-linear features from the previous inputs, and it's a shared layer for all the sources outputs, and acts as a bottleneck layer for the network, and therefore it's meaning in the network it's to compress the CNN features in order to get more efficient representations of data. It uses a Rectifier activation function (ReLU) to output it's values, due to the benefits mentioned in previous points. It also uses a small number of units, 128 by default, in order to avoid overfitting and computation cost and complications in gradient descent computations.

The next steps correspond to the decoding stage of the autoencoder, and are reproduced four times in parallel, one for each source, in order to obtain four final independent outputs corresponding to each source.

- Second fully connected Layer: the point of having a second connected layer comes to the relation that we want to obtain four different outputs as in explained previously, therefore with that need of creating four independent outputs, or masks, as we have said before, comes the obligation of somehow letting the network choose for each output which are the best features to represent each independent source data. Therefore, the second fully connected layer will act as a filter to which features are the best to create the most reliable output for each source. The number of features outputted by this layer are the same number of features outputted by the horizontal convolution layer, which is important as if we want to perform a valid deconvolution and have the same number of values we had at the starting, without any major information loss.

- Horizontal Deconvolution Layer: this layer will take the number of features from one of the second fully connected layers, and will perform the deconvolution of it's elements, which is the opposite step made on the Horizontal Convolution Layer. It will output the timber-based representation learned from the network.

- Vertical Deconvolution Layer: identical to the previous layer but will apply a vertical deconvolution, the opposite as in the Horizontal Convolution Layer. This layer will give the final output of the network, which we will interpret as the mask of a source, and will later be applied to calculate the loss of the network.

Once these steps are done in the algorithm, which correspond to the feed-forward step in the network, the mask for each source is computed. Before that, it's important to say, that before we said that these networks are running in parallel 4 times, one for each source, but in reality, we don't need to do that, as we can compute the output of 'others' by taking the STFT mixture input, and then subtracting the STFT outputs given by each one of the sources except 'others'. We then compute the mask for each source by applying the following equation, similar to Wiener filtering (Section 1.1):

$$Mask_{source}(t,f) = \frac{|\hat{y}_{source}(t,f)|}{\sum_1^N |\hat{y}_n(t,f)|}$$

Finally, this mask is used to compute the final output for each source as in:

$$\tilde{y}_{source}(t,f) = Mask_{source}(t,f) \cdot x(t,f)$$

Where $x(t,f)$ it's the original mixed STFT given as an input to the network. With this small trick we are forcing the network to actually learn the masks of each source, instead of directly learning each source contribution in the initial input data.

c) Training

Up until now we have seen how this network, from a STFT containing a mixture of signals, performs a variety of data transformations and ends outputting four different STFTs for each source, which are created by applying the learned mask by the network belonging to each source. This means that the training step of the model needs the ideal target for each one of the final independent signals, to be compared with the model's output. To compute the error *DeepConvSep* uses the squared error (SE) to measure the difference between the network's output $\tilde{y}$ and the ideal output y:

$$E_{sq} = \sum_{i=1}^{N} \|\tilde{y}_n - y_n\|^2$$

Despite the model architecture and the error calculation justified in the baseline model paper [4], it was proved that this error function wasn't enough to let the network learn good data representations, especially for the "voice" source. To overcome this problem, corrections were made in the loss function in order to attempt to reduce the "others" presence in the cost function, for example by not directly calculating the error between the others source and its targets, but by calculating the differences with the other sources:

$$E_{diff} = \sum_{i=1}^{N-1} \|\tilde{y}_n - \tilde{y}_{\hat{n}\neq n}\|^2$$

$$E_{other} = \sum_{i=2}^{N-1} \|\tilde{y}_n - \tilde{y}_{other}\|^2$$

$$E_{othervocals} = \|\tilde{y}_{vocals} - \tilde{y}_{other}\|^2$$

And the total cost for the network is computed with:

$$E_{total} = E_{sq} - \alpha E_{diff} - \beta E_{other} - \beta_{vocals} E_{othervocals}$$

Where α, β and β_{vocals} are weights chosen to be 0.001, 0.01 and 0.03, respectively [4].

Finally, this whole implementation, which some of the previous points were based on, can be publicly found in *https://github.com/MTG/DeepConvSep/tree/master/examples/dsd100*, as part of the *DeepConvSep* repository (*https://github.com/MTG/DeepConvSep*), which includes other methods and tools to perform source separation.

d) Original Dataset

The chosen dataset to test this algorithm with was the Demixing Secrets Dataset (DSD100) [24], which contained 100 professionally studio-produced musical songs with a total of 100 different tracks. Each track contains the mixture, and each of the sources from the same mixture as separate tracks and as the final target sources to separate from the mixture. Although DSD100 also includes an evaluation framework, it wasn't used in the final evaluation task for the network. As we can see, this is a pretty small network with just 100 tracks, but due to the lack of better datasets at his time, it was the best data to work with.

4.1 Possible improvements

We have seen the principles of our baseline model algorithm, it's architecture and how overall works to perform source separation on a given mixture. Although it works pretty well, and gives good approximates on the separated source outputs, it isn't exempt from certain flaws.

First of all, the original designed system it's only designed for monaural audio, and to calculate the STFT's what it originally did was adding together the weighted average of each channel of signal, that is $\hat{x}_t = \frac{1}{2}(x_1(t) + x_2(t))$ where the subindexes represent the first and second channel of the stereo mixture. The error isn't on creating a monaural track by applying this approximation, but rather to discard original stereophonic recordings, with the loss of information that that this implies, in favor of training the dataset with a type of audio mixture it's not widely used anymore. Therefore it would be better to be able to train and afterwards synthesize files without the constraint of having to be monophonic, but stereophonic, since it's right now the norm on audio recording mixtures. As well, it's not only the fact that we are discarding *half* of the original recording information, but that stereophonic signals depicting the same signal imply that there is a correlation between the stereo channels outputs, and therefore if we use this information, we are potentially adding to our model important time-frequency features that will help us to obtain better audio related features to perform the separation.

The next problem this architecture has, is that despite giving good approximations on the sources outputs, there is also a present flickering noise present during most of the time in the outputs given by the network (of course, only perceived when performing the inverse STFT on it's output), plus some sources work better than others, for example bass and drums give better results than voice, while others is where there is more present noise. We believe that one way to further decrease these problems in the network, is to add a denoising network in each of the sources outputs, which also means that these denoising modules would be independent between them; this should be a good advantage, as in the network before, the whole encoding step of the algorithm was shared among all of the sources, while in this step each source will have it's each denoising module. Aswell, we thought of creating a network which takes into account temporal information from the inputs, whose only way of being implemented is by creating a Recurrent Neural Network (RNN); to further strengthen this RNN we will use LSTM (Long short-term memory) units, as we have explained before (Section 3.6)

Another problem we considered wasn't a baseline model related problem but a dataset-related problem. In the music production world, musical rights are very protective, and it's industry it's very reticent on to release original musical mixtures publicly. Therefore, one of the big problems we can't control or solve it's the total number of data we have to train our models. To kinda improve on this point, we will try to perform data augmentation on the original dataset, by creating new data input samples and feeding them to the network.

Finally, we want to mention that the baseline model is a mask-based source separation method, and it is related to CNN's by treating an input 2-dimensional representation of data (STFT's) as we would do with any image in order to find masks and apply them in order to separate sources from the mixture. Although this is not inherently a bad design, it's important to say that treating an audio signal as an image it's not the most correct way of performing this separation, so a change of point of view in this regard could be made to create better models and architectures, although this is not an improvement to consider in the baseline model, as it would imply the creation of a whole new model architecture.

4.2 State of the art

Our baseline model is considered a state-of-the-art model because it implements a newly approach to separating audio signals from a mixture, based on a CNN autoencoder. Although it does a very good job at it, it's not the "best" model or the more widespread one in this area; some other state-of-the-art models are worth mentioning in that regard, for example the UHL algorithm (named after it's creator Stefan Uhlich) [44] is a network which combines two other separately trained networks, one being a DNN whose data inputs are pre-processed through PCA and later weighted with a Wiener filtering, and the second one being a bidirectional LSTM network; also, both networks include data-augmentation techniques; the final network takes these two networks and blends it's output with the linear combination $\hat{s}_{i,final}(n) = \lambda \hat{s}_{i,DNN}(n) + (1-\lambda)\hat{s}_{i,LSTM}(n)$ to get the final result. Aswell, some other algorithms attempt to solve multichannel separation by adding spatial covariance matrices information into it's deep neural network model in order to obtain consistent results, as for example does the NUG algorithm (named after Aditya Arie Nugraha) [29], also based on wiener filtered masks. We can see how the general state-of-the-art models are mostly based in deep learning, although it's still normal combine it with more classical data transformations methods, and using masks in order to obtain better performing results.

5 METHODOLOGY

We have explained in the previous point what is the baseline model we will work with, and we will try to improve the results as well as well as updating it into a framework that is currently supported, unlike the one it's built in right now, and with the possibility of being tweaked in a future date. This section will be divided in various points, and will be expanded in the same order of development for this thesis. It will start by explaining the new dataset that will be used for the new framework; the network will be rebuilt so it can be adjusted this new dataset plus porting it to PyTorch, which is a potent framework for deep-learning related tasks, and this step is needed since the old framework where *DeepConvSep* was build is no longer supported; the robustness of the framework will be later attempted to be increased by performing data augmentation on the dataset; and finally a new denoising framework will be proposed to be used in combination with the baseline model, for each one of the source outputs, to give a cleaner and more usable result.

5.1 Dataset

In this thesis we have chosen to use the MUSDB dataset [35] to be implemented with the framework. This database is an extension to the previous DSD100 dataset [24], and this new dataset is composed of 150 tracks, 100 of which will be used for training and 50 will be used for testing. All of the files in this dataset are encoded using the Native Instruments STEMS format (.mp4), which is a multitrack encoding format, that separates the signals in a total of five separated stereo channels: total mixture, voice, drums, bass, and others as the rest of the accompaniment. Each one of the stereo streams is encoded in AAC format, which is a lossy but reliable compression method, with a sampling frequency of 44.1 kHz. The average duration of each track in the dataset is about 236 ± 95 seconds.

One note that it is important to make out of this previous point, is the fact that the files are compressed using a lossy format such as *.mp4* which is equal to *.mp3* for audio-only signals. This is important, since the old model data signals were encoded using a *.wav* file format, which is lossless. What is important to note is that *.mp3* has a variable cutoff frequency in it's high frequencies depending on the bitrate the file was encoded. This means, that when performing the STFT and representing it on a spectrogram, the high elements on the frequency spectrum won't appear as they are not present in the *.mp3* encoding, therefore representing a loss of information that otherwise we wouldn't have with a lossless format. We solved this by applying a linear transformation on the data in each STFT feeded into the network, instead of just taking it and feeding it to the network, to kinda force it to focus on the lower parts of the frequency spectrogram we applied a weighted linear data transformation along the frequency bins from the 0-th index to the 513-th index, corresponding to the frequency bins of the STFT, with this weighted value going from 1 in the lowest index to 0.5 in the highest value.

The same way as in the *DeepConvSep* model, we will have to treat the data given to us in order to obtain the inputs of the model. As we have explained in the state-of-the-art we will be using STFT's as an input to compute the features of it as masks and then output the final four output signals. STFT's were computed stereo-wise, which means there is an STFT for each channel, while in *DeepConvSep* we only used one channel (see [*Monoaural audio source* *separation using deep convolutional neural networks*] [6]). Therefore, one of the improvements we wanted to make on the old model was to update it to be able to use two channels instead of one, we will later enter in details when explaining the new framework. As we are working with encoded audio signals, we use the discrete-time Short Time Fourier Transform (STFT) to perform our computations:

$$X(m, \omega) = \sum_{n=-\infty}^{\infty} x[n] \cdot w[m-n] \cdot e^{-j\omega n}$$

We used a hopsize of 256 units (75% overlap), each computed with a size of 1024 units, therefore due to the symmetric property of the FFT we will only need to keep the positive side of this magnitude plus the central zero bin, therefore having 513 units to represent the magnitude. Finally we have a known sampling frequency given by the dataset of 44.1 kHz.

The window chosen to perform the computation was a Hanning window with a size of 1024 units:

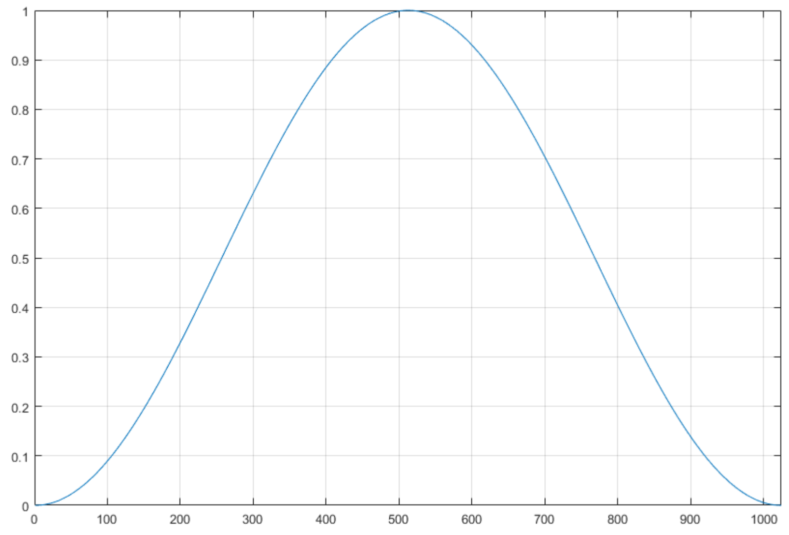


Fig. 10: Plot of a Hanning window with a size N = 1024 in the horizontal axis. The vertical axis is unit-less and goes from zero to one, and applies its value as a weighted transformation for each bin it multiplies.

The stereo STFT's were computed on separately on each of the sources inputs and the mixture, and saved in a $(k \times N \times L \times 513)$ where k is the track identifying index, N are the number of channels, L the length of the file, and 513 represents the frequency bins on each index of time. The dataset is structured as such: for the inputs, which is the mixture, one dataset was made, with an $N = 2$ (the number of channels corresponding on the stereo STFT), and the targets on a separate dataset with an $N = 8$ as four sources with 2 channels give out 8 STFT's.

It's also important to say that the data wasn't normalized in this step (we started working with the network with non-normalized data) but it was later solved at the step of feeding data into the network, by first finding the maximum and minimum values across the dataset for each frequency index, separately for the mixture, vocals, drums, bass and others, and then dividing it by it's corresponding STFT data chunk.

5.2 Framework

a) Network design and architecture

This part's objective is to implement the architecture proposed in [6] in a net deep learning framework. First of all, the reason we need to do this, it's because the original framework was developed in Theano, which became discontinued back in 2017. As a new framework, we have chosen PyTorch, which is a fairly new release and the evolution of the old Torch framework. Choosing a deep learning framework over another usually isn't a complicated decision because it doesn't matter that much: Tensorflow could have been chosen, which is a more mature and used framework, or Keras, to do the our application, and the result would have been the same, as these frameworks can be think of as a deep learning set of tools to do deep learning related applications. Despite of this, PyTorch has all of the benefits of other much more used frameworks as Tensorflow, for example, PyTorch data is also tensor-based (tensors are data structures with many usable methods and transformations) and that means it can accelerate data computations by also processing tensor-data computations in a GPU, by using CUDA as an interface between the framework and the physical GPU (or GPU's). As an advantage compared to other frameworks, PyTorch creates it's networks dynamically, that means they are not pre-computed and network structures to compute the gradient are created on-the-fly [32]. This is an especially big advantage in this thesis, as dynamic neural networks ease a lot the task of coding reliable recurrent neural networks (RNN), which are kind of the norm for audio-related applications, and we will also later propose for later usage.

The building of neural networks in PyTorch is pretty straight-foward: first the network base architectures are built as classes that will later contain network structures by inheriting from a base network class, containing all network elements and methods; also iallows more network modules to be nested inside, which is a feature we are using since our architecture has various elements to it that need to be used sequentially and independently from other elements. Aswell a forward method needs to be defined, in which the forward step of the network will be created and computed, and needs to be in the order the data is forwarded from the input to the output. As we have said before, PyTorch will dynamically create the network from this forward step, so the backpropagation step is inherently calculated in this function and doesn't need to be further implemented, as it is automatically generated. We also used sequential containers to further nest the functions in order to make them more understandable, as for example adding in a container each linear transfer function with it's activation function, or adding together the two convolutional filters for the input; this is both useful for arranging the network in an understandable manner, but it also helps to later easily make changes to each element of the network separately, for example if we later want to use different neural layers or activation functions.

As commented before, one of the improvements made in this network is that it allows stereo signals to be processed in the input as such and generated in the output. We consider it is a big improvement as stereophonic signals of course have more information than a single-channel track, plus, as in stereo recordings the sources are separated into the two channels, we are adding a time-spatial information in the network (for example, a consequence of that are Interaural Level Differences (ILD) [3] there are also Interaural Time Differences (ITD), but in recordings made in studio as the ones in our database, we can discard them), which could further increase the number of features obtained and could favorably impact in the final result, as we also assume there are present audio correlations between the two channels that can be used to further improve the performance of the previous algorithm. (The previous point also means that the current network does not allow monophonic signals as an input. Although it could easily be changed, the network would need to be trained again, but since most signals are stereophonic nowadays, we don't see the need in having another model with that configuration). The old network with our new stereophonic architecture can be pictured with the following sketch:

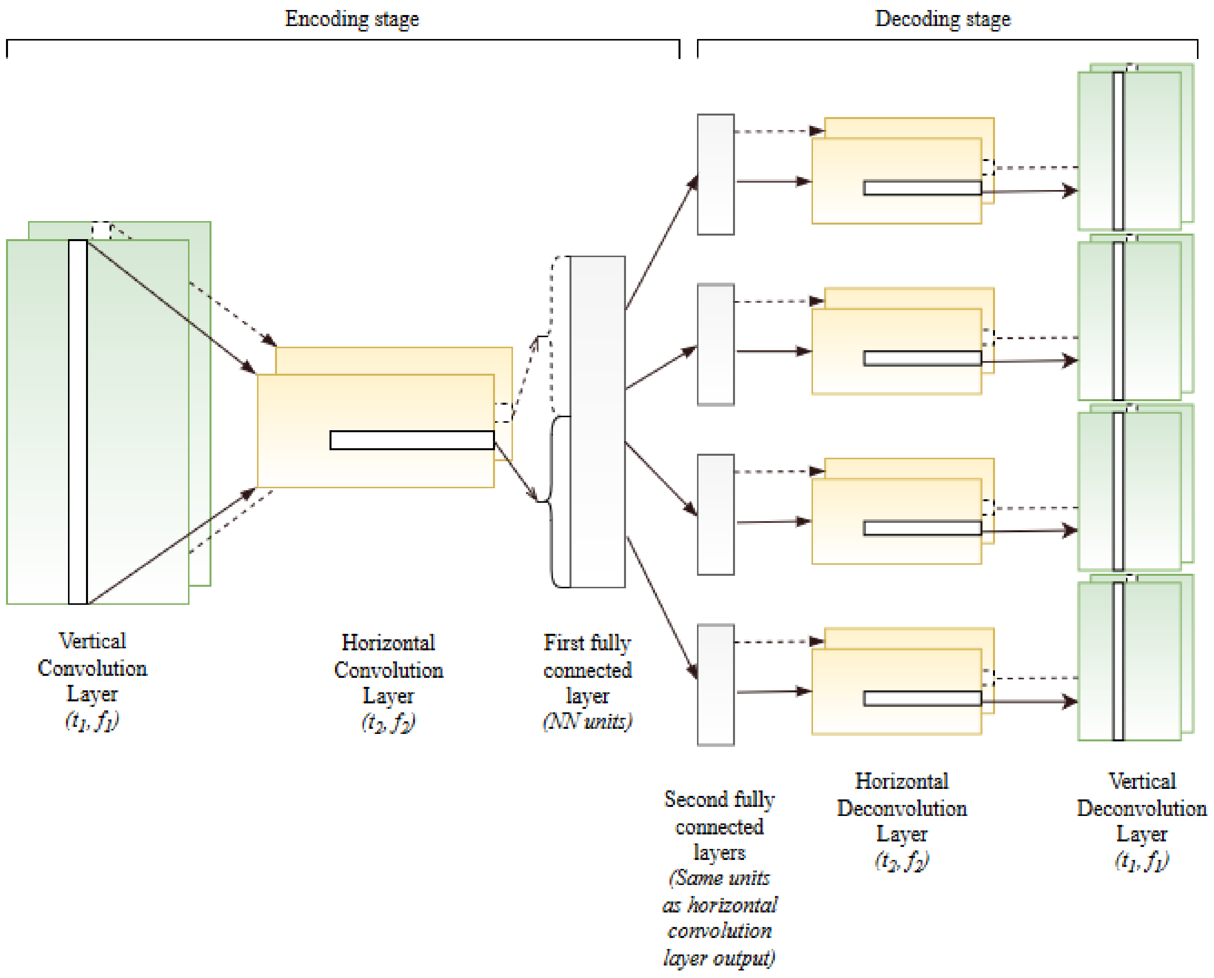

Encoding stage
Decoding stage
Vertical Convolution Layer (t_1, f_1)
Horizontal Convolution Layer (t_2, f_2)
First fully connected layer (NN units)
Second fully connected layers (Same units as horizontal convolution layer output)
Horizontal Deconvolution Layer (t_2, f_2)
Vertical Deconvolution Layer (t_1, f_1)

Fig. 11: The new architecture, which takes stereo inputs and processes through the net. We see how the fully connected layers take both channels together as a single input, and give a smaller number of outputs, which then are reshaped to the same number outputs previous to the fully connected layers once we are on each source decoding stage.

Note: the pair of channels that are pictured on each convolution and deconvolution stage don't represent the number of channels generated in each stage, it's simply used to picture that the stereophonicity of the input data is kept and passed through the whole network.

As we can see, the core structure of the network it's the same as in the baseline model, but this time we have a doubled number of input channels and it's information is forwarded through all of the convolving/deconvolving layers, representing the stereophonic STFT's. The first vertical convolutional layer outputs 50 different filtered outputs (by 50 different filters), and the next horizontal convolutional layer generated 30 filters from the previous inputs. These number of filters are directly taken from the baseline model, as the first step was to replicate the exact same network but with stereophonic data. The first fully connected layer output will have an input with a size of $N \times M \times K$ being $N \times M$ the size of the convoluted output and K the number of output channels, it's output has a total number of 128 units, a smaller number in comparison that helps this layer act as a bottleneck with the benefits mentioned earlier in the baseline model; finally, the second fully connected layers take these 128 units as inputs and transform them through the same number of units as the number of features the first Horizontal Convolution layer gave as outputs. Now with the same size as we had before the first fully connected layer, we can reconstruct the STFT's by performing the deconvolution with the same filter's shapes, but in the reverse order that we used in the encoding stage, for each source individually.

In PyTorch, the dimensionality of an input sent to a convolutional layer takes the dimensionality of $(N \times C_{in} \times H_{in} \times W_{in})$ and gives an output of $(N \times C_{out} \times H_{out} \times W_{out})$, and as we see, the only number that remains being the same is the N, which is the batch size. Therefore, we can use an input for the encoder with a dimensionality of $(N \times 2 \times X_{in} \times 513)$ where N is the batch size, the 2 channels are the mixture stereo channels, X_{in} is the number of bins in the time context in the original STFT (remember that STFT's had a length of L , therefore $x = \{x_n, x_{n+1}, x_{n+2}...x_N\}$ for $0 \leq n < N \leq L$ and therefore $x \in L$, and the 513 bins are the frequency bins. We used a default size for the input of $(N \times 2 \times 30 \times 513)$, with a variable batch size N. The output channel size C_{out} is 50 for the first vertical convolution and 30 for the second horizontal convolution.

As it was seen at the baseline model explanation, the encoding stage has two different convolutional layers, one horizontal and one vertical. The first one is the vertical convolutional layer and it's objective is to take features from the frequency context of the STFT, the receptive field we are using has a dimension of $(1, 513)$, therefore it takes all of the frequency context on each bin of time. If the input of this layer was $(N \times 2 \times 30 \times 513)$, the output would be $(N \times 50 \times 30 \times 1)$. This serves as an input to the next layer, which is the horizontal convolutional layer; we use a receptive field with half the size of the time context, so $(15, 1)$ in this case. This time it doesn't take the whole length of the STFT, and the final encoding output will have a size of $(N \times 30 \times 16 \times 1)$. (the 16 value comes from the operation (N - F + 1) where $N = 30$, the dimensionality of the axis we are encoding, $F = 15$ as the filter length in that coordinate. We don't take into account the frequency axis as the filter shape there only takes one bin and will remain the same.)

Up to this point, the network modules explored previously correspond to the encoding stage of our Autoencoder, and as we have explained before, we now have a compressed output acting as a representation of the data patterns in both time and frequency space of the mixture, plus they are passed through a small neural layer. This layer will take the previous outputs, which we have demonstrated have a size of $(N \times 2 \times 16 \times 1)$ on each channel, so the effective size of the input by the number of channels 30 in the vertical convolutional layer will be a total of $2 \times 16 \times 30 = 960$; the N is not multiplied as the linear layer also work with batch sizes, so the dimension N still remains in the input; aswell, as mentioned earlier, the number of units in this layer is 128 and it's the number of outputs it will give aswell. The next stage now is the decoding step of this data to obtain four representations of the same dimensions as the first input corresponding to each source. The network is designed in such a way that the decoding stage it's independent between the four sources, so the next steps will be repeated four times, one for each source:

The first step in the decoding stage, it's another neural layer. Each source will have it's own layer, with the objective of letting each source choose from the previous output what are the features it wants to keep and are more relevant to compute it's output. The size of this network will take the 128 units of the previous layer, and pass them through its 960 units and give its outputs, the same number that the last convolving layer had. This must be made to respect the same dimensionality for the original input and the final network output, because if we want this condition to be true, in the decoding stage we must process the same data with the reverse order but the same window sizes as in the encoding stage.

Finally we will change the shape of this layer once again to the previous dimension of $(N \times 30 \times 16 \times 1)$ and process it first to the horizontal deconvolving layer, which will be use a window of $(15, 1)$ and give an output of size $(N \times 50 \times 30 \times 1)$, by generating 50 different filters; and finally passing this output to the vertical deconvolving layer, which will give an output with the same size as the original stereophonic input $\hat{y}_n = (N \times 2 \times 30 \times 513)$ for $n = \{voice, bass, drums, other\}$ after using the same window $(1, 513)$. When stacking all of the outputs in the last layer by it's channel dimension, we will obtain a multidimensional tensor of size $(N \times 8 \times 30 \times 513)$, where the only value that changed is the number of channels, now being 8, two for each source.

Now that the network has forwarded the data and generated outputs, it is needed to compute the total error given by the network by comparing them to the ideal output saved in the dataset. But first, as explained in the baseline model, we will use the soft-masks of each source as a medium to compute the error of each output. So for each source n, and having its network output $\hat{y}_n$, the soft mask is computed by doing:

$$Mask_n(t, f) = \frac{|\hat{y}_n(t, f)|}{\sum_{m=1}^{N} |\hat{y}_m(t, f)|}$$

Where $\sum_{m=1}^{N} |\hat{y}_m(f)|$ is the sum of all of the sources, included the one being computed at the moment. This is done by all of the sources except for the "other" source, whose mask is computed by doing:

$$Mask_{others}(t, f) = 1 - Mask_{drums}(t, f) - Mask_{bass}(t, f) - Mask_{voice}(t, f)$$

We do this because that way, we make sure we take the total STFT's bins are distributed across in the total sum of the masks. This is done to smooth the results of performing a time-frequency masking technique, as proposed in [15]. Once each mask for the sources is obtained, we multiply the original input of the network that generated these masks by each source mask, in order to obtain the final output for each source. This is applied as a simple product:

$$\tilde{y}_{source}(t, f) = Mask_{source}(t, f) \cdot x(t, f)$$

Where $x(t, f)$ represents the original input, and $\tilde{y}_{source}(t, f)$ is the final output for each source.

The loss between the time-frequency masked output and the ideal output for each source is later computed. We used the SE function to calculate this loss, which is done by applying the function:

$$E_{source} = |\tilde{y}_{source} - y_{source}|^2$$

Then we apply the adjustments in the total error cost estimate proposed by [4] and explained in (section 4.c).

b) Data pipeline

With large datasets such as the one we are working with, not because of the amount of elements but the amount of data each element needs to be represented, we need to create a data pipeline as efficient as possible, and that takes the minimum resources and time possible to feed large amounts of data into the network. Although python it's not the fastest data processing language, it can give good results by applying some optimizations such as generators and binary data-treating libraries to save and process the data. We have calculated the performance of the data pipeline, for each batch iteration, and different number of files per batch:

Table 1: Performance of data pipeline batches (average of 100 iterations)

Batch size	Elapsed time (s)
1	0.047
2	0.079
5	0.323
10	0.697
15	1.014

As we can see, the table results are pretty much linear, across all of the batch sizes. There is also an overhead, which follows a non linear distribution, which is caused by the normalization step we are adding in this point. At batch size 15, this overhead is around 0.03 seconds, which is a relatively low number, compared to the whole second we takes to generate the data. As we explained before, we didn't normalize the data in the STFT, so the best point to normalize it before feeding it to the network it's on the data pipeline. The normalization of the data is made by finding across the whole STFT dataset, which are the values for each frequency bin of the STFT, therefore, needing the minimum and maximum values of the whole dataset in each frequency bin of the STFT, and for each source, including the mixture:

$$max(f) = max_f \left(\sum_{k=1}^{K} x_k(t, f) \right)$$

$$min(f) = max_f \left(\sum_{k=1}^{K} x_k(t, f) \right)$$

Which both will give a vector of 513 elements, and then performing the following operation, for any STFT, before sending it to the network:

$$\|STFT_n(t, f)\| = \frac{STFT_n(t, f) - min(f)}{max(f) - min(f)}$$

5.3 Data Augmentation

As the theory dictates, any neural network can get to be as good as the data we are feeding into it in the training step. In cases similar to ours, where we have a lot of features (thousands of STFT's) but very few data points (dozens of audio tracks, a small number in comparison), we get to a relatively soon point in training where our network won't be able to learn anymore from the data, which is not a bad thing (considering we are not overfitting our model), because we want to get to that point, but still we can improve on that by creating new data with the tools we have in order to obtain an even more reliable model.

As seen for example in image processing, where it's common to apply transformations on images when training a model, such as applying rotations, change of dimensions, or most kinds of affine transformations, in audio processing we can also perform transformations to generate new coherent and realistic data, which helps training a model. We can see examples in speech processing [18] or audio classification [38]. To keep it simple, in this thesis we will explore two solutions that can help to create new data without causing an excessive overhead on the model.

First of all, as we commented before, we can say that the data is divided in two different datasets, both of which have a shape of $(N_{track} \times N_{channels} \times t_{bins} \times f_{resolution})$, where t_{bins} is the length of the file and $f_{resolution}$ is default to 513 frequency bins. Then, one for the datasets contains inputs of the model, which has a size of $(N_{track} \times 2 \times t_{bins} \times 513)$, and for the same N_{track} will have a counterpart in the targets dataset, with a size of $(N_{track} \times 8 \times t_{bins} \times 513)$ (remember the 8 channels for 4 sources in stereo).

The first approach on the model, it's to generate new samples not by adding, but by taking out information from the dataset. That is, to sometimes, send as an input to the model not a whole mixture, but a mixture with some of the final sources of the mixture taken out. This can be particularly beneficial for the sources that are hard to train, for example the voice, because it shares a lot of formants with the accompaniment when they overlap. The way we thought about doing this, is by taking a mixture, and randomize what sources we take out, for example, being the mixture $\hat{m}$ and the sources being $\hat{y}_n$, and we take out a set of sources $y_r = \{\hat{y}_m, \hat{y}_{m+1}, ...\hat{y}_M\}$ with $M < 4$ out from the mixture:

$$\hat{m}(f,t)' = \hat{m}(f,t) - \sum_{m=1}^{M} \hat{y}_m(f,t)$$

And the targets where $N_{channels}$ are in y_r are replaced with zeros. In this case, we choose randomly the number of sources that will be eliminated as long as the final mixture has at least one source, and it is not the 'others' source.

The second approach is to take random samples from mixtures and add them together in a new mixture. Once we have selected a set of samples, we generate the new mixture by performing the following addition:

$$\hat{m}(f,t)' = \frac{\sum_{m=1}^{M} \hat{y}_m(f,t)}{\sum_{m=1}^{M} max((\hat{y}_m(f,t)) - min(\hat{y}_m(f,t)))}$$

Finally another approaches that could be explored and are believed to be useful and provide good results could be pitch or amplitude changes, but controlled in a way that some meaning is created in the data generated. This is a bit tricky to make it work well in the dataset, so it wasn't implemented in this thesis.

Where we sum the STFT of the signals and divide it by the sum of the normalization for each source (instead of just dividing it by four). We created an algorithm that randomizes two parameters: first which ones of the four sources will be randomized by choosing a separate track for those, and then randomizing which out of the sources will have their time index randomized.

Aswell, we introduced a probability in the data pipeline so each time it loops to create a new batch, it doesn't always perform data augmentation, aswell as choosing one of the two methods.

5.4 Proposed denoising framework

As we explained before (Section 4.1), our state-of-the-art network has a problem in the yielded outputs, as there is a strong presence of flickering noise in most of the duration of the synthesized outputs created from the network. As the network itself, when disregarding this problem, still gives acceptable outputs approximating each source, plus this is not an architecture related problem but a model related problem derived from the usage of masks to create the outputs, we won't add the denoising framework to the existing source separating network. As well, we won't do it because when creating increasingly complex networks, training this networks with gradient based techniques, which are the ones we are using, becomes increasingly difficult. Therefore, we decided that the best solution was to add a denoising framework outside the baseline model architecture that takes as an inputs the generated and -noisy- separated sources from the network, and sends them to the denoising framework to be cleaned. Additionally, we have created one denoising framework for each source, which should further improve the independence of results and learning more features related to each source. We can picture the whole framework in the following sketch:

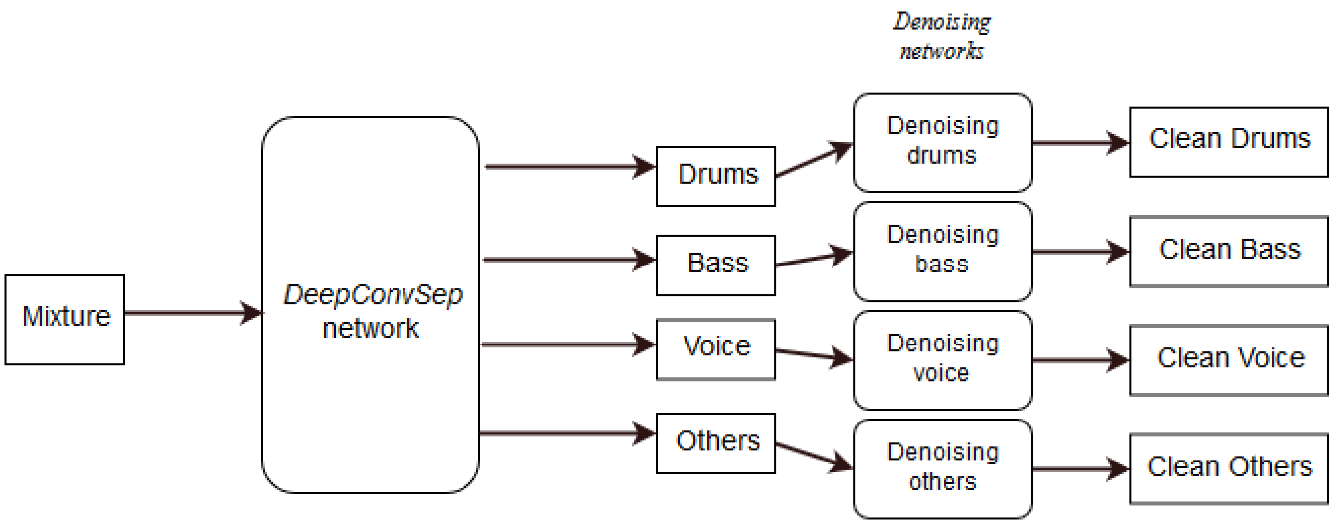


Fig. 12: The proposed framework, denoted as *Denoising networks*, and how it relates to the baseline network.

A deep learning approach was chosen to denoise the forenamed sources, as we explored back when explaining the Autoencoders and it's uses (Section 3.5), we saw how one of their most used applications was to perform data denoising, in our case these data inputs will be STFT's. We will therefore use an autoencoder to perform our denoising of the data, similar to the one in the baseline model, but this time we will only encode the frequency context of the input (which was the horizontal filtering layer in the base model).

This is due that we will change the way the fully connected layer it's implemented in the base model, and we will instead use a bidirectional LSTM (Long short-term memory)(Section 3.6) unit as a bottleneck between the encoding and decoding stage of the network, which as explained in the state-of-the-art has been used with some good success. We do this because we believe that it's important to take into account the time-context of the signals and can help us to get better representations of the data, as at a time context t where a set of frequencies and formants are present, it's very probable that contains information from the time contexts $t - n$ for $n = 1, 2, 3...$, or an undetermined sequence of prior inputs.

The network architecture we propose it's the following:

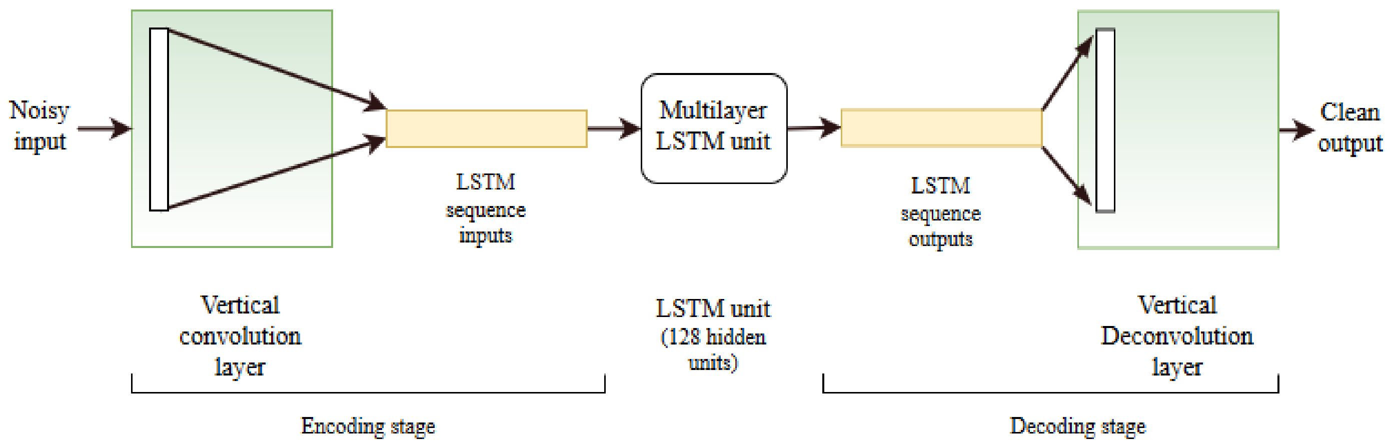


Fig. 13: The proposed denoising architecture, which is an Autoencoder that has a LSTM layer instead of a fully connected layer.

Now we can see why we only used the vertical convolution, as a LSTM layer takes a series of sequences which are related between them with an ordinated manner, for example in NLP they would be sequences of words in a sentence or letters in words; in our case, are compressed frequencies from an STFT and the sequence is related between the inputs by their time bins. The inputs will have the same size as the DeepConvSep network, with $(N \times C_{in} \times H_{in} \times W_{in})$ = $(N \times 2 \times 30 \times 513)$, where N is the batch size. Another difference this autoencoder has, it's that the vertical filter receptive field doesn't take the whole 513 bins of the STFT, but we will reduce these 513 bins to 128 instead of just one as we had in *DeepConvSep*, therefore the receptive field will have a size of $N \times M$ with $N = 1$ and $M = 513 - 128 + 1 = 386$, so a receptive field with a size of 1×386. The LSTM sequence inputs will then have a size of $(N \times 2 \times 30 \times 128)$.

In the Recurrent Neural Networks explanation (Section 3.6) we also included an extended explanation on how LTSM process an input and generate outputs by applying various functions that create an abstraction on how memory works. The only thing we need to comment here that wasn't commented before, is that the outputs of the LSTM unit have always the same size as the LSTM inputs, therefore being $(N \times 2 \times 30 \times 128)$ the size of the LSTM sequence outputs. We then deconvolve these outputs by the same window as before 1×386 and obtain a denoised output with the same size as the input.

We will therefore train four of those networks in parallel, to perform denoising on all of the four separated sources with separated networks. The training data pipeline it's similar to the one from *DeepConvSep*, but instead of taking the data directly from the dataset, we will first send this data to a pre-trained *DeepConvSep* model, and then take it's output as the noisy input for this network. We will then compare it to the ideal output for the source to force the network to learn denoising patterns.

Although the

5.5 BSS evaluation tools

To qualify and quantify the quality of our source separation algorithm, we need a series of measures that are able to extract information from our separated sources from model output, and be able to compute a quantifiable and reasonable measure that relates it to what it should be it's ideal output. In that regard, these performance measures have long been discussed and have been settled by extendedly using the BSS toolkit, proposed by E. Vincent, in *Performance measurement in blind audio source separation* [45]. We will use three of the proposed measures to analyze the algorithm performance, but first, let's define the difference between an estimated source $\hat{s}_j$ and the target source s_j as:

$$D = min_{\varepsilon=\pm1} \left\| \frac{\hat{s}_j}{||\hat{s}_j||} - \varepsilon \cdot \frac{s_j}{||s_j||} \right\|^2$$

Where the squared difference of each signal divided by it's module will approach zero the closer they are, until they converge at 0 in the ideal case where $\hat{s}_j = s_j$; otherwise the difference between the two sources increases until in the worse case scenario when it reaches 2 (as the sources have been normalized). The problem this measure has is that in the case of small correlated noise in the signal $\hat{s}_j$, when we apply this distance function, the error still it's too close to zero, despite the signals being significantly different, which in some cases it's to be considered, for example, if we want to further process the signal, we can accept this error. Therefore, the E. Vincent proposes in [45] three new measures, by treating the estimated source $\hat{s}_j$ as the following decomposition:

$$\hat{s}_j = s_{target} + e_{interf} + e_{spat} + e_{artif}$$

Where s_{target} is a modified version of s_j where certain distortions are over it and are allowed, the element e_{interf} represents the interferences of other sources j' where $s_{j' \neq j}$, e_{spat} are interferences caused by the spatial distortion of the recording, and finally e_{artif} are artifacts that are not allowed in s_{target}. The three measures derived from this sum can be written as:

$$SDR = 10 \cdot log_{10} \left(\frac{||s_{target}||^2}{||e_{interf} + e_{spat} + e_{artif}||^2} \right)$$

$$SIR = 10 \cdot log_{10} \left(\frac{||s_{target}||^2}{||e_{interf}||^2} \right)$$

$$SIR = 10 \cdot log_{10} \left(\frac{||s_{target} + e_{interf} + e_{spat}||^2}{||e_{artif}||^2} \right)$$

Where SDR measures the Source-to-Distortion Ratio, SIR is the Source-to-Interferences Ratio, and finally SIR is the Source-to-Artifacts Ratio.

5.6 Code implementation

The whole open code implementation can be found at the following link: *https://github.com/joangro/PytorchConvSep*

6 EVALUATION

To perform the analysis on the performance of our new framework, a single model was trained to do it's analysis. We empirically choosed the following parameters after training various models by changing it's hyperparameters, and ended up choosing the ones that fitted the problem the best. As a note, we will use previous information taken from the baseline implementation paper [4], where it lets us know that the batches parameters aren't relevant to the final network output, although we will still are mentioning them.

6.1 Standard network evaluation

It was decided that this point would be the main focus of this thesis, as the change from the previous monaural model to this stereophonic model should be a big enough change and upgrade to evaluate it as a new model. Therefore most of the focus of the evaluation is centered around this point.

The created model parameters can be found on the following table:

Table 2: Model parameters used to test the new framework

N. epoch	*Learning rate*	*Batch size*	*Batch/epoch*	*Time context*	*Data augmentation*
3250	-	5	50	30	No

The model was trained using *Adadelta* as a gradient descent optimizer [50]. It's benefits over other methods is that it doesn't need a manually input learning data, as it dynamically adapts it over time, and also it seems to remain robust enough when treating with noisy data, which is the case we have on our hands. Still, the optimization function we use it doesn't have too much of an impact to the final outcome, although they work differently and apply different optimizations to the gradient, the result it's close to the same; some other optimizers were tested, but it's results were harder to get, as the parameters needed more tweaking, and it was decided to just use Adadelta as it was already giving good results. There is also further improvements that could be explored, such as adding L1 and L2 regularization on the gradient, that even though this algorithms have it natively implemented, it probably could be improved to work better in our specific task.

As a general rule, training these models was proven to be a very delicate process, as we run into constant problems regarding the gradient computation, such as vanishing gradients or exploding gradients. We did end up solving this problems by realizing there was a mistake in the dataset: unlike with the old baseline model, this time the data was treated directly and not processed by an algorithm that checked for silences present in the original mixture and removed them. To solve this, we added a small algorithm in the data pipeline that checked the average value on each STFT (from the 0-th bin to the 415-th bin, as over this one more or less there is only silences due to the *mp3* encoding), and if it was over a certain threshold we would use this STFT as a non-silent STFT. This threshold fas found empirically by trying different values (the data is normalized, so the values were between 0 and 1) and choosing one that filtered some of its values while not taking too much time finding an appropriate STFT chunk. The value finally used was *0.02*, and is the one that gave good enough values. The appearance of Nans during the training of the network was always at the beginning of a new epoch, and it was because at the start of each new epoch we reset the gradients for all the parameters θ of the network back to zero, and when in this point we did send an empty STFT, the backpropagation of a zero value error forced the network to divide by zero, then giving the Nan values.

To perform the evaluation, we will use the measures in (Section 5.5) and apply them in each of our models plus on the old *DeepConvSep* framework. We will perform the measures over the testing dataset, which contains a total of 50 songs.

Before the evaluation, we performed a perceptual evaluation to see if the generated STFT's for each source were correct, because the times the network was overtraining the model, STFT's had a very distinctive shape, an example depicting it can be found on the following figure:

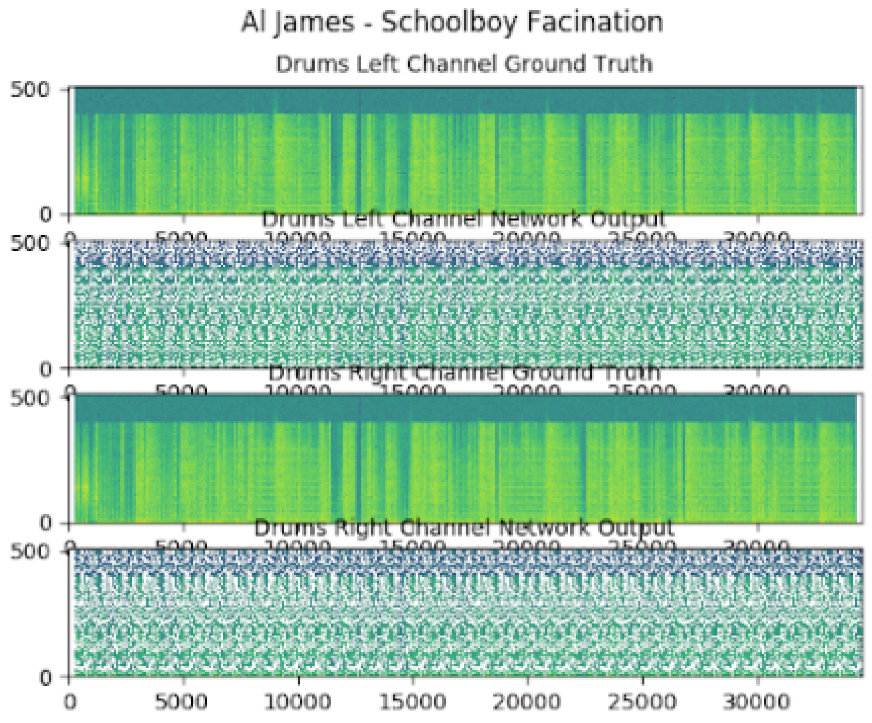


Fig. 14: Example of an overtrained model, derived from the gradient computation in the training step, depicting the drums targets and it's separated STFT's, for each stereophonic channel. The first and third STFT's are the targets, whereas the second and fourth are the separation approximation. We see how the estimated STFT's have a very distinct shape from the original, with a lot of apparently random white points (where the STFT is 0, therefore silences at that time and frequency) and therefore we see how it has lost all of the information of the original, giving us an unacceptable result with 'random' information.

In the other hand, the following STFT's have been correctly trained and give an acceptable result, these are generated with the model described above:

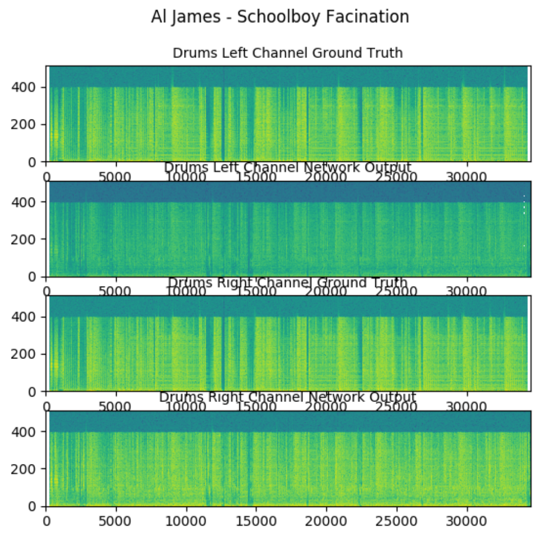


Fig. 15: Drums STFT.

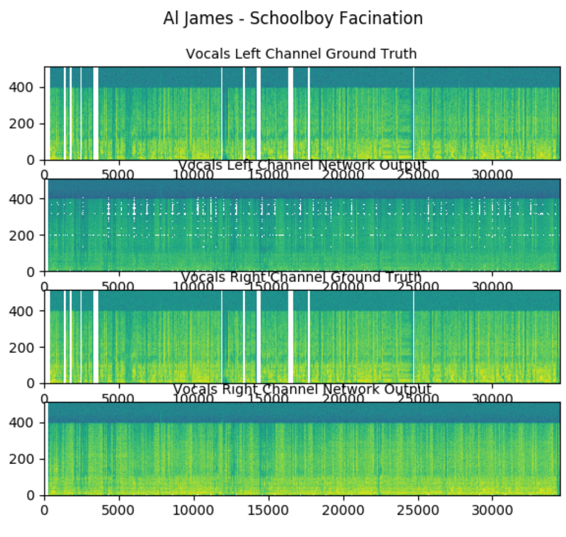


Fig. 16: Voice STFT.

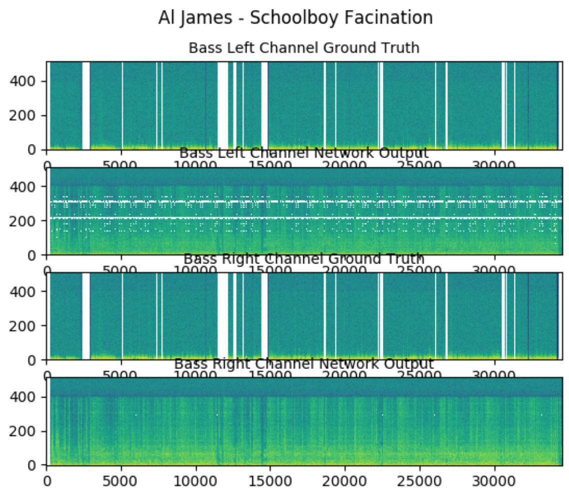


Fig. 17: Bass STFT

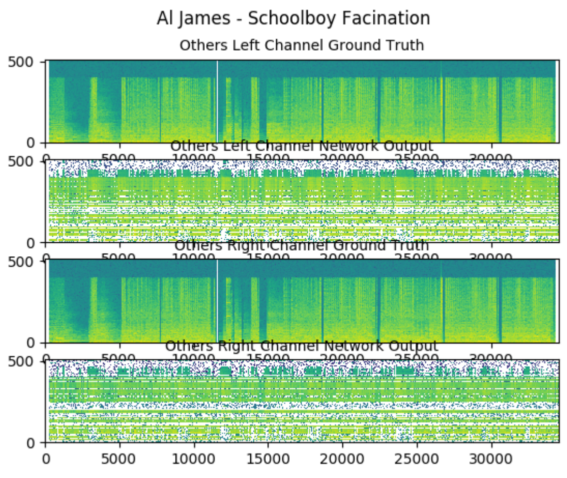


Fig. 18: Others STFT.

Once again the first and third plots on each figure represent the original STFT, and the second and fourth the estimated STFT.

From the previous figures we can extract some observations. First of all, we see how the *Drums*, *Voice* and *Bass* sources, at first glance, are pretty similar to the original STFT's we want to reconstruct. Furthermore, for example, in for example the *drums*, we see how most of the harmonics have been kept, similar to the voice, where most of the harmonics are kept from the original to the reconstructed. The best case though, it's the bass STFT, as we can see how the STFT values have been approximated for the low frequencies, although in this case there is still some noise from the other sources. Finally, in the *others*, we see how the estimated STFT is nowhere near the objective one; we see how most of the frequencies have been eliminated, and therefore the rest of the sources will have these information. Although the result here it's not acceptable, we don't care that much, since the sources that contain most of the song's information are the others, as they are the principal sources, therefore we can and will accept this STFT as an approximated one.

Even though these figures are the best way to exemplify visually the result of the network, they aren't a good way of getting a sense of the performance of the model. The only way of getting a good sense of the result generated is to make a perceptual test of the result, that is, generating the audio samples from the STFT's and checking if the result is actually separating the mixture. It is possible to do so in our code, but only with the test tracks in our database, which are on STEMS format. Aswell, the training error of the network also doesn't necessarily exemplify the training status of the model, for example, in the following graphic we can see it's evolution for the first 500 epochs:

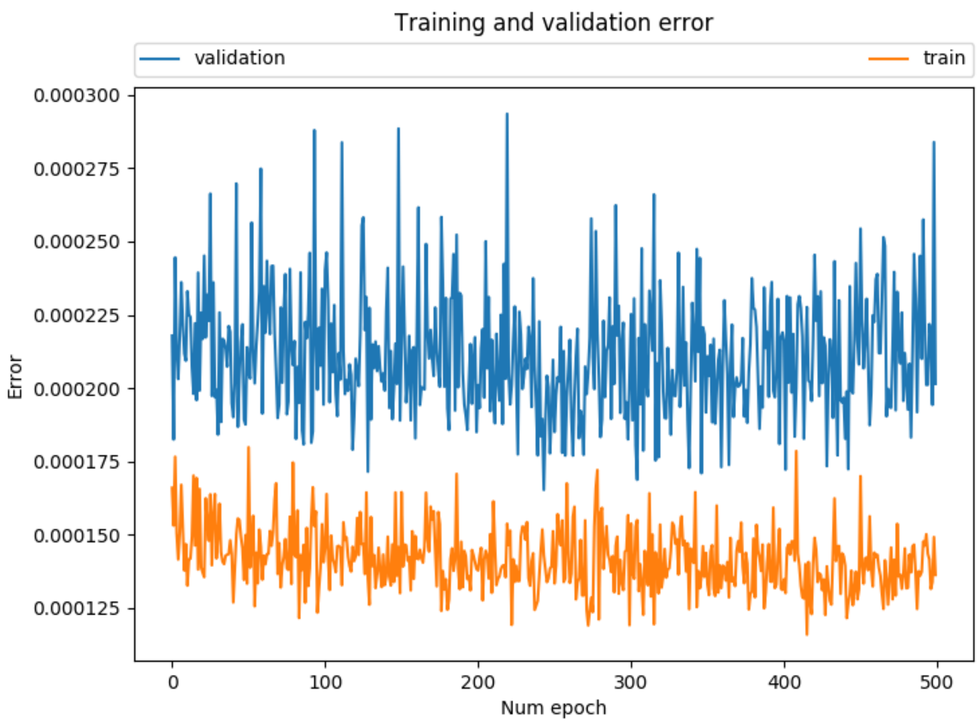


Fig. 19: Evolution of both the training and the validation error for our network on the first 500 epochs. We can see how both error measures hardly go down when increasing the epoch number, while still learning and giving noticeable perceptual differences between epochs.

Another point that we want to make is that here, we see in the original STFT's have some blank spaces corresponding to the source being silent in that time index, which is the problem commented before. Despite so, the estimated STFT always reconstructs the STFT in those time spaces. This is due the fact that we are using a mask to perform the separation, therefore the mask when applied to the mixture, will in some cases take frequency information from other sources that happen to have similar frequencies / harmonics as the source being separated.

Finally, the results were computed by using an evaluation dataset conformed of 50 different songs, what we did is randomly process these songs total number of STFT frames through our trained models, and then send them to the perform the evaluation measures discussed before (Section 5.5), which are already implemented in the python library *mir_eval* [34]. To reduce the computational expense of processing full songs, we randomly select long and therefore musically coherent frames to apply these methods, in this experiment we used a length of 8 seconds for each tested song, which by the sampling frequency we are working with (44.1 kHz) it's roughly 350.800 time units long frames. These results are taken from a sample of 100 different frames and depict the mean ± the variance:

Table 3: Evaluation for our models using standard BSS measures, for *vocals* and *drums*. Measures presented in decibels shown as Mean ± Standard Deviation

Measure	Vocals			Drums		
	CH1	CH2	Mean	CH1	CH2	Mean
SDR	0.76 ± 0.5	1.2 ± 0.9	**0.98 ± 0.7**	1.1 ± 1.6	1.5 ± 1.6	**1.3 ± 1.6**
SIR	2.8 ± 2.4	4.3 ± 3.4	**3.5 ± 2.9**	3.8 ± 1.1	3.8 ± 0.8	**3.8 ± 0.9**
SAR	2.3 ± 1.8	3.7 ±3.2	**3.0 ± 2.5**	7.3 ± 2.1	8.3 ± 2.7	**7.8 ± 2.4**
ISR	1.8 ± 1.4	0.9 ± 0.7	**1.3 ± 1.1**	4.4 ± 1.6	4.7 ± 1.7	**4.6 ± 1.7**

Table 4: Evaluation for our models using standard BSS measures, for *bass* and *others*. Measures presented in decibels shown as Mean ± Standard Deviation

Measure	Bass			Others		
	CH1	CH2	Mean	CH1	CH2	Mean
SDR	2.4 ± 1.9	2.5 ± 1.5	**2.5 ± 1.7**	8.9 ± 4.6	9.5 ± 4.6	**9.2 ± 4.6**
SIR	3.5 ± 1.82	2.2 ± 1.8	**2.85 ± 1.8**	11.1 ± 4.6	11.7 ± 5.3	**11.4 ± 4.9**
SAR	8.4 ± 2.2	8.2 ± 2.3	**8.3 ± 2.3**	8.6 ± 2.8	5.8 ± 3.9	**7.2 ± 3.3**
ISR	5.7 ± 1.8	5.4 ± 1.6	**5.5 ± 1.7**	10.7 ± 3.7	9.0 ± 3.6	**5.3 ± 3.7**

To put these values in context, we are going to compare then to the original baseline model. There are two points to take into account before reading further onto the numbers given in the original baseline model: first, it is trained and tested with a different dataset, so the values, even though probably would be close for this dataset aswell, don't directly reflect the differences, even though they still are indicators on the performance of the model; the reason why it wasn't recomputed with the new dataset it's because this new one is done with STEMS (mp4 files) and the old one with wav's, so the old baseline model should be adapted to this new format, aswell, this change of format could also mean that the old base model could probably also need to be trained with the new dataset. Secondly, there are also other models that are currently working with this new dataset and we could actually directly compare the results; the reason why we can’t do it is because as of June 2018, the results still haven’t been published, which will be part of the MUSDB2018 challenge, but when they are finally published they can be compared to ours. The old results from the baseline model in the old DSD100 dataset are shown on the following table:

Table 5: Old baseline model results for the old DSD100 dataset. Measures presented in decibels shown as Mean ± Standard Deviation

Measure	Vocals	Drums	Bass	Others
SDR	-0.6 ± 4.9	-0.6 ± 1.6	2.6 ± 2.5	4.5 ± 1.2
SIR	1.9 ± 6.2	-0.3 ± 3.6	7.5 ± 4.2	14.9 ± 5.6
SAR	3.6± 2.1	1.3 ± 1.3	11.7 ± 3.2	16.4 ± 2.9
ISR	7.3 ± 2.7	8.2 ± 2.6	10.3 ± 2.5	6.9 ± 1.0

Having now both the results taking the BSS evaluations we kinda can compare the performances of both models. First of all, there is a lot of variance between the measures and the sources between both models, in some cases they perform close, while in others the results are better or worse. Starting with the *vocals* source, we can see in this case how the performance has been similar to the old baseline model: the SDR SIR and SAR scores are pretty similar, but in general the new model has less variability and in that regard can be considered more consistent, there is also a jump in accuracy in the ISR measure, where we now have almost 6 dB less of error. Now onto the drums, this has been by far the worst performing of all of the sources, in all the cases except for ISR the score being worse than the old baseline model, we still don’t know what is generating this worse performance in drums, but probably some correction should be made in the algorithm to ensure drums mask can be learned with more reliability to produce more accurate results. Next onto the bass, in this case happened the opposite as in the drums, in this regard the performance has been much better compared to the old model, where we can highlight a decrease in 4.7 dB in the SIR measure and 3 DB in the SAR error measure, we believe that this is due the old baseline model being too restrictive in some cases with the ’length’ of the bass source, as if we listen to the old model results we can see how most of the bass information is located at the ’attack’ of the instrument. Finally the ’others’ source has also been positively improved, except for the SDR calculation, the rest of the values are smaller than in the original, although this is not a source that is the main focus of the separation, this measure tells us that less information from the rest of the sources is now being kept compared to the old model.

We believe still that this model can be further improved, for example, most of the parameters in the old model were throughly tested and multiple models were trained in order to get to the conclusion of which hyperparameters are the best for the model; in this thesis we did the (naive) assumption that these parameters were the best that we could try and directly implemented them despite our new architecture being slightly different than the original, and only trained one model to test it; we believe that small changes in the architecture could positively impact the result, although it wouldn’t make too much difference, but for example we could add more units and filters since we now are working with stereophonic information, and this extra information may need more elements to be accurately represented, so for example in the case of the drums source, the worst performing, the result could definitely be further improved.

Finally the generated model's box plots can be found in the following representations to give us a better image of the accuracy and variability of the problem:

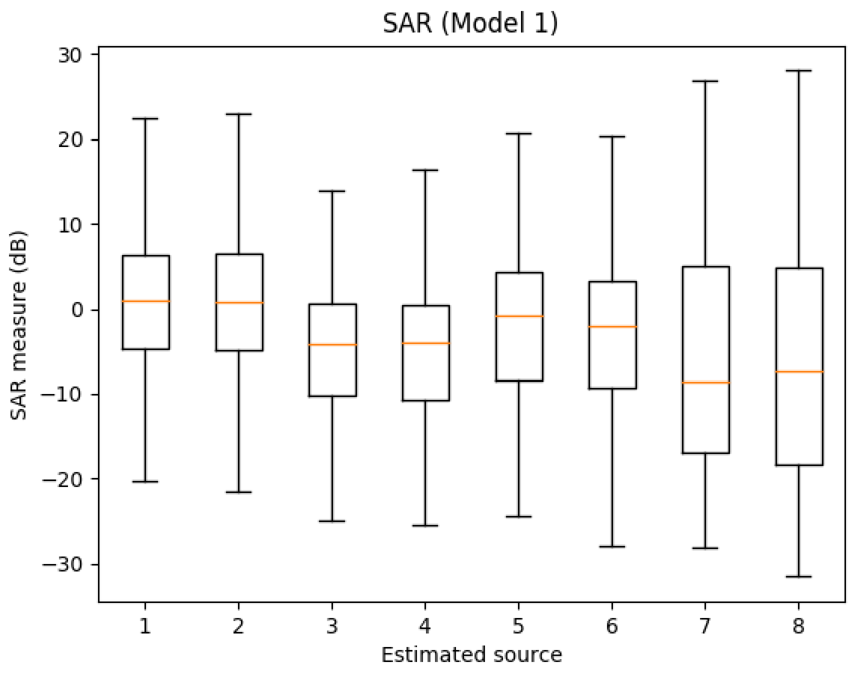


Fig. 20: Box-plot of the SAR measure on model 1, for each source's channels. Source 1-2: *Vocals*; source 3-4: *drums*, source 5-6: *Bass*; source 7-8: *Others*

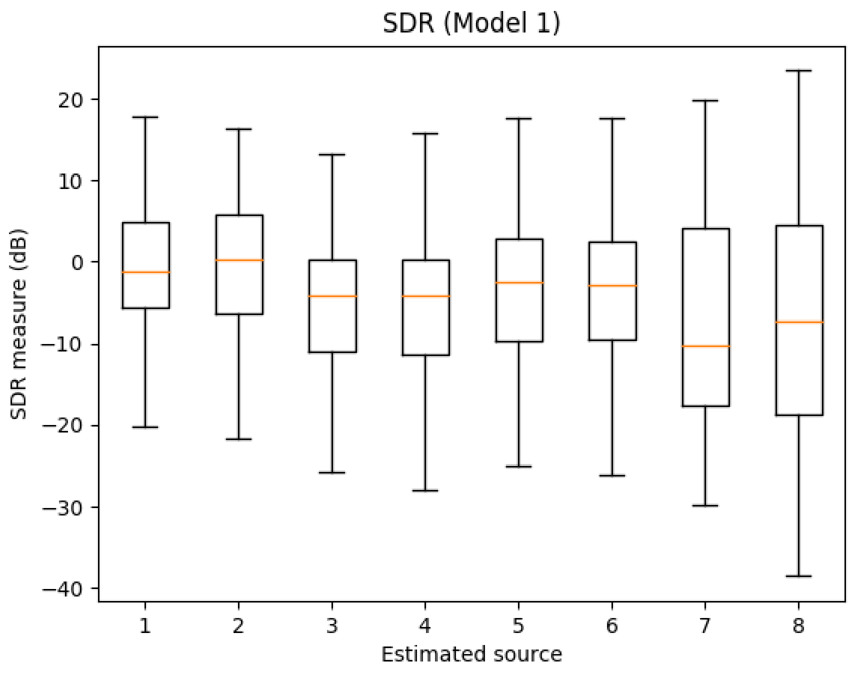


Fig. 21: Box-plot of the SDR measure on model 1, for each source's channels. Source 1-2: *Vocals*; source 3-4: *drums*, source 5-6: *Bass*; source 7-8: *Others*

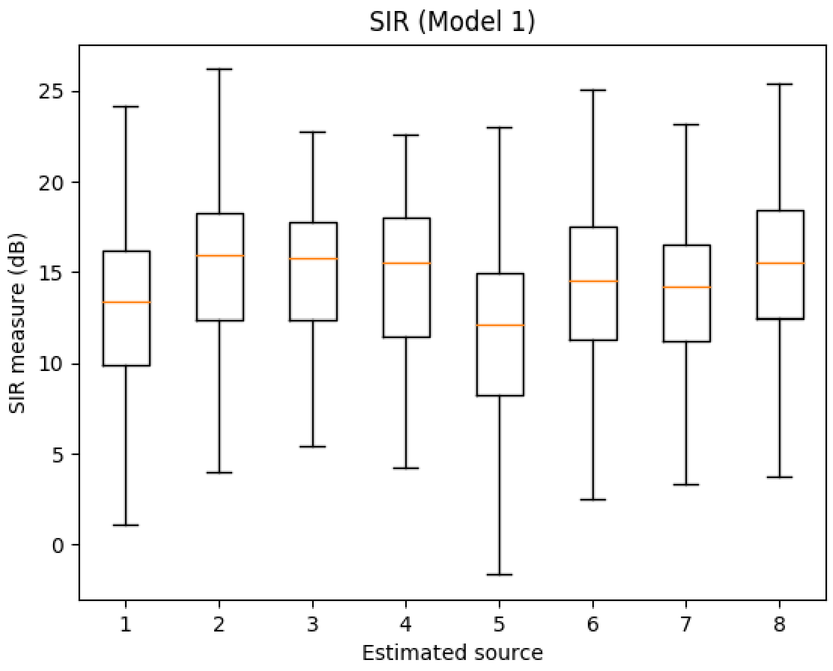


Fig. 22: Box-plot of the SIR measure on model 1, for each source's channels. Source 1-2: *Vocals*; source 3-4: *drums*, source 5-6: *Bass*; source 7-8: *Others*

Remembering the scale of measures explained back in section 5.5, the obtained measures are better the closer they are to zero, in SAR and SDR and we can see how most sources are centered around that mark, excluding the *others* source. The SIR (Source to interference) measure although, it gives a pretty bad score overall, which is relatable to the noise we talked about when introducing some of the state-of-the-art flaws (Section 4.1), which is throughly present in the estimated sources. Aswell, this plots show that the high variance present in the first is probably due the outliers present in the dataset, as we have explained before, in the dataset we are working on there is presence of several musical styles, and it's probable that some of the ones with less appearance it's more poorly estimated than others.

Finally, we choosed *Model 1* to be the default model for our network, to be used for evaluation and the model that will generate the outputs for each separated source, as is the one with better results.

6.2 Data augmentation evaluation

As we explained before, our intent was to implement a data augmentation (DA) algorithm to our network that helped generate more data, in order to train a more reliable and robust model. We choosed to keep it simple, as we also didn't think the methods for DA would generate a considerable improvement to our model, so we did some changes. First of all we implemented an algorithm that puts one of the sources outcomes to silence, by putting all of the values in the STFT to zero and then removing the STFT bins from the mixture STFT.

We then, implemented the second data augmentation technique, which is to generate new mixtures by taking different random songs for each source, and randomizing the time index we take samples from in all of them. We then put them all together to generate a mixture, taking into account that this new mixture needs to be normalized by the maximum values for each one of them, by applying this normalization method. To perform the normalization we need two values, the maximum and the minimum, which are taken across the dataset for each source and across each frequency bin of the STFT, therefore giving us a number of 513 maximum and minimum values on the frequency axis:

$$max_{total}(f, ch) = \sum_{n=1}^{N} max_n(f, ch) \ , \quad min_{total}(f, ch) = \sum_{n=1}^{N} min_n(f, ch)$$

Where N is the total number of sources; f has a length of 513 values, for each frequency bin; and ch represents each channel of the stereophonic mixture. Finally, once we have this total sum, we perform the normalization of the generated mixture as the equation:

$$\|STFT_{mixture}(t, f)\| = \frac{STFT_{mixture}(t, f) - min_{total}(f, ch)}{max_{total}(f, ch) - min_{total}(f, ch)}$$

One of the limiting problems data augmentation has if we are doing it online such as our case, it's that it takes a big amount of resources. For example, taking the table of reference in the data pipeline explanation (Section 5.2.b), where it took around 1 second to generate a batch size of 15 elements, in our data augmentation setup it took an average of 25 seconds, which is too much. The problem that came text was the application of these normalization to the inputs: for a batch size of 5, the computations went up to 125 seconds for each batch epoch. This time was unacceptable, therefore we finally discarded the online data augmentation techniques to be further evaluated, but opened the gates at the possibility to being applied in the future for our framework by formulating the necessary constraints, such as the normalization, to be used if it was to be used, as the steps that need to be taken care of in order to correctly use DA. Simpler DA techniques could still be used, such as for example swapping the stereophonic channels in the sources outcomes, but we don't believe a single DA technique and much less a one so simple would do any difference in the outcome results. Still, the code it's fully implemented, but due to time constraints and focus on the first part of the evaluation process, we decided not to further evaluate the DA in the network.

6.3 Denoising framework evaluation

As we have explained in the methodology, the denoising framework works by denoising the outputs of the source separation framework. We also explained how the aim is to have a denoising framework for each one of the noisy sources, so each source n will have its denoiser $denoising_n$. Since this would mean that we would have to train four networks, maybe in parallel, in this thesis we decided to just implement one and see it's results, and in case they are favorable in the foreseeable future it could be considered to be fully implemented for each source. Therefore we choosed the *voice* source to perform denoising to, since voice it's arguably the most interesting element and the one that can have more applications outside of separating it from music, since trained and applied into a different dataset, this framework should still perform separation from vocals to any source, which it's a good thing for speech processing and enhancement.

Table 6: Model parameters created to test the denoising framework

Source	*N. epoch*	*Learning rate*	*Batch size*	*Batch/epoch*
Vocals	100	0.01	1	300

This model was trained using Stochastic Gradient Descent (SGD), and the learning rate was set to 0.00001 which is similar that the one we used before, even though this time the network it's smaller we need to take care since is a recurrent network, so this is made in order of preventing gradient related problems.

First of all we performed the training and the evaluation of the set at the same time, giving us this evolution through the 100 epochs:

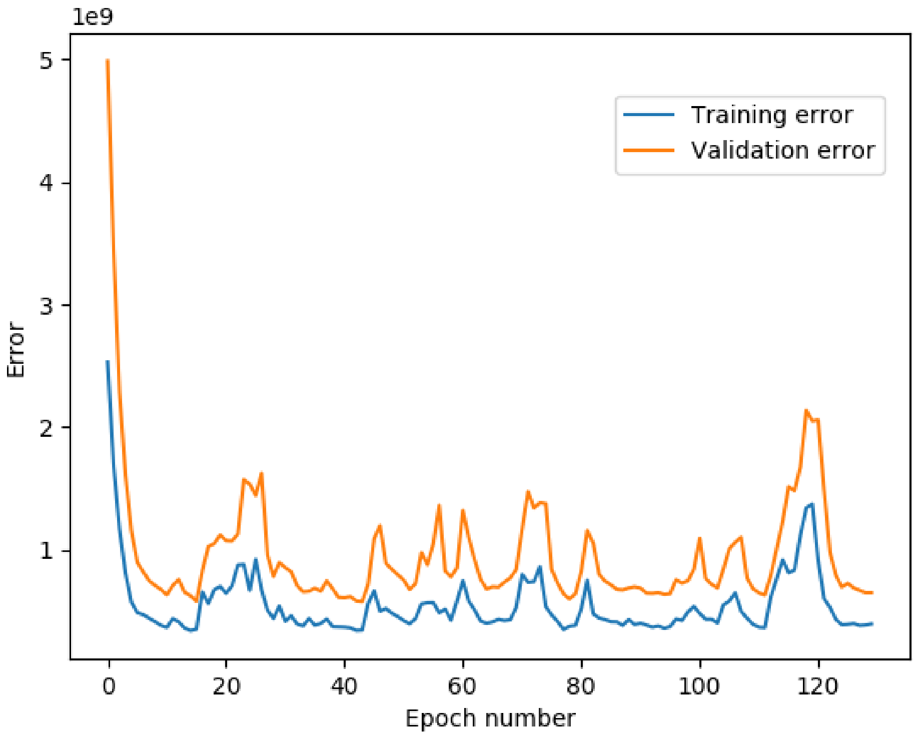


Fig. 23: Evolution of the training and evaluation for our denoising framework model.

In the previous picture we can see the training and evaluation error in the network, up to 120 epochs. The reason why this number is so low it's because the network overfitted easily, so further steps to make this network consistent need to be made. Just as the first point on the evaluation, here the best measure it's aswell synthesizing the outcome of the network to perform a perceptual analysis of the outcome. In this case, as the network wasn't too trained, the result was similar or worse than the original signal. To solve this a little bias was added to the synthesized STFT's, in order to force the network not to put too many points in the STFT to zero (LSTM networks, as it was explained before, have a memory and a forget gate that simply puts these values to zero, and in our case this happened too much). The denoised STFT result is the following:

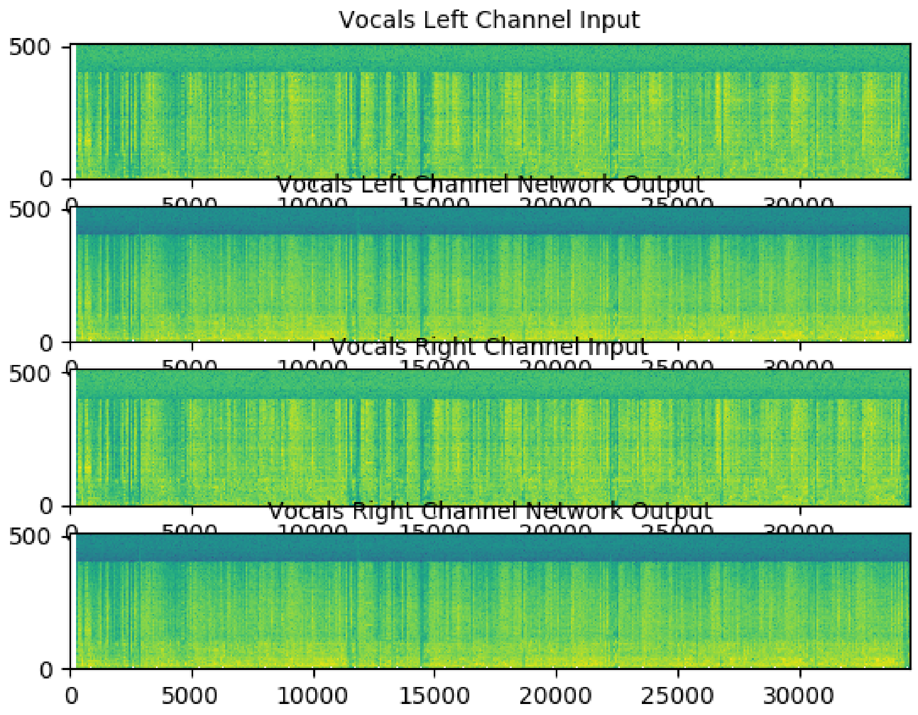


Fig. 24: First result, comparing the denoised signal to the original. The original corresponds to the first and fourth plot, whereas the denoised it's the second and last plot.

We see how it has replicated pretty well the original STFT, at least we see how the frequencies with big magnitude are present thorough the estimated STFT. Although we can see how the high frequencies have not been totally reconstructed. Still, in this case the difference wasn't that big between the estimated vocals from the original network to the denoising network. We think although that this network could have some usage, it still needs to be improved and further tested, but we have shown that recurrent autoencoders could work for denoising signals in the form of STFT's. Like in the DA section, it was decided not to further evaluate this step, as training and tweaking a whole new network from scratch takes a lot of time, specially in the case of recurrent networks, since they can overfit easily (this is why a low number of epoch was used), and usually need to be combined with DA techniques to make them robust enough. Still, in a future we believe these denoising recurrent autoencoders can have some potential in denoising signals

7 CONCLUSIONS

We have updated the framework where the baseline model was built in order to keep it up to date and to be able to be improved in the future. We have also implemented positive improvements, such as revising the architecture of the old model to be able to fully process stereophonic signals, which implies tweaking the old architecture into a new one within the framework, aswell, the architecture has been updated to be able to work with new datasets with more musical tracks. We have chosen as a dataset MUSDB2018 which contains an extensive number of testing songs, and we have shown in the results that the separation it processes, with some success. Aswell, we have proposed various methods which the network could be improved upon, such as performing data augmentation to generate a more robust model that gives out more reliable separations; or using a recurrent denoising framework based in autoencoders and LSTM layers, which have the possibility of further improving the network adding these elements as many other state-of-the-art algorithms are currently using.

7.1 Future work

We have also set some new basis to work in the future in order to improve source separation algorithms. We proposed a data augmentation framework to increase the quality and quantity of datasets in order to train network models; although due to the architecture to the model online data augmentation models aren't the best suited for it, we opened the gates and wrote the basis to be able to perform DA in our new model; as well, further improvements to generate that data could be explored, such as performing data augmentation by performing musically coherent pitch changes, adding spatial variations to sources located within our dataset, creating more reliable mixtures by using musical informed methods, etcetera. As well, musical-style datasets and models could be explored, in a way to give models the ability to specialize in a certain musical styles by further dividing the datasets, but keeping the same amount of information by performing data augmentation, if it was to ever be used in a specific application. As well, by performing the stereophonic separation the benefits of being able to achieve a binaural separation could be explored in the case we want to see it's applications in cochlear implants, or simply to perform speech/audio enhancement in applications. It would also be considered using this new model in combination of sound localization techniques to further increase this system results and reduce it's variability. Finally we have proposed a model for denoising signals that could be further studied in the future, in order to create more strong and complex networks to perform signal denoising, which has a simple architecture that would allow the building of stackable denoising units to further denoise inputs.

8 ANNEX

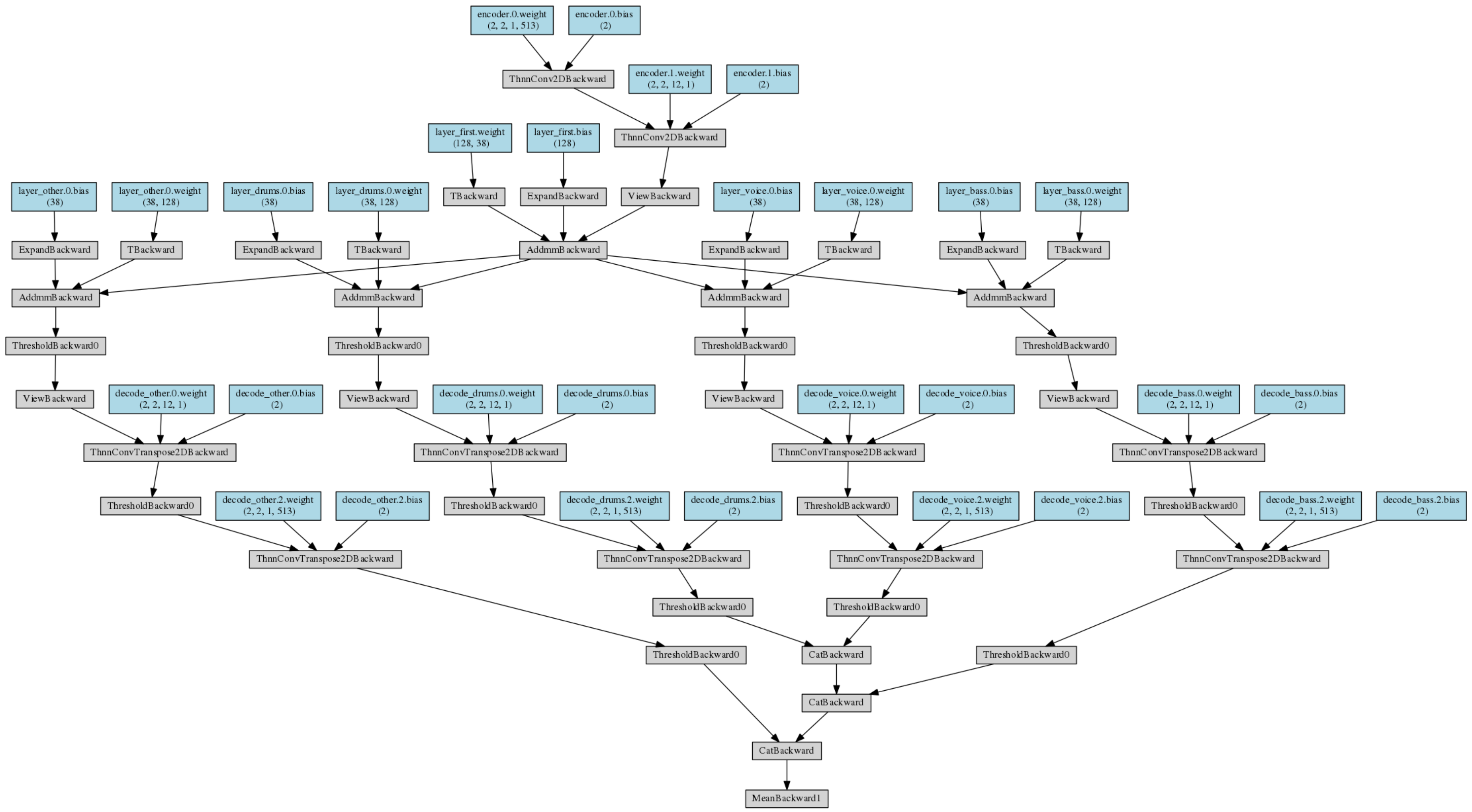


Fig. 25: The new framework backpropagation step as in understood from a PyTorch dynamic view.

REFERENCES

[1] Ossama Abdel-Hamid, Abdel-rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on audio, speech, and language processing*, 22(10):1533–1545, 2014.

[2] Abien Fred Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[3] Stanley T Birchfield and Rajitha Gangishetty. Acoustic localization by interaural level difference. In *Acoustics, Speech, and Signal Processing, 2005. Proceedings.(ICASSP'05). IEEE International Conference on*, volume 4, pages iv–1109. IEEE, 2005.

[4] Pritish Chandna. Audio source separation using deep neural networks. Master's thesis, 2016.

[5] Pritish Chandna, M. Miron, Jordi Janer, and Emilia Gómez. Monoaural audio source separation using deep convolutional neural networks. In *13th International Conference on Latent Variable Analysis and Signal Separation (LVA ICA2017)*, 02/2017 2017.

[6] Pritish Chandna, Marius Miron, Jordi Janer, and Emilia Gómez. Monoaural audio source separation using deep convolutional neural networks. In *International Conference on Latent Variable Analysis and Signal Separation*, pages 258–266. Springer, 2017.

[7] E Colin Cherry. Some experiments on the recognition of speech, with one and with two ears. *The Journal of the acoustical society of America*, 25(5):975–979, 1953.

[8] Andrzej Cichocki, Rafal Zdunek, and Shun-ichi Amari. New algorithms for non-negative matrix factorization in applications to blind source separation. In *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, volume 5, pages V–V. IEEE, 2006.

[9] Mike E Davies and Christopher J James. Source separation using single channel ica. *Signal Processing*, 87(8):1819–1832, 2007.

[10] Lawrence T DeCarlo. On the meaning and use of kurtosis. *Psychological methods*, 2(3):292, 1997.

[11] Li Deng, Dong Yu, et al. Deep learning: methods and applications. *Foundations and Trends® in Signal Processing*, 7(3–4):197–387, 2014.

[12] Sebastian Ewert and Meinard Müller. Score-informed source separation for music signals. In *Dagstuhl Follow-Ups*, volume 3. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[13] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2017.

[14] Simon Haykin and Zhe Chen. The cocktail party problem. *Neural Computation*, 17(9):1875–1902, 2005.

[15] Po-Sen Huang, Minje Kim, Mark Hasegawa-Johnson, and Paris Smaragdis. Singing-voice separation from monaural recordings using deep recurrent neural networks. In *ISMIR*, pages 477–482, 2014.

[16] D. H. Hubel and T. N. Wiesel. Receptive fields and functional architecture of monkey striate cortex. *J Physiol*, 195(1):215–243, Mar 1968. 4966457[pmid].

[17] David H. Hubel. Exploration of the primary visual cortex, 1955âeur"78. *Nature*, 299:515 EP –, Oct 1982.

[18] Tom Ko, Vijayaditya Peddinti, Daniel Povey, and Sanjeev Khudanpur. Audio augmentation for speech recognition. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[20] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[21] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object recognition with gradient-based learning. In *Shape, contour and grouping in computer vision*, pages 319–345. Springer, 1999.

[22] Daniel D Lee and H Sebastian Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788, 1999.

[23] Daniel D Lee and H Sebastian Seung. Algorithms for non-negative matrix factorization. In *Advances in neural information processing systems*, pages 556–562, 2001.

[24] Antoine Liutkus, Fabian-Robert Stöter, Zafar Rafii, Daichi Kitamura, Bertrand Rivet, Nobutaka Ito, Nobutaka Ono, and Julie Fontecave. The 2016 signal separation evaluation campaign. In Petr Tichavský, Massoud Babaie-Zadeh, Olivier J.J. Michel, and Nadège Thirion-Moreau, editors, *Latent Variable Analysis and Signal Separation - 12th International Conference, LVA/ICA 2015, Liberec, Czech Republic, August 25-28, 2015, Proceedings*, pages 323–332, Cham, 2017. Springer International Publishing.

[25] James Martens. Deep learning via hessian-free optimization. In *ICML*, volume 27, pages 735–742, 2010.

[26] M. Miron, J. Janer, and Emilia Gómez. Monaural score-informed source separation for classical music using convolutional neural networks. In *18th International Society for Music Information Retrieval Conference*, Suzhou, China, 24/10/2017 2017.

[27] Kayvan Najarian and Robert Splinter. *Biomedical signal and image processing*. CRC press, 2016.

[28] Andrew Y Ng. Feature selection, l 1 vs. l 2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, page 78. ACM, 2004.

[29] Aditya Arie Nugraha, Antoine Liutkus, and Emmanuel Vincent. Multichannel music separation with deep neural networks. In *Signal Processing Conference (EUSIPCO), 2016 24th European*, pages 1748–1752. IEEE, 2016.

[30] Sanjeel Parekh, Slim Essid, Alexey Ozerov, Ngoc QK Duong, Patrick Pérez, and Gaël Richard. Motion informed audio source separation. In *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*, pages 6–10. IEEE, 2017.

[31] Mathieu Parvaix, Laurent Girin, and Jean-Marc Brossier. A watermarking-based method for informed source separation of audio signals with a single sensor. *IEEE Transactions on audio, speech, and language processing*, 18(6):1464–1475, 2010.

[32] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[33] Barak A Pearlmutter and Lucas C Parra. Maximum likelihood blind source separation: A context-sensitive generalization of ica. In *Advances in neural information processing systems*, pages 613–619, 1997.

[34] Colin Raffel, Brian McFee, Eric J Humphrey, Justin Salamon, Oriol Nieto, Dawen Liang, Daniel PW Ellis, and C Colin Raffel. mir_eval: A transparent implementation of common mir metrics. In *In Proceedings of the 15th International Society for Music Information Retrieval Conference, ISMIR*. Citeseer, 2014.

[35] Zafar Rafii, Antoine Liutkus, Fabian-Robert Stöter, Stylianos Ioannis Mimilakis, and Rachel Bittner. The MUSDB18 corpus for music separation, December 2017.

[36] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[37] D.B. Rowe. *Multivariate Bayesian Statistics: Models for Source Separation and Signal Unmixing*. CRC Press, 2002.

[38] Justin Salamon and Juan Pablo Bello. Deep convolutional neural networks and data augmentation for environmental sound classification. *IEEE Signal Processing Letters*, 24(3):279–283, 2017.

[39] Hiroshi Saruwatari, Toshiya Kawamura, Tsuyoki Nishikawa, Akinobu Lee, and Kiyohiro Shikano. Blind source separation based on a fast-convergence algorithm combining ica and beamforming. *IEEE Transactions on Audio, speech, and language processing*, 14(2):666–678, 2006.

[40] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[41] Paris Smaragdis and Judith C Brown. Non-negative matrix factorization for polyphonic music transcription. In *Applications of Signal Processing to Audio and Acoustics, 2003 IEEE Workshop on.*, pages 177–180. IEEE, 2003.

[42] Richard Socher, Brody Huval, Bharath Bath, Christopher D Manning, and Andrew Y Ng. Convolutional-recursive deep learning for 3d object classification. In *Advances in Neural Information Processing Systems*, pages 656–664, 2012.

[43] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[44] S. Uhlich, M. Porcu, F. Giron, M. Enenkl, T. Kemp, N. Takahashi, and Y. Mitsufuji. Improving music source separation based on deep neural networks through data augmentation and network blending. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 261–265, March 2017.

[45] Emmanuel Vincent, Rémi Gribonval, and Cédric Févotte. Performance measurement in blind audio source separation. *IEEE transactions on audio, speech, and language processing*, 14(4):1462–1469, 2006.

[46] Emmanuel Vincent, Maria G Jafari, Samer A Abdallah, Mark D Plumbley, and Mike E Davies. Blind audio source separation. *Queen Mary, University of London, Tech Report C4DM-TR-05-01*, 2005.

[47] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11(Dec):3371–3408, 2010.

[48] Tuomas Virtanen, A Taylan Cemgil, and Simon Godsill. Bayesian extensions to non-negative matrix factorisation for audio signal modelling. In *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*, pages 1825–1828. IEEE, 2008.

[49] Xinchen Yan, Jimei Yang, Kihyuk Sohn, and Honglak Lee. Attribute2image: Conditional image generation from visual attributes. In *European Conference on Computer Vision*, pages 776–791. Springer, 2016.

[50] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.