EMULATING ELECTRIC GUITAR EFFECTS WITH NEURAL NETWORKS

Graduation Project
Computer Engineering
Universitat Pompeu Fabra

David Sanchez Mendoza

Advisor: Xavier Amatriain Rubio

September 2005

Dedications and Gratitudes

Thanks to:

Xavier Amatriain and Alex Loscos for their help and dedication.
All the people at Music Technologhy Group and specially to Amaury Hazan for helping me on the real-time application design, Jose Lozano for being a great sound technician and Pau Arumi and Miquel Ramirez for their useful help.
All my friends at the Universitat Pompeu Fabra: Xava, Chesco, Ferran, Gustavo, Xavi, Mar, Xev, Helena, Raul, Dani, Marc, Vicens, Carol, Victor, Jesus, Ignasi, Jandro, Montse, Quimo, Victor, ... and a long etc.
My music teacher Lluc for his wise advices and for making me understand that not all is based on maths and formulas, some things simply have no explanation.
Herminia for correcting this document.
And, obviously, to my parents Gabriel and Gloria for their affection and confidence.

This graduation project is dedicated to all the people who are always fighting for the union of art and science, because they are exactly the same wonderful thing.

'Genius is 1% inspiration, and 99% perspiration.'

(Thomas A. Edison)

Contents

List of Figures

Chapter 1

Introduction

1.1 Goal of the project

The project main goal is to implement, with the highest possible accuracy, one of the most classic guitar effect, that is overdrive, using neural networks.

1.2 Context

In the world of the electric guitar effects are often used in order to modify the "natural" guitar sound, the most commonly way to apply effects to a guitar is by means of the so-called pedals.
In the 60's and 70's those effects were totally analogical but from 80's to nowadays those effects became digital, that is, with an internal processor that emulates the circuit of the original analogical effect or implements a filter which transforms the natural guitar sound closer to the original analogical effect. Because of this change guitar effects became cheaper but also lost some quality, that is, those digital ones sound sometimes "too digital" and processed, an obviously don't sound as warm as the analogical ones. I'm not saying those effects are absolutely bad, it's a fact that their sound is very near the original one but obviously it's not the same.
On 1.1 we can see that the natural guitar signal is processed by the effect pedals, then the modified signal is sent to the amp.

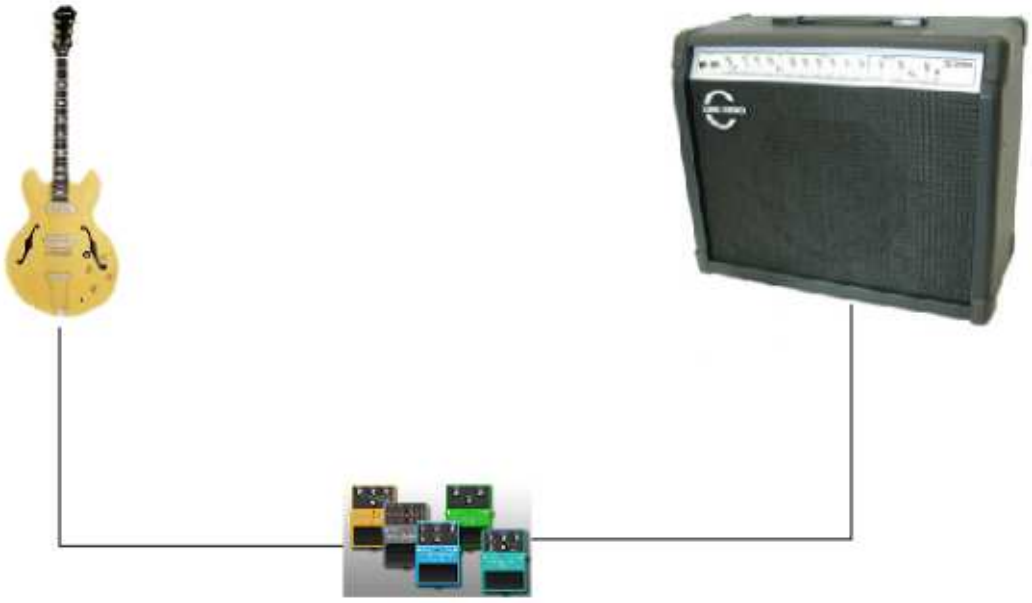

Figure 1.1: Processing the guitar signal

1.3 Abstract

Our guitar effect modeling will not be done by emulating the original circuit or implementing a software filter, instead of this we will show the computer how an analogical pedal modifies the sound and then the computer will learn it; late in this report the learning process will be discussed with more deepness.
It is very important to notice that we will not take into consideration the internal pedal circuit, instead of this the computer will just learn "the sound"; due to this we have a very important advantage: don't care if the pedal circuit is so complicated or not, system should be considered a blind entity which only watches the audio input and the pedal output, thus, taking account of the differences between those signals the system should be able to learn "what's the difference", that is, the transfer function.
In order to make the algorithm learn the transformation we are going to use neural networks, we will consider a pedal effect as a function which applies some transformation to the guitar sound, thus the computer is going to learn in which way the

pedals transforms the natural (not modified) guitar signal. Obviously we want our neural networks to be universal in the sense that they should transform with the highest possible accuracy any kind of guitar signal, from a single note to a four note chord.

Chapter 2

Requirement Analysis

2.1 Functional Requirements

In this section we are going to discuss what does the system should perform, we are going to explain just the different features but for now we will not give details about their implementation.
As we've said before the system should be able to emulate some kind of effect pedals, concretely it should modelate an *overdrive* and a *chorus*; those are two of the most popular and most used guitar effects: the first one saturates the signal as the tube amps do, it gives the sound a very warm and beautiful saturation which is the base of 60's and 70's rock (think about Hendrix, Led Zeppelin...); the second effect delivers to the sound some kind of deepness, it makes the sound thicker and give it some presence; this effect is used in any kind of music when you want to emphasize some phrase.
Both effects have some parameters in order to adjust it, on the *overdrive* there is one called *gain* which controls the amount of distortion given to the signal; on the other hand, the chorus has the depth, which sets the effect deepness, and the rate (a sophisticated parameter that sets the effect tremolo). Obviously the system should also be able to perform those parameters for the user to use them.
Among the guitarists is very common chaining some effects, for example if someboy wants a saturated sound with some delay one should concatenate an *overdrive* and a *delay*; in this sense our system must be able to chain any effect that has learned.

Another important aspect is the effect order, for example: it's not the same *overdrive + delay* that *delay + overdrive*, the correct way is the second one; the system should advice the user if it wants to do an effect chaining which is not correct.
Another important guitar effects feature is that they work in real-time, one play its guitar and the effect is applied "instantly" almost for our human perception; our system should do exactly this, a real-time tranformation of the natural signal so that it can be used in the same way as a common pedal.
In order to explain better the general system way of work let's see the system use case diagram (2.1), which will clarify it all.

2.2 Use Cases

On figure 2.1 the use case diagram can be seen, then we will explain each of them with more detail.

Now we are going to expand a little each one of the use cases for the reader to have a deeper vision into them.

2.2.1 Add Effect

Use Case: Add Effect.
Context: User adds some effect to the effects chain.
Primary Actors: User.
Support Actors: None.
Pre-conditions: Effect must exist.
Succes Post-Conditions: Effect is added to the effects chain and its size is incremented by one.
Fail Post-Conditions: Effects chain size is the same as before the operation.
Main Succesful Scenario:

1. User chooses an effect.

2. System checks if it exists.

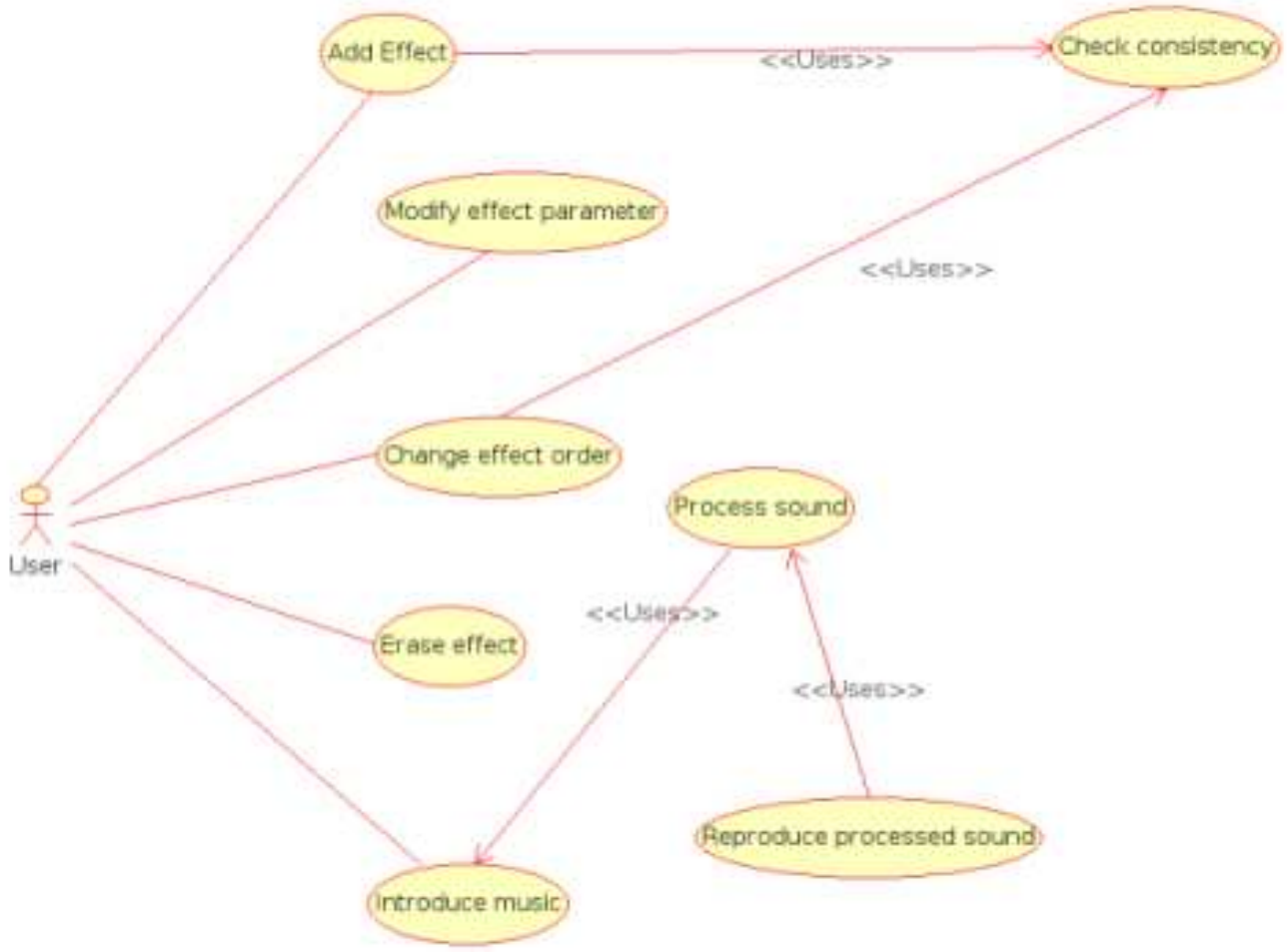

Figure 2.1: System use case diagram

3. System adds it to the effects chain and increases its size by one.

Extensions:

- 2.a Effect doesn't exist.

- 2.a.1 Error Message shown to the user.

2.2.2 Change Effect Chain Order

Use Case: Change effect chain order.
Context: User changes the effect chain order .
Primary Actors: User.
Support Actors: None.
Pre-conditions: Chain size must be greater than 1.
Succes Post-Conditions: The order of the effects are changed according to the user.
Fail Post-Conditions: The effects order is the same as before the operation.
Main Succesful Scenario:

1. System checks if chain size is greater than on
2. User chooses an effect.
3. User chooses the new position.
4. System changes the effects order.

Extensions:

- 1.a Effects chain is ¡= 1
- 1.a.1 Error Message shown to the user.

2.2.3 Modify Effect Parameter

Use Case: Modify effect parameter
Context: User modifies some effect parameter.
Primary Actors: User.
Support Actors: None.
Pre-conditions: Effect must belong to the effects chain.
Succes Post-Conditions: Effect properties are changed according to the new parameter value
Fail Post-Conditions: Effect properties remain the same as before the operation
Main Succesful Scenario:

1. User chooses an effect.

2. User chooses a parameter.

3. System changes the properties of the effect according to the new value.

Extensions:

- 1.a Effect doesn't belong to the effects chain.

- 1.a.1 Error Message shown to the user.

2.2.4 Erase effect

Use Case: Erase effect
Context: User clears some effect from the effects chain.
Primary Actors: User.
Support Actors: None.
Pre-conditions: Effect must belong to the effects chain.
Succes Post-Conditions: Effect does not belong to the effects chain and its size is reduceb by one
Fail Post-Conditions: Effects chain size is the same as before the operation
Main Succesful Scenario:

1. User chooses an effect.

2. System erases it from the effects chain.

Extensions:

- 1.a Effect doesn't belong to the effects chain.

- 1.a.1 Error Message shown to the user.

2.2.5 Introduce Music

Use Case: Introduce music
Context: User introduces music to the system playing a guitar.
Primary Actors: User.
Support Actors: None.
Pre-conditions: System is listening and receives data.
Succes Post-Conditions: Effect chain is applied and the sound is reproduced
Fail Post-Conditions: No effect applied or no sound reproduced
Main Succesful Scenario:

1. User play its guitar.
2. System reproduces the processed sound.

2.2.6 Check Consistency

Use Case: Check consistency
Context: System checks if the effects chain order is correct.
Primary Actors: None, system does it.
Support Actors: None.
Pre-conditions: Effects chain must be greater than one, otherwise there's nothing to check.
Succes Post-Conditions: System decides correctly if the effects chain order is correct or not
Fail Post-Conditions: System decision is not consistent
Main Succesful Scenario:

1. User adds an effect.
2. System checks if effects chain is greater than one.
3. System checks order consistency.

Extensions:

- 2.a Effects chain size is ¡= 1
- 2.a.1 Skip operation.
- 3.a Order is not correct
- 3.a.1 System advices the user and corrects the effects chain order

2.2.7 Process Sound

Use Case: Process sound
Context: User applies the effects chain to the inputs.
Primary Actors: None, system does it.
Support Actors: None.
Pre-conditions: There must be an input ontroduced by the user.
Succes Post-Conditions: System applies some changes to the input according to the effects chain
Fail Post-Conditions: Sound process is not consistent with the effects chain
Main Succesful Scenario:

1. User plays its guitar.
2. System receives the data and applies the changes according to the effect chain.

2.2.8 Reproduce Processed Sound

Use Case: Reproduce processed sound.
Context: System synthesizes the transformed sound.
Primary Actors: None, the system does it.
Support Actors: None.
Pre-conditions: System needs the data spectrum (magnitude and phase).
Succes Post-Conditions: The processed sound is synthesized.
Fail Post-Conditions: There's no sound or the sound is not correct.
Main Succesful Scenario:

1. System synthesizes the sound with the data spectrum.

2. System sends the result to the audio card.

3. System adds it to the effects chain and increases its size by one.

2.3 Non-Functional Requirements

Now we are going to focuse in how the system will perform the features explained before. The training process will be done with the audio signal directly or instead of this we will work with the spectral domain of the signal as in many audio processing applications, it will depend in which manner the network perfoms in both domains. Obviously working on spectral-domain implies that the system should use a Fourier transform implementation, concretely our system must use the FFT (Fast Fourier Transform) algorithm, which performs this transformation in an efficient and fast way.
Once we have the signal spectrum we want a tool for the computer to learn the transfer function (effect transformation), the system will learn it with a neural network, concretely it will use the Multi-Layer Perceptron, which is a universal neural network architecture and can fit any kind of function with any kind of dimensions (concretely we are going to use a four-layer perceptron). Apart from the architecture, we need to define the stop criterion, in our case it will be the mean-square error (it will be explained later).
In order to have a consistent training, the training data should be recorded in pairs, that is, one must record the guitar clean signal (with no effect) and split this signal towards the effect and record it after it has been applied (first one stands for the input data and the other one for the target data), 2.2 shows how it has to be done. All those audio files should be in WAV format and mono.
Now let's focuse a little bit on the technology that the system has to use: the application code language must be C++ and it must run with no problems under GNU/Linux; apart from this, any kind of framework or library used in the project must be free software because this project is free software too.
To perform any kind of operation which involves audio or signal processing the system must use the CLAM framework (www.iua.upf.es/mtg/clam), and for all the aspects concerning to machine learning we will use the TORCH framework (www.torch.ch); both frameworks will be detailed later.

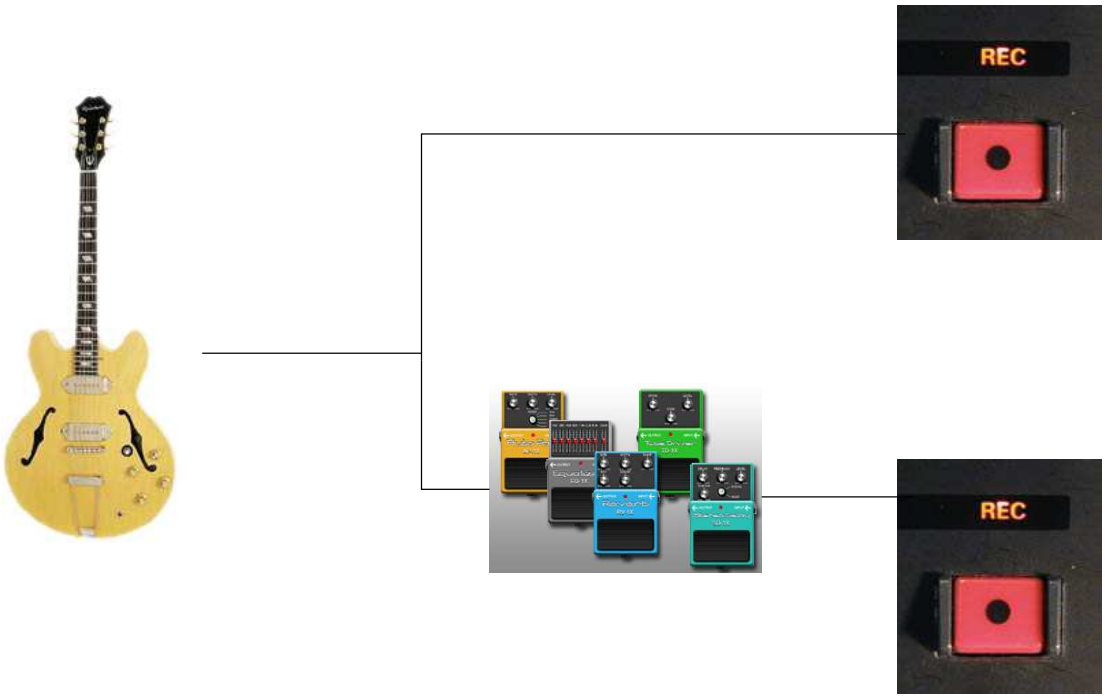

Figure 2.2: Recording the training data

Chapter 3

Tools and Concepts

3.1 Object-Oriented Programming

The object-oriented programming (OOP) is a programming paradigm in which all the concepts of a program are mapped into an entity called *object*.
The main difference in relation to traditional programming is that the OOP is based on *Which objects do I need to perform some system?* while traditional one stads for *What does this system do?*
A key concept in OOP is obviuosly the object concept: an object is a software piece related to some methods and some members (explained below) and it is often used to model real-world objects we find in our daily life. These objects are built essentially around these parts:

- Methods: the actions that an object is able to do, usually an object has at least two actions: the constructor (builds the object) and the destructor (destroys the object once it has been used). Methods tell us about the object's behaviour
- Members: the parts which compose the entire object. These parts determine the object's internal state.

We call *class* to the prototype that defines the methods and the members which are common in all the objects of this class, these concrete objects are called *instances*; thus, one class could have N instances while one instance only belongs to a class (or

more if there is inheritance, but now it doesn't matter).
Let's see now an example of a class with its methods and members, we are going to define a class which models a car, this class could be:

```
class Car
{
     // members
     integer speed;
     iteger fuel_level;
     integer gear;
     bool isStopped;
     bool fuelEmpty;

     // methods
     Car();  // constructor
     Accelerate();
     Brake();
     IncreaseGear();
     ReduceGear();
     FillFuelTank()
     ~Car(); // destructor
}
```

As we have seen, conceptually, a class is very close to the object that it represents in the real world, thus the OOP makes it easier to build complicated systems in which there are a lot of objects. It must be said that, before starting to program some software in OOP, the previous conceptual task is longer and harder than in the traditional programming, needless to say that after the system is modeled it is easier to program and mantain.
In this project all the code is pure object-oriented, which means that everything in the program is related to an object and a class. There are some programming languagues like C++ that permits a hybrid OOP, having both classes and "traditional" code, others like Java only permit pure OOP. Despite this, we have chosen C++ as our programming language but taking in account what we have said before,

our code will be truly object-oriented eventhough the chosen language permits us another techniques.

3.2 The C++ Programming Language

To begin with, in every application a very important decision involves in which programming language this application is written. Particularly we have chosen C++ for several reasons:

- It's a non-propietary programming language, it's a very important feature according with our free software philosophy.
- Supports object-oriented paradigm, which is our preferred way of programming due to its capabilities structuring the code and its world conception, which is very close to the real one.
- It's faster than other similar languages, specially under GNU/Linux, and this is a very important feature because our project should work in real-time.
- It's standard: ANSI (The American National Standards Institute), BSI (The British Standards Institute), DIN (The German national standards organization), several other national standards bodies, and ISO (The International Standards Organization).

Apart from this features (which are more *conceptual*) C++ has more features that makes it, in my humble opinion, the best general purpose programming language. Some of these features are:

- Possibility to create abstract classes, which work as an interface.
- Multiple inheritance and class hierarchies for OOP.
- Templates: a very powerful tool to develop generic software.
- Exception handling.
- Possibilty to create namespace in order to clarify large-scale systems.

- Constains the C subset of C++ to perform full compatibility.
- Standard containers and algorithms (Standard Template Library) which contain lots of data strucutures and algorithms such as stacks, linked lists, sort algorithms...

3.3 Machine Learning

Machine Learning (ML) is a science that studies the ability for a computer system to develop new knowledge based on its past experiences, in this sense a computer can learn some properties of some given data with no prior knowledge about this data. This learning process can be supervised or unsupervised: on one side supervised learning stands for the process in which the learning process comes *supervised* from outside the system: for example if we want our system to learn a simple sine function we will tell it some sine function values and the computer will be capable to generate any value of this function; on the other side, under the unsupervised learning the system generates some knowlegde from some unknown data: for example, somebody has a recording in which two instruments play a piece of music and want to separate them, an unsupervised system is able to do it.
On ML there is a large amount of architectures and algorithms to perform a lot of learning capabilities, one of them are the neural networks; this architecture is able to solve two kind of problems: fitting functions are separating some data in groups that have the same features. Taking in account that we need to fit some functions (spectrums) we must use this kind of architecture, concretely the non-linear neural networks: those are able to learn any kind of non-linear function with any number of dimensions. Our project involves the learning of one non-linear function by the computer (a transfer function, more concretely), thus non-linear neural networks were used by the computer to be able to learn it.

3.3.1 The Multi-Layer Perceptron

The structure of neural networks we've used is the Multi-layer perceptron (MLP), this kind of architecture works as a universal function fitter or a classifier, that is,

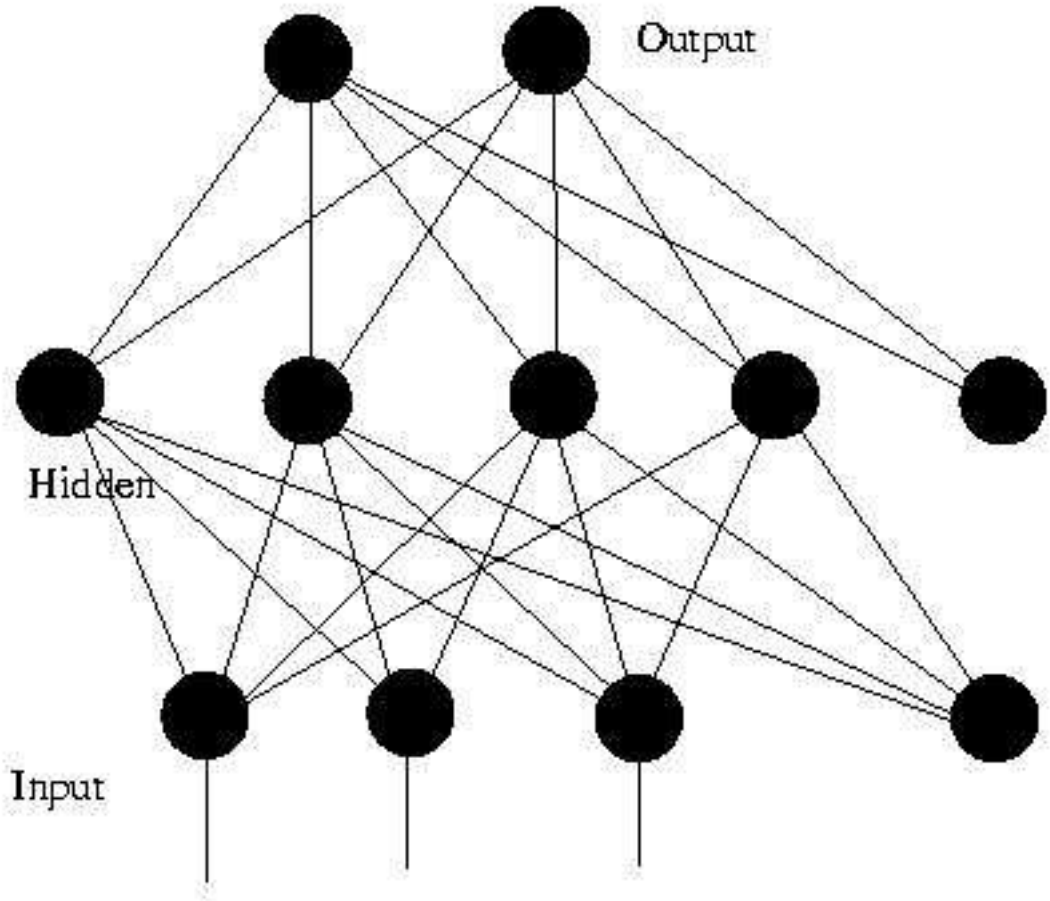

Figure 3.1: Structure of a simple MLP

on one hand it can learn any kind of function with any number of dimensions and on the other hand, virtually it can solve any kind of classification problems. An MLP learns in supervised mode, which means that, apart from the input, the MLP knows where it has to arrive because it requires a desired response to be trained.
An MLP has one input layer, some hidden layers (usually one or two) and one ouput layer: the hidden ones are formed by non-linear neurons while the output layer can be formed by linear o non-linear neurons (see figure 3.1, which shows an MLP with only one hidden layer, don't care about not connected neurons, just retain the MLP structure). MLP are trained with an algorithm called *back propagation*, it's not the purpose of this project to explain in deepness on what consists this algorithm,

however let's see a conceptual abstraction of it in order to understand (in a very "high level" way) how it works.
In the backpropagation learning algorithm the network begins with a random set of weights for each neuron. An input vector is computed through the network, and the output values are calculated using this initial weight set. Next, the calculated output is compared with the measured output data, and the squared difference between this pair of vectors determines the overall system error. The network attempts to minimize this error using the gradient descent approach, in which this global error is propagated through all the hidden layers and the network weights are adjusted in the direction of decreasing error.

3.4 Digital Audio Processing: Analysis and Synthesis

This project needs to work with the audio signal spectrums in order to perform the learning and the transformations, in this sense we need to implement an analysis/synthesis module.
Let's see a little theory to understand the necessary concepts to analyze an audio segment, make some transformations in its spectrum and reconstruct it. The concepts explained below are used in the same way in this project.
Our data will be an audio chunk that has 1024 samples (20 ms.) in which we will apply a Fourier Transform and then we will apply an Inverse Fourier Transform to reconstruct it. Once we have applied the fourier transform to our signal, we obtain the signal specturm (both magnitude and phase components) which size 513 (half chunk size plus one) each component.
If we take the audio chunk and input it directly to the FFT what we are doing is multiplying the audio with a rectangular window, this means that we are making a convolution between the spectrum and the rectangular window transform (which is not a good practice in order to get a good analysis. What it has to be done is multiplying the audio by a temporal window that shows better spectral features, for example the Blackman-Harris 92 (see figure 3.2). Apart from applying the window, we also normalize it in order to make easier to do the spectral transformations done

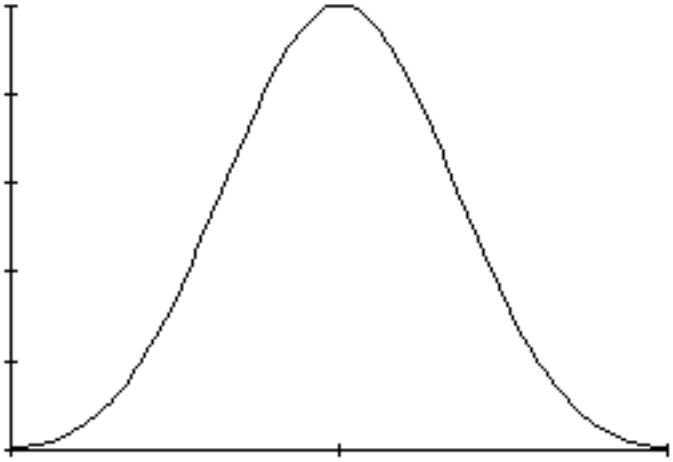

Figure 3.2: Time Domain Representation of a Blackman Harris window

later. Once we have our windowed audio signal we should shift this new signal with the aim of having zero-phase conditions in the spectrum.

Let's see now some pictures that will show in a graphical way which are the exactly transformations that we are doing to our audio chunk before doing the Fourier Transform.
In figure 3.3 we can see the original audio chunk, it belongs to a sweep signal, in figure 3.4 there's the same signal but with a blackman harris window applied and normalized to have a better spectrum; and then in figure 3.5 we have the shifted signal, we will apply the Fourier transform to this last signal. All those last figures represent audio chuncks, so the X axe stands for the samples and the Y axe for the magnitudes.

Now that we have obtained the spectrum we can do any kind of transformation we want on it before starting the reconstruction. In order to get the audio signal from the spectrum we must, apart from applying the Inverse Fourier Transform, "undo" all the modifications that we have done to the audio before starting the analysis process. This involves the next operations:

- Apply the Inverse Fourier Transform.
- Undo the circular shift (shifting winth the same amount but multiplied by -1).

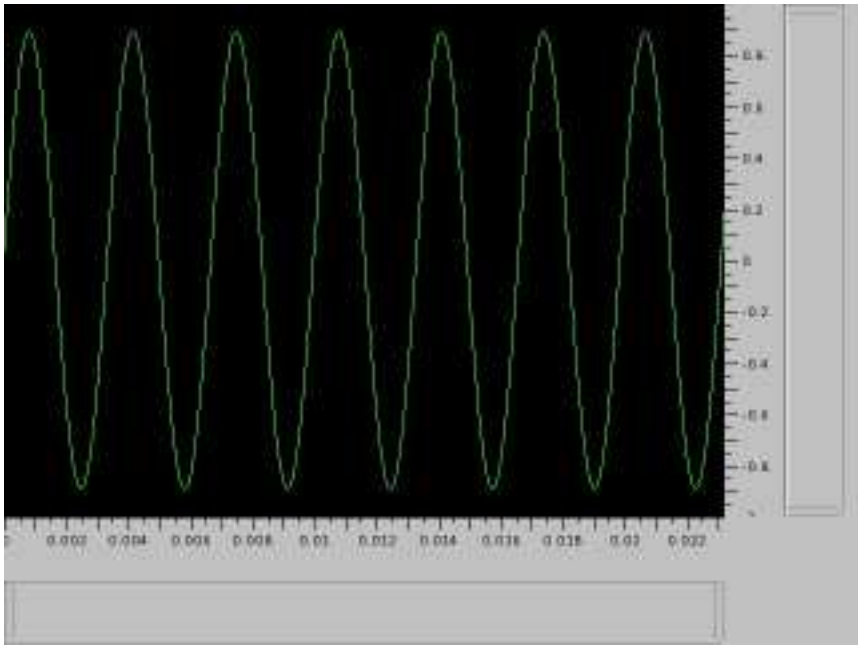

Figure 3.3: Original Audio Chunk

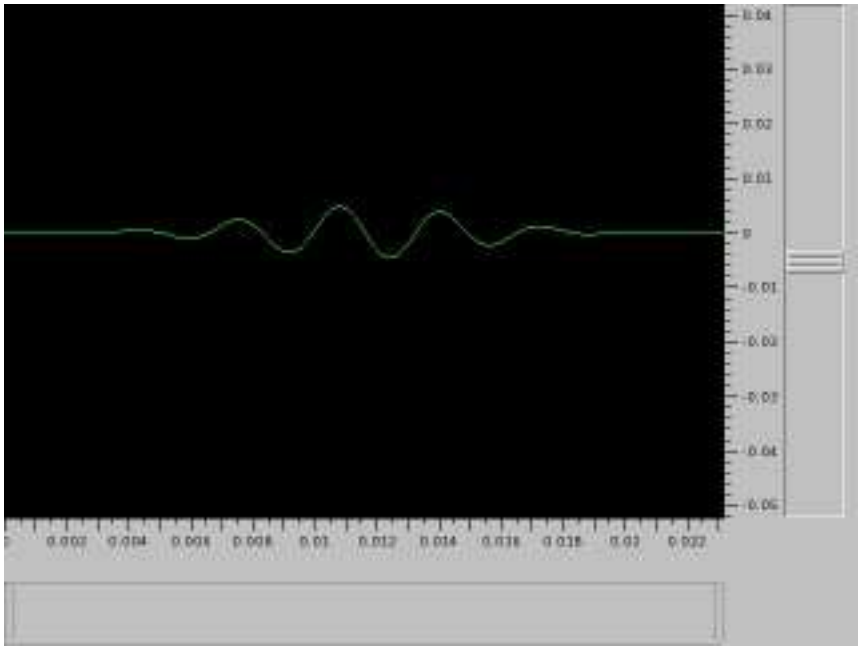

Figure 3.4: Windowed Audio Chunk

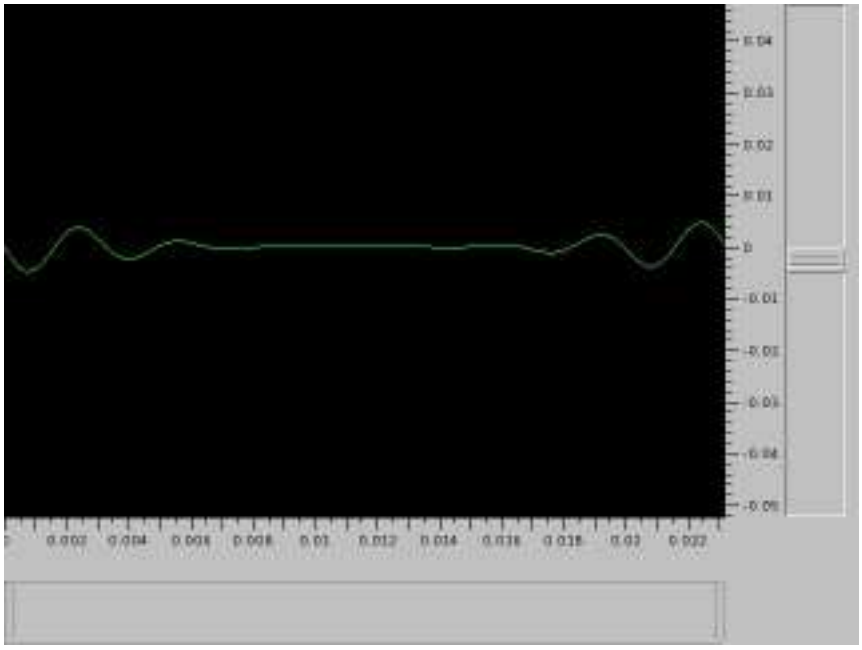

Figure 3.5: Shifted Audio Chunk

- Undo the windowing multiplying the signal by the inverse of the window used in the analysis.

After this transformation we will have our original audio signal again.
Apart from all things explained above, it is not enough to have a good Analysis/Synthesis module, we also need one more detail called *overlap*. This concept consists on overlapping the audio chunks that we are passing to the Fourier Transform, for instance, the first audio chunk goes from sample 1 to sample 1024 and the second audio chunk goes from sample 512 to sample 1535. The overlap has to be done to avoid some artifacts such as clipping when reconstructing the audio frame by frame.
To get a good overlap process we must multiply the final audio chunk we get in the final step of the reconstruction by a triangular window which has the shape like figure 3.6, then, when we have all the audio chunks we should simply overlap them and we will get a reconstructed audio signal with no artifacts.

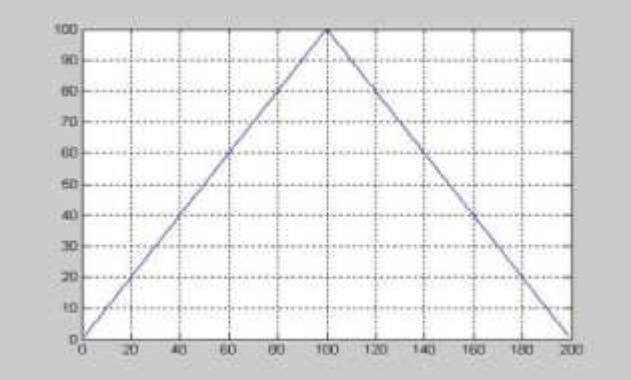

Figure 3.6: Triangular Window

3.4.1 An Efficient Algorithm for the Fourier Transform

We have talked about the Fourier Transform before but we have not mentioned yet how we are going to implement it. There are a lot of algorithms to do so, but the most effcient is the FFT (Fast Fourier Transform) algorithm. Let's see a little comparison between the FFT and another algorithm which computes the Discrete Fourier Transform (DFT).
The two algorithms above are stands for two different ways to implement this transformation, remember one can do it computing the DFT in an expensive way which is O(N^2), this function grows so fastly due the large number of operations needed to perform the algorithm. The FFT provides us a much more efficient way of implementing the DFT. The FFT algorithm requires only O($N * log(N)$) computations to compute the N-size DFT.
To get an idea of this difference: computing a 10^{12}-point Fourier transform in the expensive way spends more than 10 days in computing the algorithm while the FFT spends only 6 seconds, thus, the decision is obvious.
We have used an implementation of the FFT algorithm called FFTW (www.fftw.org) which is a free software subroutine library for computing the DFT in one or more dimensions.

3.5 CLAM: C++ Lybrary for Audio and Music

CLAM is an object-oriented framework for application development in the field of digital signal processing. It provides a large variety of algorithms and features for audio analysis/synthesis which are very valuable for this project. This framework also offers tools for basic audio I/O that help us not to lose time programming low level I/O methods. CLAM is announced to be a truly framework since apart from a set of algorithms it provides a conceptual metamodel or way of doing things.
We have decided to use CLAM due tp we have programmed a little bit under this framework before starting this project, apart from this we think CLAM offers a very wide spectrum of possibilities in the sense that is not centered in a specific aspect of the audio processing but covers most of the applications that can be done in this field. Needless to say the fact that is a truly framework is useful when programming because when one understands how does CLAM work is very comfortable to program under it.
Let's talk a little bit about the CLAM model, CLAM is based on the fact that *any signal processing can be modeled as a set of linked objects*, and on this sense there are different kinds of objects which provide the necessary tools in order to reach a complete process in digital audio. Those objects, explained below in a detailed way, are *Processing*, *ProcessingData* and *ProcessingComposite*.

3.5.1 Processing

Processing class provides an abstract model of a general process, in this sense this class offers methods for configuring and executing a process. Basically a process has the next components:

- Data Input: data which is sent to the *Processing* via the *Inputs Ports* in order to process it.
- Data Output: processed data which the *Processing* sends to the *Ouputs Ports*
- Controls: parameters that can modify the object behaviour once it is running.

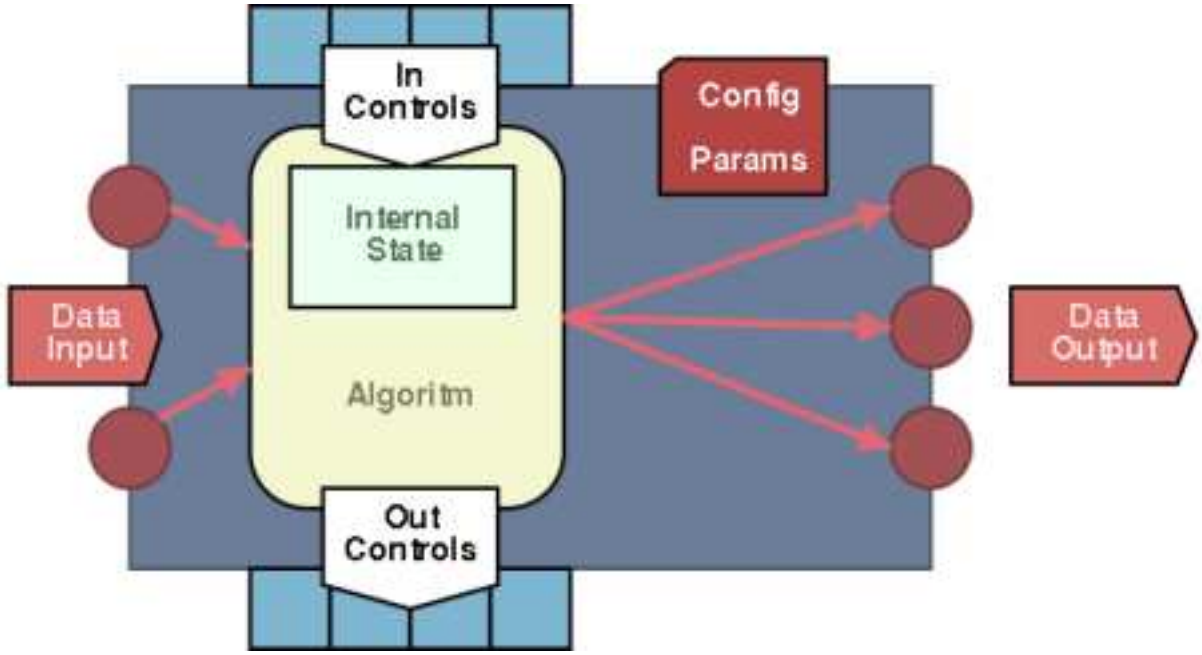

Figure 3.7: Components of a CLAM Processing (taken from [4])

- Algorithm: what the object does when one executes it.
- Configuration: parameters needed by the algorithm to perform the process, must be defined before executing the *Processing*.
- Internal State: defines the process state: Unconfigured, Ready or Running.

Figure 3.7 illustrates the different component of a *Processing*.

ProcessingConfig

Each *Processing* object has associated a *ProcessingConfig* object, this class defines all the parameters that the algorithm needs to be executed, in other words, variables that do not change during the execution.
One example of these variables could be the configuration for the FFT algorithm, which defines the audio size, the spectrum size, the window type, the amount of circular shift, etc.

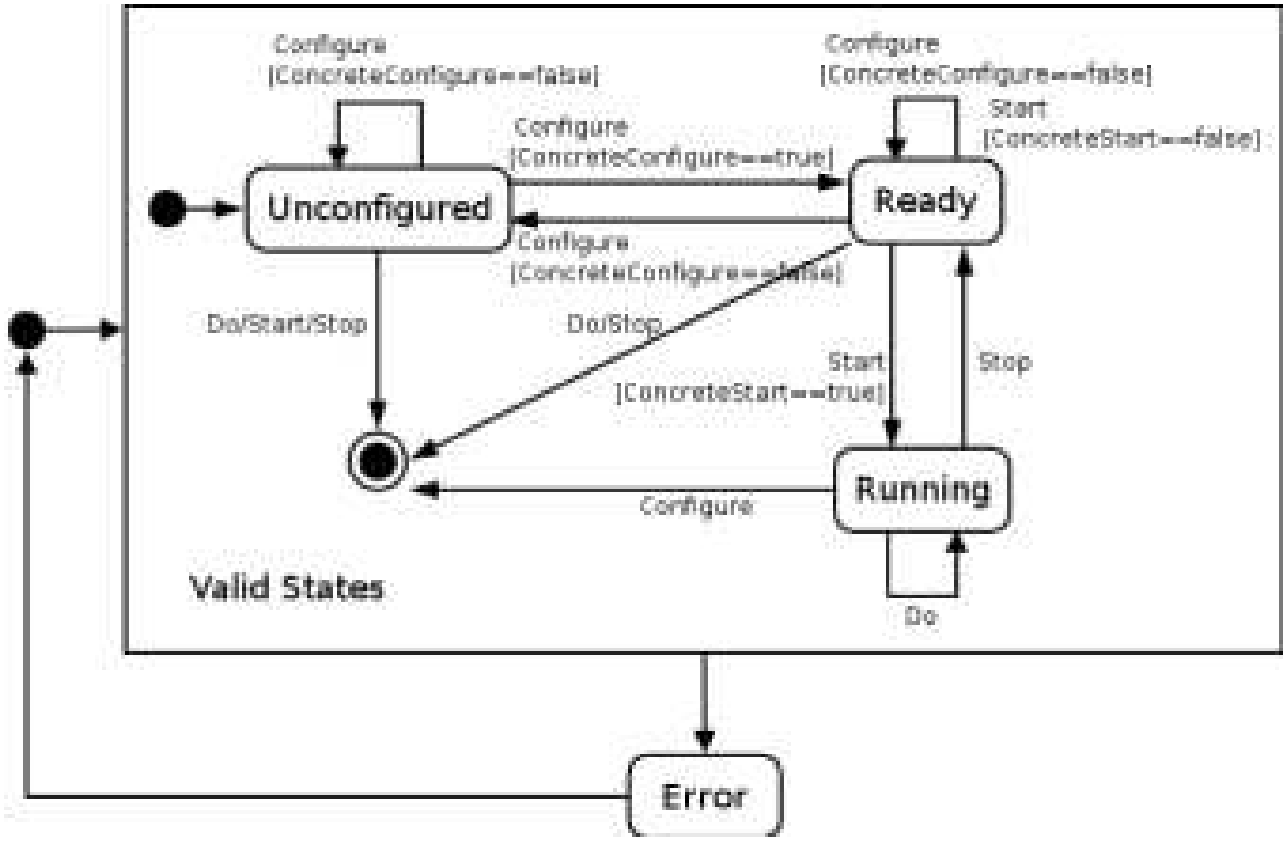

Figure 3.8: States of a CLAM Processing (taken from [4])

Processing Internal State

As we said before, a *Processing* object has three differents states that we will explain immediately. These are the three main states of a *Processing* (In figure 3.8 can be seen the different *Processing* states and its transitions):

- Unconfigured: the object has been created and it is waiting to be configured.
- Ready: the object has been configured and it is waiting to be executed or to be reconfigured.
- Running: the object is executing its algorithm.

When object is *Unconfigured* we can go to the *Ready* state if we configure it and the configuration is successful, if not we will remain *Unconfigured* and obviously the

object could not be started or stopped (before running it we must *Start()* the object and once it has finished we must *Stop()* it. Notice that the configuration is done by calling the method *ConcreteConfigure()* with a configuration as an argument.
Once we are *ready* we can, on one hand, reconfigure the object due to some changes we want to introduce or, on the other hand, we can finally *Start()* the object in order to go to the running state (notice that if we try to run or stop the object not being started there will be an error).
At this moment, we are at the *running* state, now we can call the *Do()* method to execute the *Processing* object algorithm the times we need and then, once we have finished our process, *Stop()* the object and be in the *Ready* state again.

ProcessingData

The *Processing* object explained above works with *Processing Data*, which means that as inputs and ouputs only can handle this kind of data. The main reason is that the *ProcessingData* object provides some very interesting features that make easier the programming task and the data manipulation. The *ProcessingData* objects features are:

- Have the ability to be self-described, they have some attributes referencing to its size, how many attributes contain.
- Have an homogeneous interface which permits accessing the data in the same way for all the *ProcessingData* objects.
- Attibutes are private (or protected) and only can be accessed via *Get()* or *Set()* methods.
- And many more...

Here it is very suggestive to talk about the CLAM Dynamic Types (DT), which permits us to have in memory only some object attributes, thus, the non-needed attributes are not in memory. There are three main reasons why the DT's have been implemented:

- Some core CLAM classes have a very large number of attributes and this could represent a very important waste of space if its memory is always allocated and we only need a subset of this attributes.

- We want to work with aggregates and compositions of DT's, and in this compositions we can do assignations and clone functions not having to write the specific method in each of the composition's DT's.

- As it's said before we need introspection in each DT, we want to know the name and type of each dymanic attribute, to iterate through these and having some handlers for each. On possible application of this instrospection is in data storage, it allows us to implement it generically making it totally transparent for the user.

3.5.2 ProcessingComposite

A set of *Processing* can be arranged in order to get a new and bigger *Processing*. In this new big *Processing* all the internal *Processing* are hidden, thus it can be considered an abstraction or a more complex system to which belongs the whole bunch of internal *Processing*.
The *ProcessingComposite* is defined in compilation time, it is static and it can not be modified at running time. Later we will talk about the CLAM networs, in a few words a CLAM network is a *ProcessingComposite* that can be modified in execution time. Figure 3.9 illustrates how a *ProcessingComposite* works.

3.6 Torch: A modular machine learning software library

Torch is a free software library that provides a large set of algorithms and data structures in order to develop applications that involves machine learning techniques. Torch is fully written in C++ and consequently it is very easy to integrate it with CLAM; this is the main reason to choose Torch in front of other machine learning

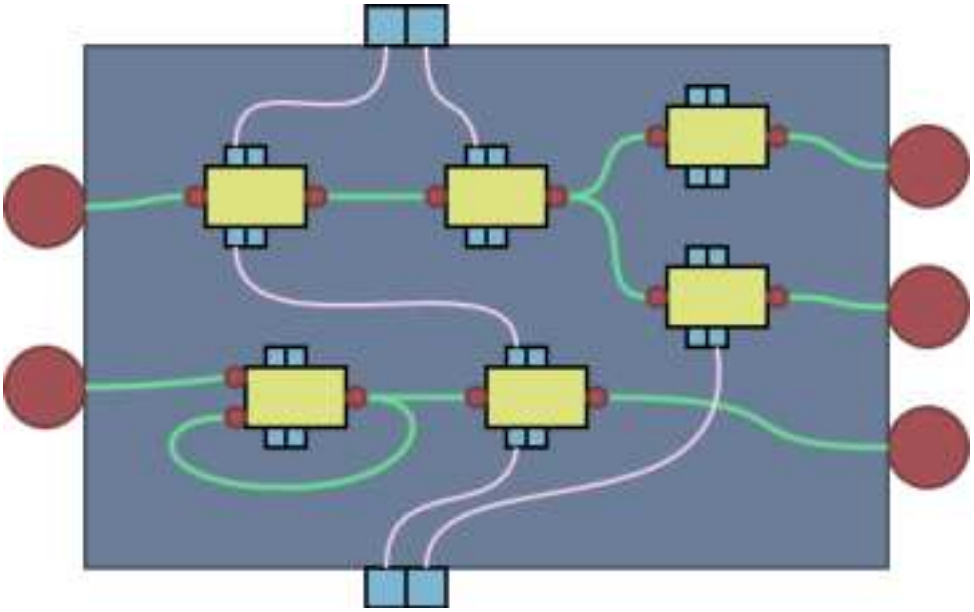

Figure 3.9: CLAM Processing Composite (taken from [4])

frameworks such as Weka (written in Java). However, apart from this there are two very important reasons to choose Torch in front of Weka:

- Weka is more focused on decision tree and classification problems than on neural networks and since our system it is totally focused on neural networks we want a framework in which they are the main feature.
- Weka stands more for a toolbench than a library or a framework, and what we wanted was a library in which we could write code clearly from scratch not taking account of the GUI's (which is another very important Weka feature) and also developing our own Machine Learning system while Weka is more focused on using their own ML systems (which are already implemented).

Another important feature is the efficiency of Torch: it must be said that it's very fast because it has been thought to be in this way. Moreover, Torch also provides a very intuitive way to program in the sense that it's very easy to understand the data strucutures and the way this framework works.
Looking at Torch in deepness we can see that there are four basic concepts (classes): DataSet, Machine, Trainer and Measurer. The concept is easy: one Machine (i.e.

MLP) needs a Trainer (i.e. descent gradient) a DataSet (containing the training and validation data) and a Measurer (which determines when the machine has finished its training). When somebody has all this, he only needs to start training the machine and Torch will do the rest. Let's see these concepts with a little deepness.

3.6.1 Data management

Remember the basic data that a neural network needs to be trained is a set of pairs of frames (one for the input and one for the target), in this sense let's see how Torch provides some data structures to hand the information we introduce to the network.

Sequence

A Torch *Sequence* is a set of vectors with the same size, each one of this vectors is called *Frame*, a sequence has a temporality mean, that is, the frames are ordered according to the time.
Sequence provides a lot of methods to add and erase frames, copy sets of frames from another sequences, etc.

DataSet

A Torch *DataSet* is a set of examples used for training a neural network, understanding an example as one pair of frames (one for the input and one for the target). As you can imagine a *DataSet* has two sequences, the input sequence and the target one.
Due to the average size of a dataset (very big) Torch provides two kinds of dataset: *MemoryDataSet* loads all the examples in the computer memory and then introduces them one by one to the network; *DiskDataSet* saves the dataset at disk an loads in the memory only the current example on which the network is working. Obviously if our dataset is very big we must use the *DiskDataSet* due to efficiency reasons, otherwise, we can use *MemoryDataSet*
Torch also provides a *DataSet* pre-processing because in machine learning it is very

common to normalize the data before training the network, notice this pre-processing is done to all the dataset in *MemoryDataSet* and only done to the current (loaded) example in *DiskDataSet*

3.6.2 Network Architectures

The base class in Torch for a neural network is the *Machine* class, this class has an output *Sequence* and method to execute the machine which modifies the output sequence according to the input one we introduce. Now we will see in a more detailed way the kind of neural network that we are going to use for the project, Torch calls them the *Gradient Machine*.

Gradient Machine

This class inherits from *Machine* and models a neural network which is trained with the descendent gradient algorithm. Each machine has some parameters (which are the different weights of the other *Gradient Machines* connected to it).
At this point, it should be noticed that Torch stands for *Gradient Machine* what we stand for the concept of *neuron*, in order to create a full neural network Torch provides the *Connected Machine* class, which is a set of machines (neurons) connected among them.

3.6.3 Training Process

The training process is done by objects called *Trainer*. Each of these objects need a *Connected Machine* (Neural Network) and a stop criterion for the training process to be finished, a criterion is nothing more than a function cost. Torch provides these criterions:

- *MSECriterion*: the error function is the mean-square error and this is the function that our perceptron will minimize.
- *ClassNLLCriterion*: focused more on the classification problem rather than on the function fitting problem. It maximizes the negative log-likelihood.

- *WeightedMSECriterion*: the same as the *MSECriterion* but ponderating each example according to its importance.
- *MultiCriterion*: Ponderated sum of several criterions.

We would like to conclude this section describing the different parameters of a *Trainer*, these are:

- End Accuracy: difference between the previous error and the current one, the training process will finish when this difference is less than the end accuracy.
- Learning Rate: controls how fast the network learns.
- Learning Rate Decay: the more iterations we do to the network the more the Learning Rate value decreases.
- Maximum iterations: determines the maximum number of epoches that the network can process.

Chapter 4

Concepts on Frameworks Integration

4.1 Introduction

This chapter will focuse on general theoretical concepts on frameworks integration. These aspects have a lot of relevance for us because in the project an integration between two frameworks has been done, concretely between CLAM and Torch.
On the first part of the chapter we will discuss some general ideas that any integration should have and the on second one we will foaudio signal generated by the neural network, which should be very close to audioIn.cuse with a little deepness in concret concepts on the integration between CLAM and Torch.

4.2 Basic Concepts

This seciton is about introducing some very basic initial considerations which should be given by any integration between frameworks; those conditions are based on the fact that a user from framework A (in this case CLAM) does not have to know about how the other framework works (in this case Torch), this user only has to know about the features of framework B.
Avoiding any inconvenience or confusion we will call the framework A as the "main"

one and the other will be called the "integrated" one.
Below are listed the main conditions that should be kept by in any correct integration:

- A user from A is able to access B in a transparent way; as we have said before, the pogrammer from A does not know anything about framework B internals.
- Framework A users can not call directly any method from framework B; the integration should provide an interface or a facade in order to let the A user invoke those methods.
- A user from A can not use any data structure which belong to B; in the same way as the last point the integration should provide a data structure conversion module between the two frameworks.
- All the data structure conversions should be done in an implicit way, which means that the framework A user does not have to do any conversion in his code because the integration does it.

All those points have a very important consequence that must be kept in mind: the integration will restrict the framework B functionalities, that is, the integration programmers decide which features from B will be integrated and which not. In this sense one must be very careful in order to offer to A framework user's the highest possible features from B.

4.3 Integration between CLAM and Torch

This section will focuse on some concrete aspects about the two frameworks and how should we handle them in order to reach a good integration.

4.3.1 Data Structures

As we have seen on the *Tools and Concepts* chapter there is a very important data structure in CLAM called *ProcessingData*, one of its most relevant children classes

is probably *DataArray*, one can imagine this class is very close to *std::Vector* and obviously it has similar functionalities.
Inside Torch the most important Data Structure is the *Sequence*, this class implements a set of ordered examples that will be introduced in a neural network and it's used in all kinds of Torch trainers.
Due to the importance of the classes named above and because of the fact that usually a CLAM user is going to train the networks with CLAM data, it is obvious that the integration should have some kind of conversion between CLAM *Processing Data* and Torch *Sequence*. One can understand the *Sequence* as an average matrix, thus apart from the CLAM data we also need to know some features of the neural netwok that will be trained such as the inputs and the number of examples, we don't need anything else for filling the Torch *Sequence*. Let's see a very small example in pseudo-code of a conversion between these Data Structures:

```
DataArray2Sequence(CLAM::DataArray data, Torch::Sequence seq, integer n_inputs,
integer n_frames)
{
     Create new Sequence with n_frames frames and n_inputs inputs;
     for i=0 to i=n_frames
          for j=0 to j=n_inputs
               data[i*n_inputs + j] = seq[i][j];
          end for;
     end for;
}
```

Another very important data strucure in Torch is the *MemoryDataSet*, which stands for a data set fully loaded on the computer memory, in order to fill this data set Torch has a formatted file in which CLAM data structures must be converted; the Torch dataset format is in the next way:

```
[number of frames] [number of inputs  +  number of outputs]
[1st input frame] [1st target frame]
[2nd input frame] [2n target frame]
.
```

.
.
```
[Nth input frame] [Nth target frame]
```

As one can see, it is a very simple format in which only the number of rows and columns of the matrix (dataset) must be in; later in the *MemoryDataSet* constructor the programmer specifies how many inputs and outputs there are.
Maybe a conversion between CLAM data and this format is a little harder than the last code example but is still very clear, one only should to take in account that the first file line is very important because it determines the "amount" of file that Torch is going to load in its *MemoryDataSet*

4.3.2 Functionalities

Here will be explained what Torch functionalities should be implemented in CLAM, first we will define these features and then we will talk about how CLAM should do it. The basic steps of building a MLP (Multi-Layer Perceptron) are:

1. Building the MLP: number, size and kind of layers must be defined.
2. Loading the training and validation data.
3. Defining MLP parameters such as learning rate, maximum iterations and convergence accuracy.
4. Defining the error function.
5. MLP training.
6. MLP testing (once is trained)

Therefore CLAM should offer at least all the functionalities described above beacuse these are the minimum requirements for using an MLP in a correct way. Let's see some details about each point.
On MLP building Torch offers a lot of kind of neurons, CLAM should implement al least the *lineal*, *tanh* and *sigmoid* ones. On the data loading the conversion

functions described below should be used because we must have in account that the programmer has only CLAM data. When defining the MLP parameters those are the minimum and more important, apart from these, Torch offers also the *learning rate decay* that could be implemented too. Obviously, apart from all this CLAM user must be able to train the perceptron.
Once our MLP is trained, it is very important to provide the CLAM user a method to use it, in this sense conversions functions are very suggestive again because the perceptron only understands Torch data.

4.4 Mapping the CLAM *Processing* concept into Torch

As we have seen on CLAM there is a very important concept, the Processings. This concept could be easily applicated to the neural networks and in extension also to Torch; we want to apply it due to this project nature, which has lots of entities that "do some process". What we are going to do is to prpose some mapping among the concept according to CLAM's frame and the different parts of our neural network.
Let's remember that a Processing is composed by: input and output data, some controls, one algorithm to execute, some kind of configuration for the Processing Object and the internal state. Looking at a standard neural network we can do our mapping as follows:

- Data Input: corresponds to the neural network inputs.
- Data Output: corresponds to the neural network outputs or targets.
- Configuration: corresponds to the neural network parameters, for example: the learning rate, the stopping acurracy, the number and kind of hidden layers...
- Algorithm: corresponds to the way in which the network updates its wheights, for example: the Back-Propagation algorithm.
- Internal State: it could be obvious, but a network can be trained, training or not trained.

In terms of controls, a neural network is not very "controllable", I mean, we cannot modify anything when it is in training phase, for example: as one can do with a CLAM processing, one can not change any neuron wheight when the network is training.

Chapter 5

Emulating a High-Pass Filter

5.1 Introduction

Why a high-pass filter? Before starting any kind of learning which involves the effect that we should study if an average neural network is able to learn a simple "effect" (a high-pass filter) which has the advantage to be linear. A high pass filter basically cuts off all the spectrum frequencies which are below a certain threshold, and do not modify any frequency above.
We can see on figure 5.1 the frequency response of this filter, it can be seen for what number each frequency is multiplied; it should be noticed that the low frequencies are multiplied by a number close to zero (they are erased) and the high frequencies are multiplied by a number close to 1, which means that they will not be modified.
As I have said before, we have chosen this kind of filter because it can be considered as a very simple effect which is *a priori* easy to learn by a neural network. Once we have finished this experiment we will be able to learn the guitar effect pedal we want. Obviously if this experiment is not successful it will mean that if our network cannot learn this simple effect it will not be able to learn a real guitar effect, which are more complicated that this simple filter.
Before starting the experiment we think that our network is going to learn this kind of effect because of its linearity and considering the fact that we are going to use a network architecture (the Multi-Layer Perceptron, explained on *Tools and Concepts* chapter which is able to learn any kind of non-linearity between the inputs and the

outputs.

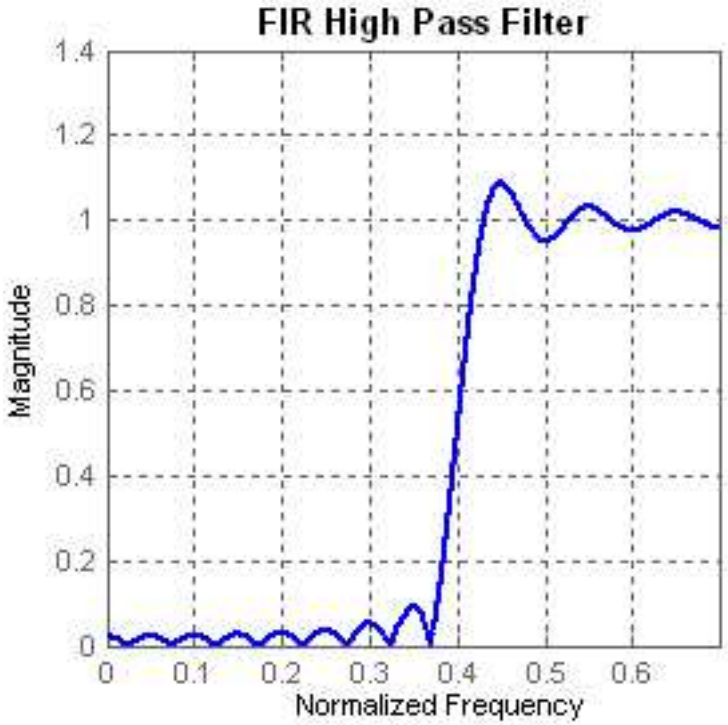

Figure 5.1: High-Pass Filter Frequency Response.

5.2 Training with "utopic" white noise

Now we should decide with which data we are going to train our MLP. Obviously a filter must be able to "understand" all the possible frequencies, this means that a filter knows what it has to do for all the possible frequencies we introduce to it. This means that, in order to train our network in a correct way, it must learn the filter response for all the possible frequencies in our spectral range (from zero to half sampling rate).

Taking in account the last considerations we decided to train our network with some kind of artificial white noise (see figure 5.2) which contains all the frequencies on each frame.

The white noise has been said artificial because what we really introduce to the

network is a totally flat spectrum (for the inputs) and the resulting spectrum from applying the high pass filter (for the outputs).
On figure 5.3, itcan be seen one target frame while its corresponding input is just a frame which contains all zeros. Apart from this frame we also have introduced to the network the same pairs of frames but atenuated for 0.1 db until the input reaches the -90 db. Thus, we have trained the network with 900 examples; so the second input frame will contains all 0.1's while its corresponding target will be the same as figure 5.3 substracting 0.1 to each value.

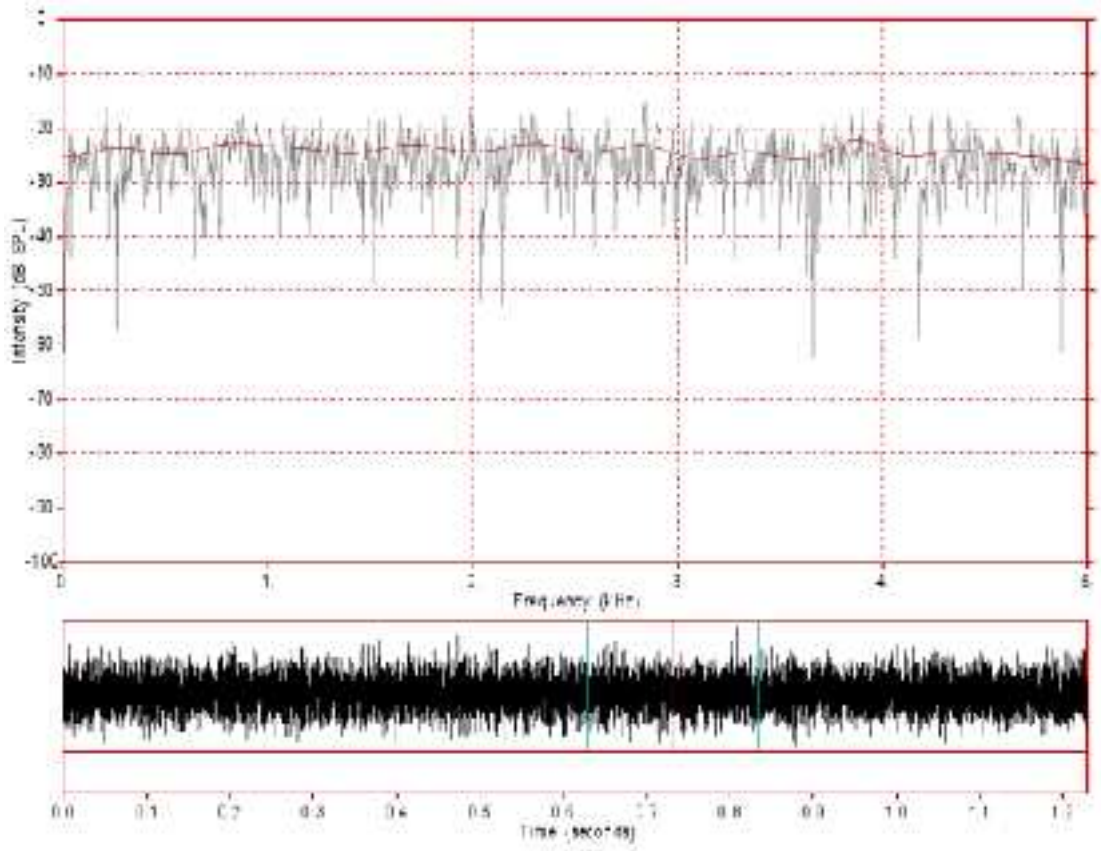

Figure 5.2: White Noise Spectrum and Wave (taken from [7])

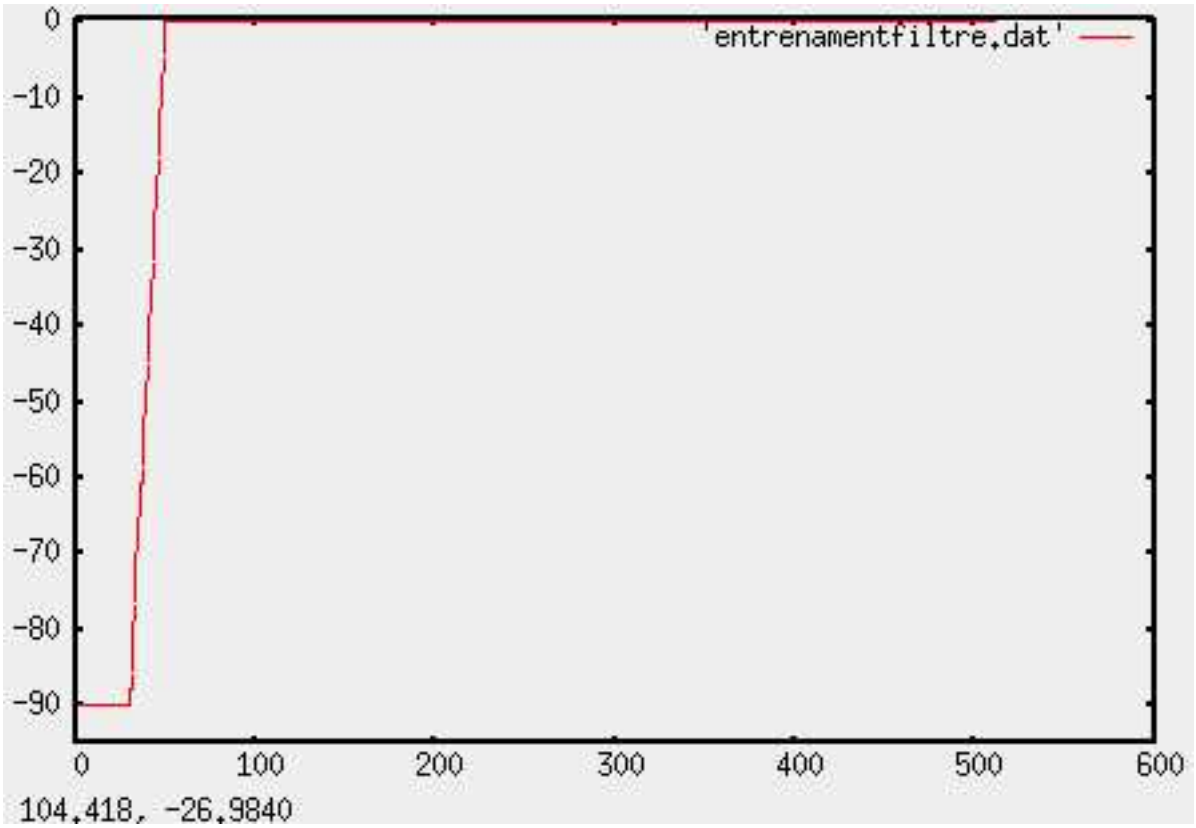

Figure 5.3: Target frame for the neural network. Y axe: Magnitude; X axe: Frequency in samples, not Hz.

5.2.1 Results and Conclusions

Now that we have trained our MLP we will check if the training has been successful, it will be done by introducing some spectrums to the trained network and testing if the results are the desired ones.

First Test: flat spectrum on -40 db

The first test is very simple, it consists on introducing to the network one frame which contains a flat spectrum on -40 db. Since one of the training frames was exactly this the network is expected to produce a succesfull result.
The output can be seen on 5.4 and as we can observe it is a very good result because

the network has returned the expected spectrum shape with the correct attenuation (-40 db).

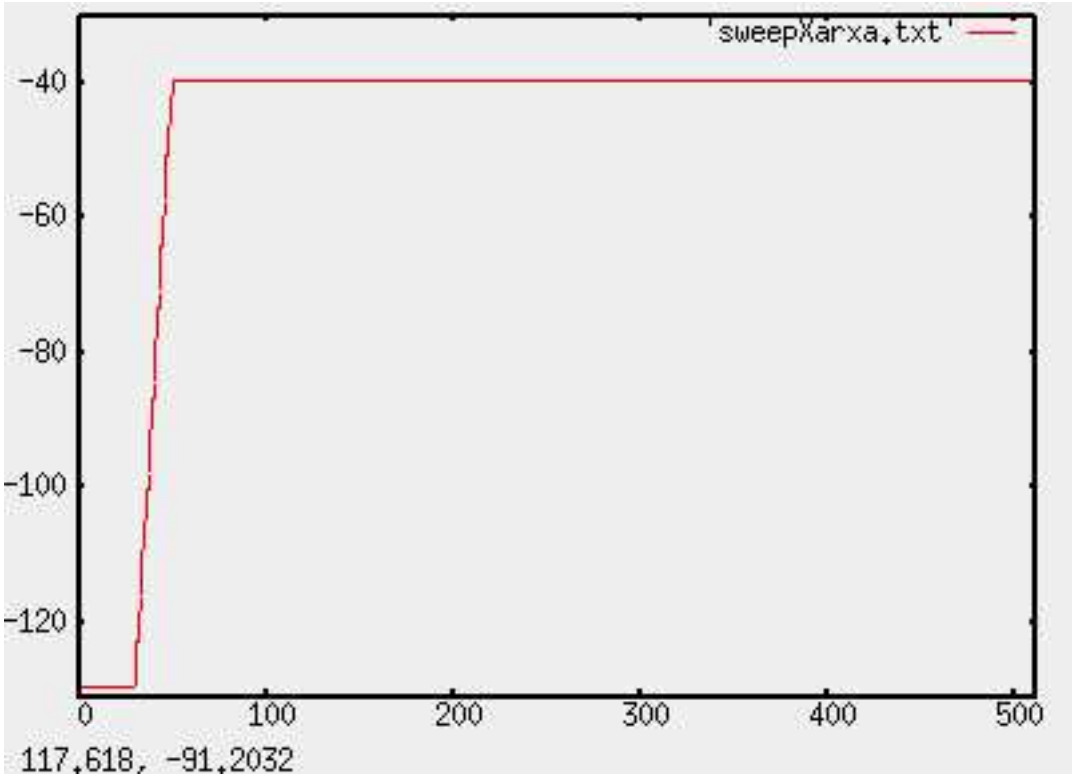

Figure 5.4: Results for the first test. Y axe: Magnitude; X axe: Frequency in samples, not Hz.

Second Test: non-flat spectrum

The second test is a little bit harder than the first one, now we are going to introduce to the network a frame which has the half of its samples on 0 db and the other half on -20 db (see figure 5.5).
Remember that our trained perceptron has never seen this kind of frame an due to this fact it is possible that the output would not be the desired one, obviously the desired output is the one which is in the same way that in the first test but with some amplitude on 0 db and the some amplitude on -20 db. Once we have runned the network with this spectrum (figure 5.5) we can observe that the result is not

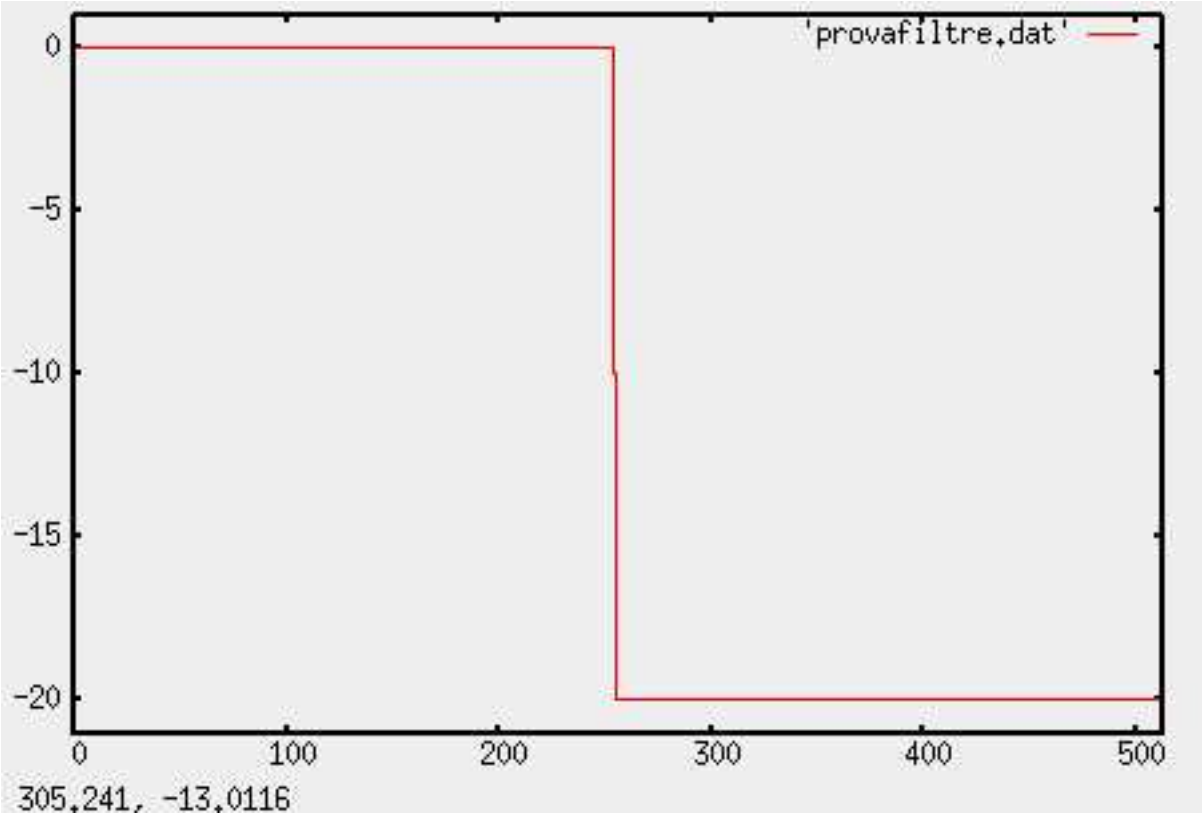

Figure 5.5: Spectrum introduced to the network. Y axe: Magnitude; X axe: Frequency in samples, not Hz.

the desired one (see figure 5.6), the resulting spectrum has its amplitude on -10 db instead of having some in 0 db and some in -20 db. *A priori* we don't know which is the cause of this effect, but hope that some experiments in the future will bring us some light over this point.

Third Test: white noise

The third test consists in introducing to our trained network some white noise, remember that we have trained the perceptron with some "utopic white noise" in the sense that is totally flat, due to this we expect the network to return a good result. After running our network we can listen to the sinthesyzed audio (have not trained the phase but this filter does no alter it, so we use the original one) and the result is

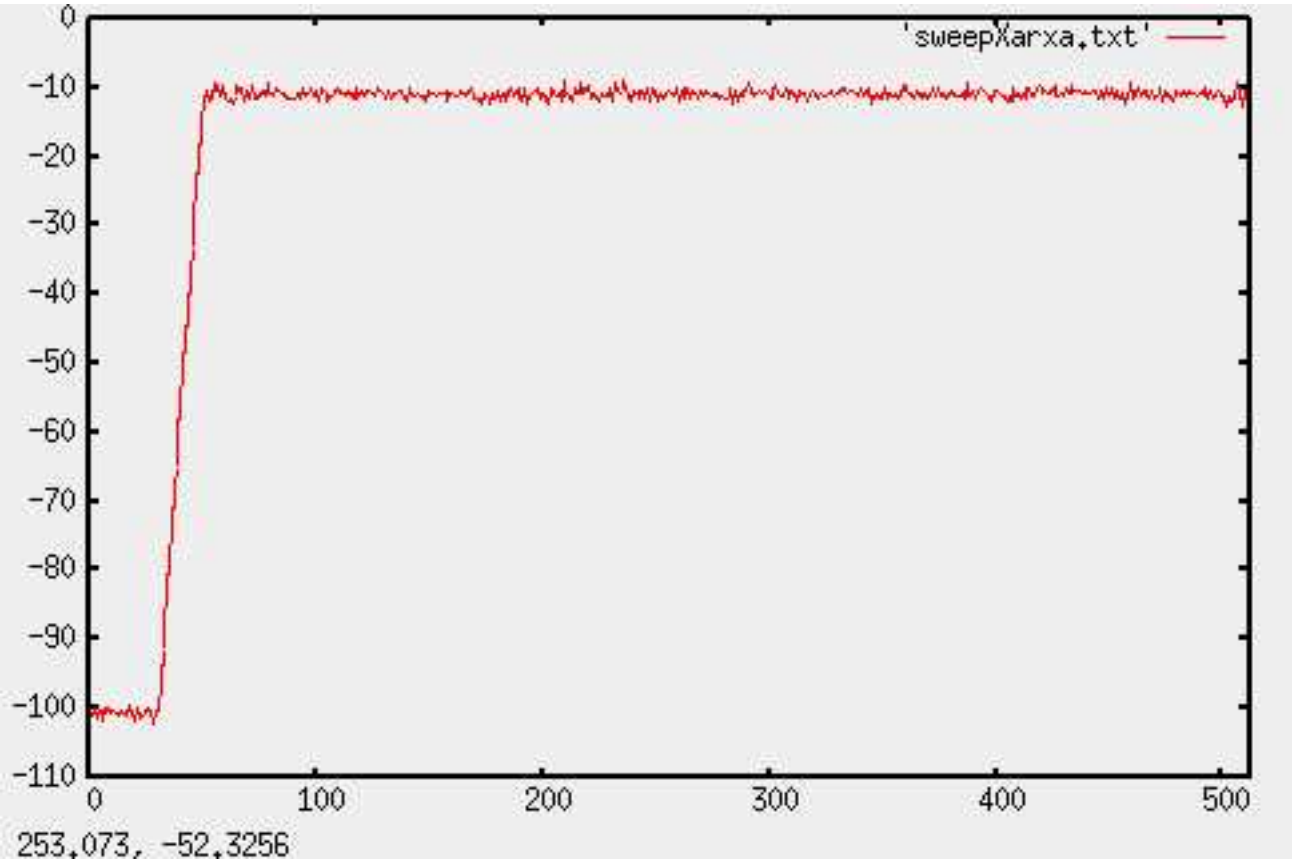

Figure 5.6: Results for the second test. Y axe: Magnitude; X axe: Frequency in samples, not Hz.

good in the sense that our perceptron has applied the high-pass filter to the signal in a correct way.

5.3 Training with "utopic" sweep wave

The last section was about training our network with some kind of white noise, remember that the results were good while we were passing white noise to the network, however, when passing a sweep wave the results were very very bad. Due to this fact we decided to train our network with not exactly a sweep wave but with some kind of "utopic" wave.

This kind of wave (also called chirp wave) is essentially a sine function which changes its frequency during the time, that is, it "sweeps" all the possible frequencies between two given ones. A sweep wave spectrum (from one frame) stands for a tipically sine wave spectrum, with the difference that the spectrum main lobe changes its position on each frame due to the frequency change.
On figure 5.7 we can observe a chirp wave spectrogram, the red parts indicate the spectrum peaks and one can see how the pitch has increased in time. Let's talk now

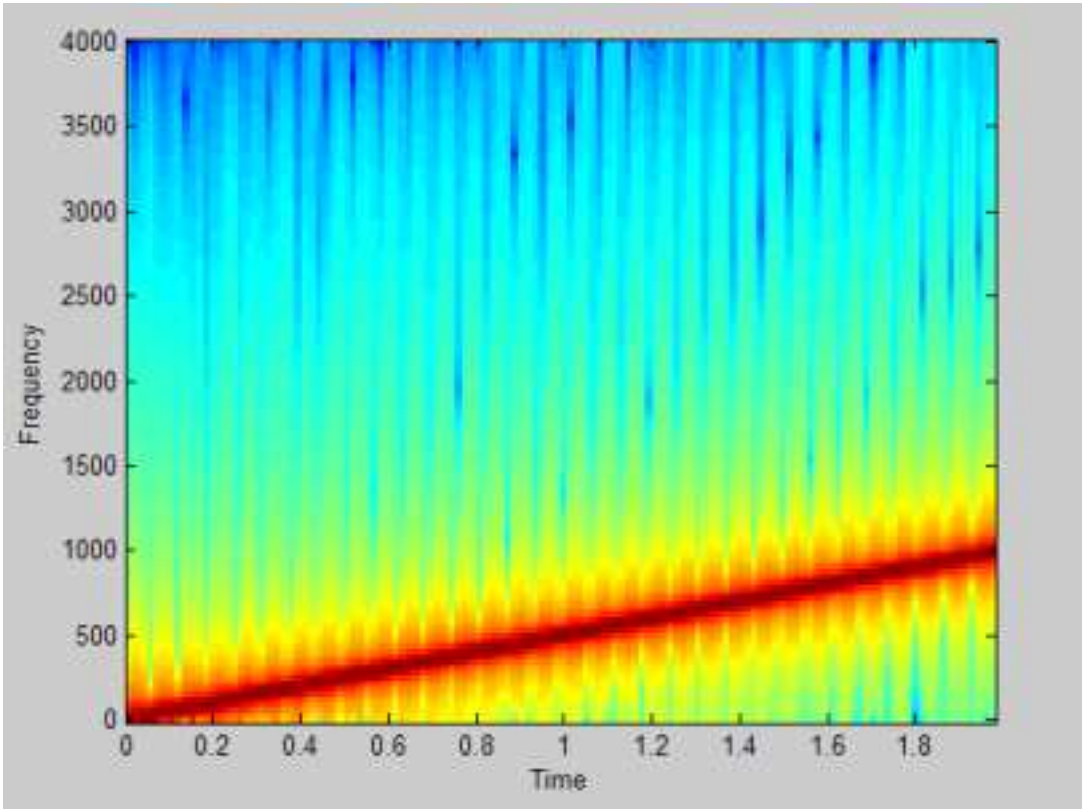

Figure 5.7: Chirp wave spectrogram

about the training, if we look at this section title we will see that we are talking about "utopic" sweep wave, utopic in the sense that we are going to generate the spectrums "from scratch", it means that they will not be real spectrums but will have their main features. We have constructed the training spectrum as follows:

1. We have extracted the main lobe (9 samples) from a simple sine spectrum windowed with a BH92.

2. Once we have the lobe constructed the input frames are generated passing this lobe through all the samples and keeping the ones which has not lobe setted to -90.0 db. Example: first frame will contain 9 samples of lobe followed by 504 samples with value -90db; second frame will contain 1 samples on -90 db, 9 samples corresponding to the lobe, and finally 503 samples on -90 db; etc.

3. For the target frames we just have taken the each input frame and have applied to it a high-pass filter response from figure 5.3. Obviously the first target frames will contain only values on -90 db while the last ones will remain exactly as its corresponding input frame.

In the same way as the "utopic" white noise we have trained our network with different spectrum amplitudes, concretely in this case we have used 100 amplitudes, from 0 to -10 with a 0.1 step.

5.3.1 Results and Conclusions

First Test: A real sweep wave

The first test is about passing a real sweep wave to our trained network, the only modification that we have done to the network is that we have introduced a threshold in which any value below -90 db is setted to -90 db. A priori the network results should be good because of the similarity among the training and test data, however let's see some plots to evaluate the network response.
The first plot that we are going to present is a comparison between the correct results that the network should give and those that the network really gives. What one can see on figure 5.8 is the maximum value of each sample for all the frames. The red line is the correct one and the green line stands for the network response. There's a very important thing to notice, the network's response has some oscillation which probably will cause some kind of problems when synthesizing, apart from this problem, it should be noticed that the network has learned how to filter, and this is a very good new.
Let's see the problem related above in a more detailed way, now we are going to zoom the top spectrums line in order to see the oscillation that we have related before. On figure 5.9 we can see how hard the oscillation is compared to the original

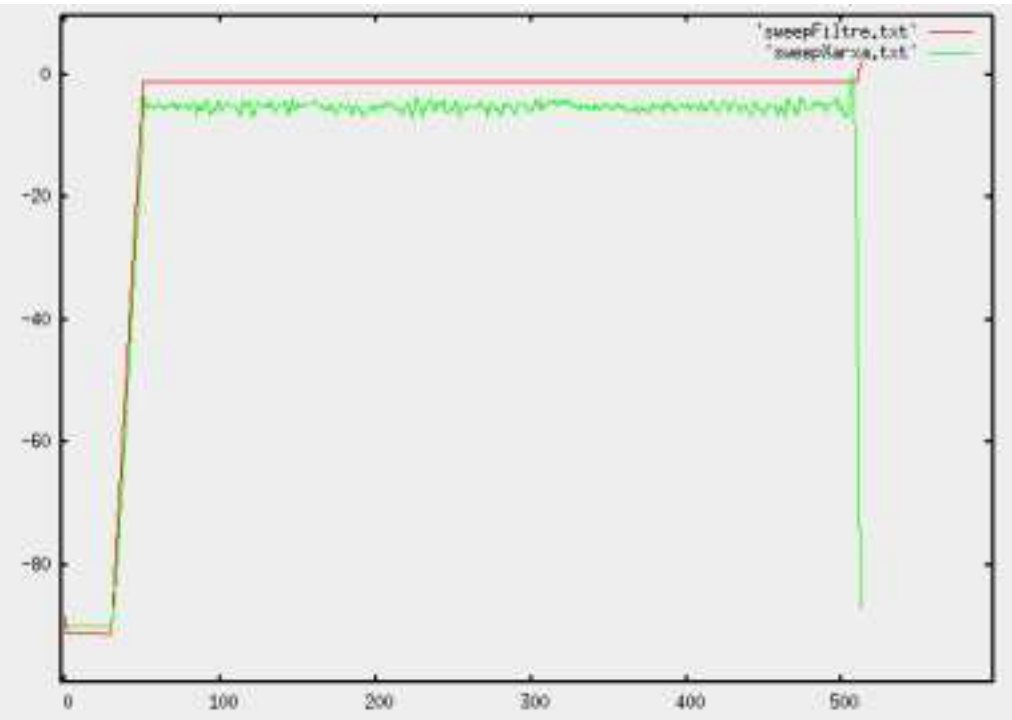

Figure 5.8: Comparison among the network response and the real response. Y axe: Magnitude; X axe: Frequency in samples, not Hz.

sweep. When listening to the synthesized audio using as a magnitude spectrum the one that gives the network, one can notice some noise, we think that probably it should be caused by the oscillation related before. Despite this noise factor the high-pass filtering is done in a correct way.
One thing we have not talked about is the fact that the networks give the peaks lower than the correct response, this is not a problem in the sense that what really matters is the shape (which is not totally correct) instead of a little attenuation on the magnitude.

Second Test: A white noise wave

The second test will introduce to the network some white noise in order to see how it performs. The only modification we are going to do on the input spectrum is to increase some db's for it to reach the level of db's in which we have trained the

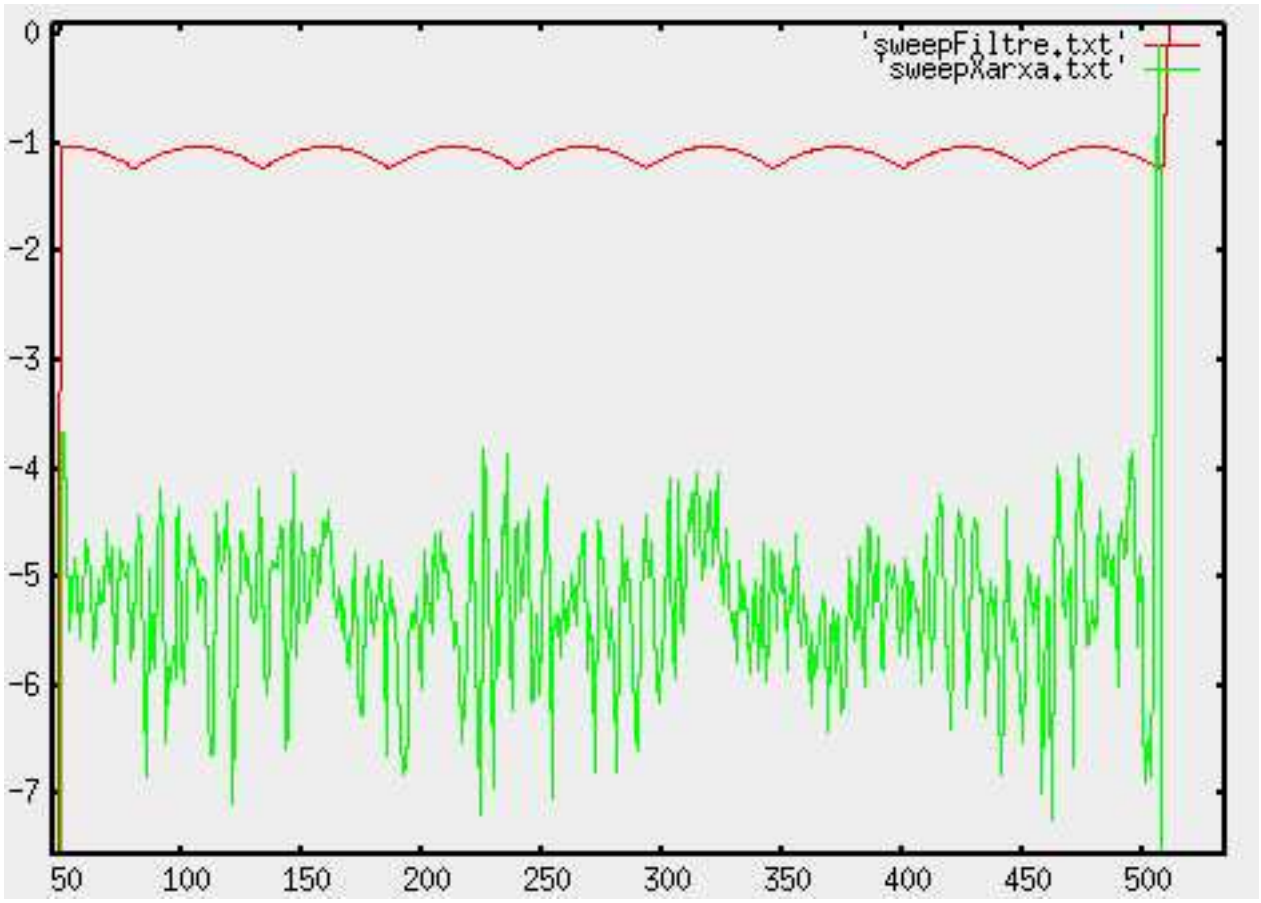

Figure 5.9: Comparison among the network response and the real response. Y axe: Magnitude; X axe: Frequency in samples, not Hz.

network. As in the prior figure, the red line represents the correct response when filtering white noise and the green line represents the network response
As we can see on figure 5.10 the result is very bad because the network does not seem to apply any kind of filter to the input. It can be seen that the network attenuates in a very subtle way the low frequencies (don't care about magnitudes, just mind the shape) but not enough to consider that it performs the high pass filter in a correct way.

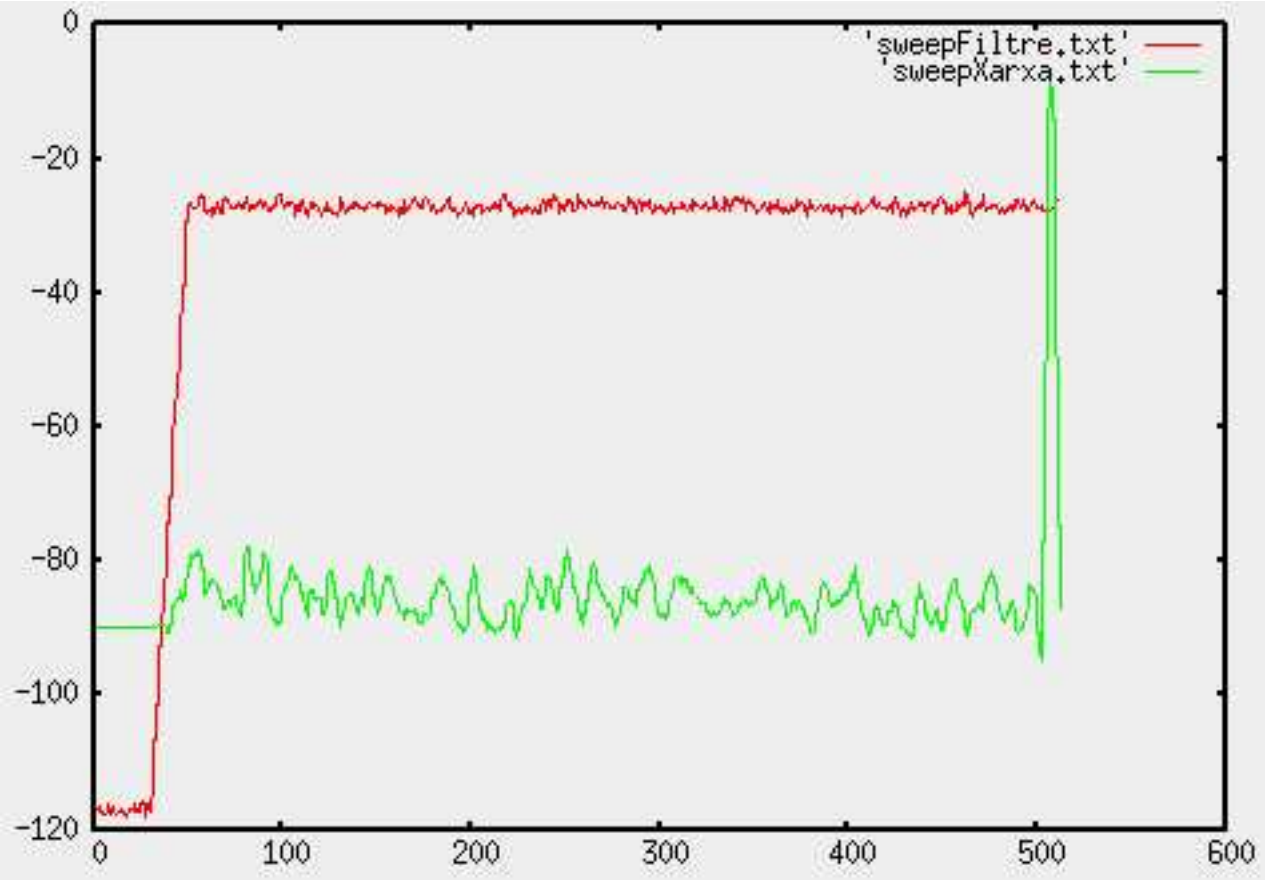

Figure 5.10: Comparison among the network response and the real response. Y axe: Magnitude; X axe: Frequency in samples, not Hz.

5.4 Conclusions on high-pass filtering

Let's remember a little bit what have we developed on this chapter: our goal was to train a neural network for it lo learn a simple high-pass filter; thus, if the network was not able to learn it trying to learn an overdrive pedal will not have any sense.
We have trained the network both with non-real (utopic) white noise and sweep wave. We have done this because we do not know which of both options is better to train the network in terms of generalization; because obviously we want a priori our network to filter any kind of signal.
However, the results of the experiment are not as "exciting" as we expected because the network only works well if we introduce it signals which are close to the training

ones, so the principle that the network should work with any kind of signal has no sense at this moment, therefore we should train and test the network with similar signals for it to perform in a correct way.
Next step, we will be trying to emulate the pedal, obviously it will be harder than this experiment but also will be focused on the guitar sound, in this sense we are going to train the network with real guitar sounds because we have seen that it is very hard to train the network in terms that it will perform its operation to any kind of input signal.
Apart from all this there is another very important thing, we must decide if we are going to train the network on the spectrum domain or on the time domain. In this experiment we have seen that in the spectral domain the network is not able to learn a single high-pass filter (which is linear). Thus, it is not very probable that our network could learn an onverdrive effect, which obvoiusly is not linear and also affects the phase.
In this sense we think that it could be more suggestive trying to train our network on the time domain, which is less complicated than on the spectral one; next chapters will relate how this training has been done.

Chapter 6

Guitar Effect on time domain

6.1 Overview

Let's change our approach in effect learning and let's go to the time domain, we have seen on the last chapter a lot of problems when learning a simple filter from the spectral domain; thus, we are going to try to learn the effect pedal (not the high-pass filter) directly from the time domain. Obviously, now any kind of transformation must be done due to the fact that we are going to use directly the audio (the wave values for each sample) to learn our desired effect.
It must be said that before starting this experiment we need some kind of guarantee that we have the possibility to get some good results. In order to get this guarantee we have done the experiment explained on the last chapter but on time-domain. It's important to notice that the results have been very good ones and have encouraged us to keep on the time-domain approach. Now we are not going to explain it in a detailed way because the method is exactly the same one that we are going to use and explain on this chapter, except (obviously) for the data all the other parameters are exactly the same.
Our approach is based on the work done by Wan and Nelson called *Networks for Speech Enhancement* taken from

`http://cslu.ece.ogi.edu/nsel/wan_manuscript/wan_manuscript.html`

this approach uses a neural network to denoise any kind of speech signal. In a few words what they do is to show to the network noised speech on the inputs and the same speech (but clean) on the targets, in this way the networks learns how to denoise any kind of noised speech signal. It is very important to say that all the data showed to the network belongs to the time domain (see figure 6.1).
According to this work a neural network is able to learn a denoiser directly from the time domain, therefore we think it will also be able to learn an overdrive effect pedal because a denoiser is not a simple filter as a high-pass one; in this sense we are going to try to model the pedal transfer function directly from the time domain in which the data will be some real electric guitar recording.

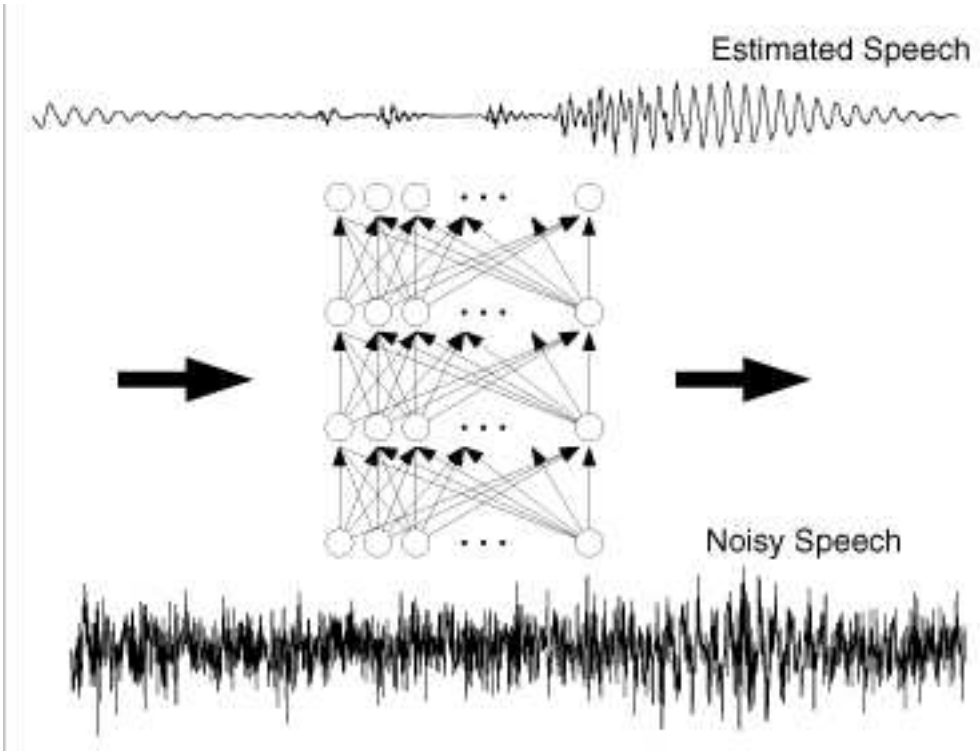

Figure 6.1: Wan and Nelson method for denoising speech

6.2 The Training Process

As mentioned before our dataset is composed by some musical piece fragment (concretely the main riff from *Rock and Roll* by Led Zeppelin) in two ways, one recorded directly from the guitar and the other one processed from a pedal effect, for this test we have used the *Ibanez TubeScreamer* which delivers a natural tube overdrive to the input signal. Obviously the processed effect will be the network target while the "clean" guitar will be the input.
The first question that came to our minds was what should be the correct frame size to introduce to the network, we think that the minimum size should be 512 samples (aprox. 10 ms.) because it's the minimum amount of time in which one can get some resolution; however the work from Wan and Nelson proposes a frame size of 80 samples (only 1.4 ms.!!!). Due to this fact, we are going to train the network first with a frame size of 512 samples and then with 60 samples. A priori we don't know which size will work better, we will know it from the testing that we are going to do right now.
Another important question more focused on the synthesis is the hop size in order to get a good frame continuation. For this experiment we have chosen an standard hop size that will be half of the frame size.
Once the frame and hop sizes have been chosen let's explain now how the training process will be done: it is very simple, we are going to take as inputs the "clean" guitar frames (in the same way as the noised speech) and as our target the corresponding overdriven guitar frames (as the clean speech); we are going to use the same architecture we have been using during all the project and the same learning method (all explained on the *Tools and Concepts* Chapter). One last detail to notice is that all the networks layer will have the same size as the frame (as proposed on Wan and Nelson work), see

`http://cslu.ece.ogi.edu/nsel/wan_manuscript/wan_manuscript.html`

Thus, our network will have one input layer, two hidden layers with sigmoid neurons and one output layer, all of them with the same size as the frame size.
When reconstructing the signal we are going to do an standard overlap and add a process in which the resulting frames (those that the network gives once it is

trained and we introduce it some data) will be overlapped after multiplying them by a triangular window.

6.3 Testing the Trained Network

Once our network has been trained we must now evaluate the results with special tests in order to measure the network performance. We are going to introduce to the network the same training data to evaluate if it has learned it correctly and then we will introduce the same data but with some silence at the beginning, obviously this last test that will be the done is the one which is going to tell us the network effectiveness.
If any of the prior tests is succesfull we are going to introduce to the network another guitar recording totally different from the used on training. With this experiment we want to know if the network has learned just "the song" or if it has learned the pedal, that it is, obviously, what we want.
At the end of this section we will do some kind of "definitive" test, its first step will consist on introducing to the network the same music fragment for which it has been trained but recorded with a different guitar; if this experiment is successful we will do the second step which consists on introducing to the network some music that the network has never seen and, what's more, recorded with a different guitar. If this experiment becomes successful or "not so bad" (remember our data set it's not rigorous at all) we could think that our network has learned the effect in a correct way.

6.3.1 Testing with with 512 samples frame size

Introducing the same training data

Results are good but some kind of little noise can still be heard, however the global result is not bad at all but taking in account that the test data is exactly the training one this little noise can not be accepted and obviously the results, despite they "do not sound bad" should not be accepted; a sample can be heard on the next file:

`rockandroll_512.wav`

Introducing shifted training data

Taking in account the last result, we can not expect so much of this experiment, now the noise has augmented and the overall sound can be considered bad, just as we have expected (a sample can be heard on the next file:

`rockandroll_512_silenci.wav`

6.3.2 Testing with 60 samples frame size

Introducing the same training data

On this experiment, the results are very good, it should be due to the minor number of neurons, we think that the network could have learned in a better way, the fact is there is no noise and the sound is very very close to the original processed one (a sample can be heard on the next file:

`rockandroll_60.wav`

Introducing shifted training data

This is the more important experiment because is going to give us a lot of information about the network generality, it must be said that the results are also very good in terms that there is no noise and the overall sound is very very good; a sample can be heard on the next file:

`rockandroll_60_silenci.wav`

6.3.3 Testing a different riff with the same guitar

At this point we have seen that the network performs better if the frame size is about 60 samples, we don't know why but on the next chapter we will do some analysis

in order to know numerically this fact, trying to know if the numbers match with our perception; due to the very good results we have obtained now we are going to introduce to the trained network a guitar recording that it has never seen, this guitar recording is a fragment from the song "Tangerine" by (yes, again) Led Zeppelin and the frame size will be 60 because of the success of the prior experiments with this size. It is important to notice that this track has been recorded with the same guitar with which the network has been trained.
It must be said that the results obtained introducing Tangerine are very very good because the neural network has applied in a very correct way the pedal transformations. One thing to be noticed: the sample you will hear does not sound so good because it's not a nice idea to play open chords with a relatively high amount of overdrive, however the transformation is very good and very close to what we were looking for; a sample can be heard on the next file:

`tangerine_60.wav`

6.3.4 Testing the same riff with a different guitar

We have seen for now that the network perfoms in a succesfull way even introducing it an audio fragment which it has never seen (recorded with the same guitar). However, we want to know what happens if we introduce the same music fragment we have used to train the network but recorded with a different guitar.
So, we are going to introduce to the network a guitar recording of the same training riff but this time played with a washburn HB-35, which is a semi-hollow guitar. This means its sound will be very different from a solid-body guitar, but not necessarily bad.
The result of this test is quite good, possibly it is not the same sound as the original pedal but we have checked the network perfoms well when introducing another guitar, what's more, another very different guitar comparing it with the training one (a sample can be heard on the next file:

`rock_60_30_wash.wav`

6.3.5 Testing a different riff with a different guitar

This test is a very hard one, now we are going to introduce a music fragment that the network has never seen and also this fragment is recorded with a different guitar in which the network has been trained.
We are going to introduce to the network a recording which includes a jazz guitar riff played with the same semi-hollow body guitar as the last test and using fingerpicking technique instead of using a plectrum. We know it is not very suggestive to play jazz chords with a high amount of overdrive, however it could be very useful for us to test our network in "strange situations"
It must be said that the result is better than we expected, let's analize whith a little more deepness:

- when playing chords (three or four notes at the same time) the result is very good, network applies the distortion to the input signal in a correct way, the network has performed very well with a different guitar and a different recording played with a different technique.
- when playing single bass notes (which are very common in fingerpicking jazz guitar) the result is very bad, the note does not sound distorted but "synthesized" with none of the tubescreamer tonal features. We must remeber at this point that the network has never been trained with single notes but with 2 or more notes ringing at the same time (chords or arpeggios).

6.4 Conclusions

As we have seen on the previous section the obtained results are much better than we expected, however there are still some details to be solved in future and more serious experiments.

- The first one is the subtle whistle that can be heard on the samples which is caused due to the triangular window (we will talk about it later).
- When talking about serious training I am referring to train the network in a serious manner, this involves not just training it with a concrete guitar

recording and training it with more than one frame size in order to see which gives the better performance.

This whistle I have mentioned before is produced on the overlapping process, its frequency is exactly $\frac{SAMPLERATE}{HOPSIZE}$ and it occurs because the overlapping process does not smoothes enough the transition among frames. This will be a very important topic discussion later when talking about the optimal parameters for a good training process because we know that the lower is the hop size the smoother is the transition. Apart from this two points it must be said that at this point the results are pretty good and we expect they will be better when doing the training process in a more correct way.

Let's talk about how does the network perform on different situations in which it has been trained. On the one hand, we have seen the network works very well when introducing recording of the same guitar, remember that the network has performed correctly when introducing it a fragment of "Tangerine" recorded with the same guitar as the training. On the other hand, the network also works well when introducing some recording played with a different guitar as the training phase. But it is in this point where we have noticed a very important aspect, the network works well while we are introducing recordings which contain chords of arpeggios but there has been a problem with the single bass notes, which do not sound as good as we expected; we think it could be due the training, which did not contain single notes.

At this point we can conclude the network works well, we have tested it with two very different guitars and the results are quite good. Now we will do more analysis in order to get the optimum parameters in terms of frame and hop sizes and the training data.

Chapter 7

Parameter Analysis for a Successful Training Process

7.1 Introduction

On this chapter we are going to discuss which approach is the best one in order to train the system. In order to get this approach we are going to do some testing and measures and then we are going to conclude which is the best way to train our multi-layer perceptron.
Let's remember the most important variables on the training process:

- The Frame Size: stands for the size of the audio chunks that we introduce to the network; each layer on our perceptron will have exactly the same number of neurons as the frame samples.

- The Hop Size: stands for the number of samples that we shift our analysis window in order to get the next audio chunk.

- The Training Data: the kind of data we use on the training process, remember that on the last chapter we have used simply a song, but obviously this is not the best way to train our perceptron, so we will talk about another possibilities to perform the training.

To do this analysis we need some numerical measures that will lead us to the best approach, the ones that we are going to use are:

- Signal Energy: we need to measure the energy that our signal has won on the transformation process when passing through the network. This measure can be obtained by subtracting the original audio to the processed one (the one which has passed though the network) sample by sample.
- Signal-Noise Ratio: is the relation between the noisy and non-noisy data on some audio signal, it gives us the amount of noise taking in account the non-noisy data; standing for "non-noisy" the meaningful information).
- The lower the frame size, the lower the peak at frame rate($\frac{SAMPLERATE}{HOPSIZE}$) is, we can see that the best results are on framesize at 40 and framesize at 60. This is the measure that perceptually is best and, moreover, it is the measure reccomended on the Wan et al. work on which we are basing our project. Due to these two facts, it is suggestive to say that a good frame size should be 60 or 40 samples.
- Once we decide a frame size, we observe that the lower the hop size is the lower is the peak at the frame rate frequency. This has a lot of sense too because the lower the hop size is the higher is the smoothness between two frames. In order to corroborate this fact we will do later a more extrem test in which the hop size will be very very small.

Apart from these terms there will be a section on which the whistle problem explained on the last chapter will be analized in deepness (because the hop size affects this whistle frequency), this study will consist on measuring the peak at the whistle frequency in order to get the lower one.

7.2 The Whistle Problem

As we have said on the introduction now we are going to perform an study about the whistle that can be heard in our recordings, remember that this whistle is placed at

a frequency of $\frac{SAMPLERATE}{HOPSIZE}$. What we are going to do is to measure it and derive which is the best combination of framesize and hopsize according to the magnitude on this frequency.
The study consists on getting all the audios (the ones which contain a fragment of "Tangerine" by Led Zeppelin) that we have generated with the trained network and the measure in their spectrum the magnitude of the peak at $\frac{SAMPLERATE}{HOPSIZE}$. We have chosen different frame sizes and for each one we have done the measure with HOPSIZE at $\frac{FRAMESIZE}{2}$ and at $\frac{FRAMESIZE}{4}$.
On figure 7.1 we can see all the measures that we have done on the audios, the red line stands for the measures done when hopsize is at $\frac{FRAMESIZE}{2}$ and the green one when hopsize is at $\frac{FRAMESIZE}{4}$; the x axe stands for the frame size while the y axe stands for the peak magnitude in Db. Observing the picture we can arise two very important conclusions that we will try to confirm on the following experiments:

- The lower the frame size is the lower the peak at frame rate($\frac{SAMPLERATE}{HOPSIZE}$) is, we can see that the best results are on framesize at 12 and framesize at 20. However, those examples are perceptually very bad, and obviuosly they will not be taken in account. On the side of the perceptually-good samples the best result is obtained when having a frame size of 80 with a quarter of hop size.

- Once we decide a frame size, we observe that the lower is the hop size the lower the peak at the frame rate frequency. This has a lot of sense too because the lower the hop size is, the higher the smoothness between two frames. In order to corroborate this fact we will do later a more extrem test on which the hop size will be very very small.

Apart from the test exposed on the last figure we have done a test with frame size equal to four, obviously there was no whistle since it was on frequency 22050 and 44100 (according to the hop size) and then it could not be heard; not to mention that the quality of the resulting audio is absolutely horrible with this kind of frame size.
Now we will analize which problems should cause this noisy whistle sound, we will rule out some of them proving that they can not be the cause and we will propose others to be the real cause of this problem.

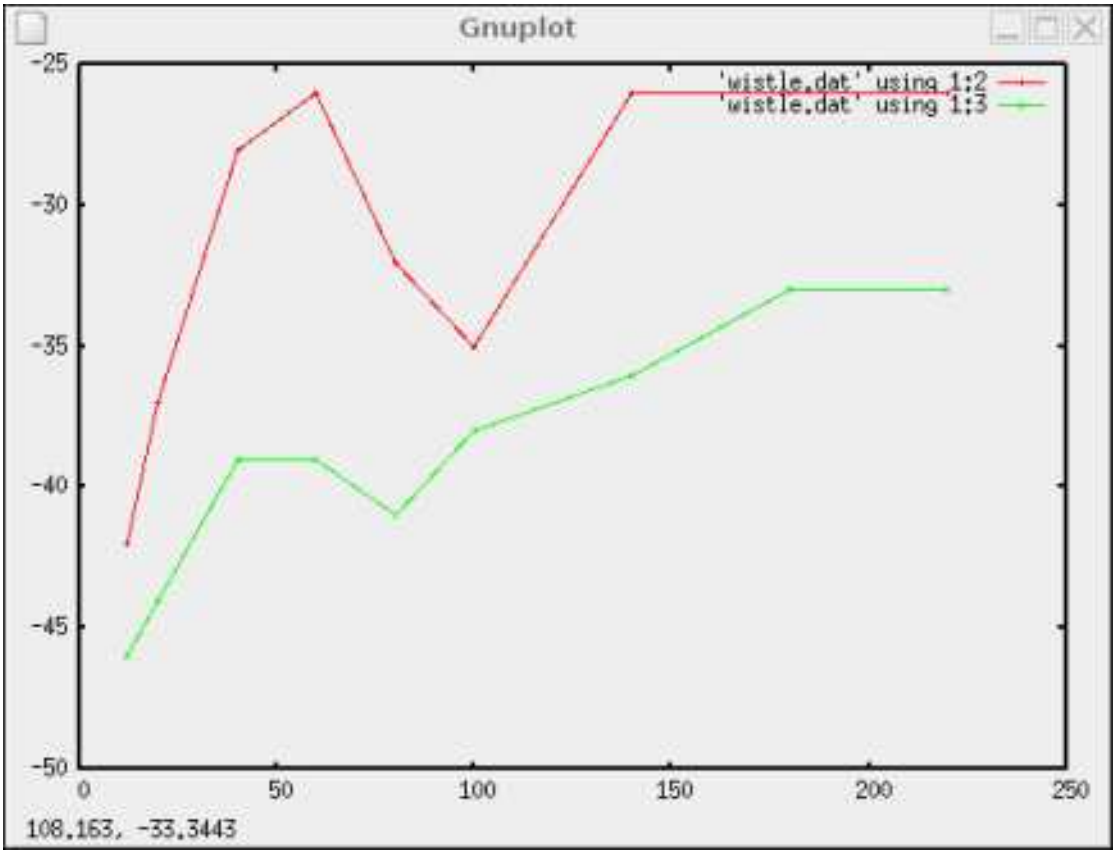

Figure 7.1: Wistle Magnitude Measuring. X axe: frame size; Y axe: Magnitude in Db's

7.2.1 The Overlap-Add Process

This subsection is about to show that the *overlap and add* process is not the cause of the whistle artifacts that we can hear in our audio samples. We are going to prove it testing our overlap-add with some signal which has stronger discontinuities than our audios; therefore, if our overlap-add process can save these discontinuities it should also save those related to our audios, which are for sure more subtle.
The concrete test will be making each network output frame to be a saw signal from 0 to 1 and then doing an standard overlap-add process with 50 per cent of hop size. It must be said that the network perfoms very well and saves the discontinuity perfectly; on this sense the network should not have any problem when saving more subtle discontinuities like the ones from our sampled audios returned by the network

shape.

7.2.2 The Phase Continuation

Maybe the whistle could be caused because the network had some problems related to the phase continuation. We are going to do a test in order to prove that our network continuates the phase in a correct way and so that the whistle is not produced by problems on this field.
To do our test we are going to introduce to our network a sinusoidal signal but having the hop size equal to the frame size, so here there will not be any kind of overlap and the audio will be formed by the simple concatenation of the network output frames.
On picture 7.2 we can see that the network continuates the phase in a good way, the little discontinuity that you can see does not matter at all because it is very subtle and does not affect the overall sound. Thus, we can conclude that the phase

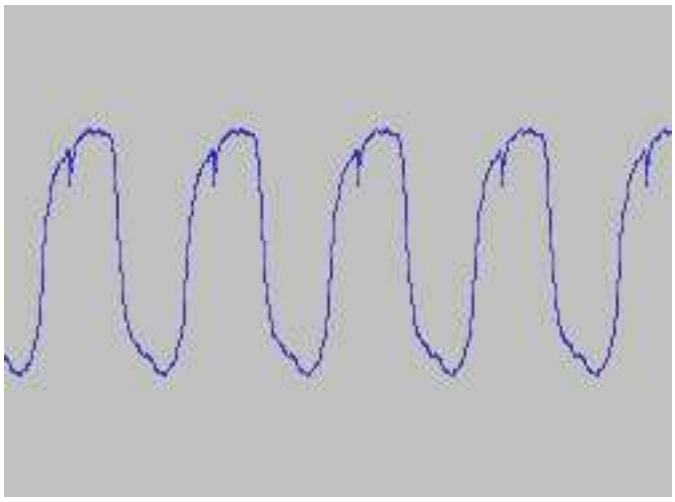

Figure 7.2: Sine Signal Continuation

continuation is not the cause of our whistle problem because even using frame size equal to hop size the phase remains correct and it does not affect the sound nor perceptually nor on the wave form.

7.2.3 The Network Ouput Energy Distribution

Now we are going to discuss one aspect that, with lots of probability, will be the cause of our signal whistle. This is about the distribution of the energy on the frame that the network returns.
We have measured the normalized mean value of each sample on the network output for all the examples and also the same measure on the training data; on figure 7.3 we can compare the energy distribution of the inputs and the outputs and it can be observed a very strong difference. This distribution should be more or less uniform (as on the input and output training data) but we can observe on 7.3 that the output has a non-uniformaity on the left side. On 7.3 we can see two measurings: the

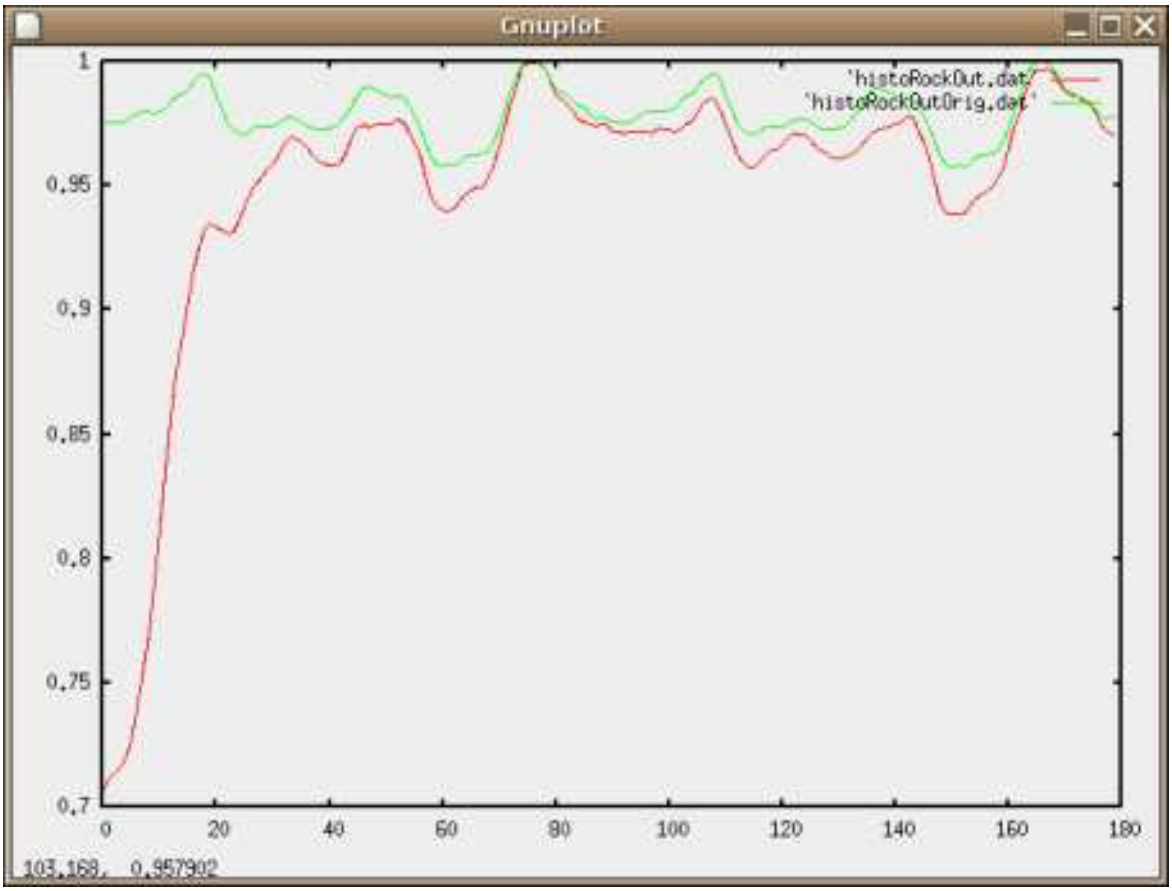

Figure 7.3: Sample Mean Measuring. X axe: samples; Y axe: normalized mean

green one is the one from the output data in which the network has been trained

and the red one is the one for the audio that the network has generated. We can see that the measure on the training data is good, that is, it is relatively uniform; however the measure done on the data generated by the network is good except for the left part.
The measures that we have done show the error difference among the network neurons, that is, the left neurons are learning somehow less than the other ones, to show it we are going to plot the squared mean error for each neuron; as we can see on figure 7.4 the left neurons have not learned anything while the other ones have done it; what's more, having 1 as a hop size with a reasonable frame size (which gives excellent perceptual results) the left neurons are still making a lot of errors. On 7.4 the green line corresponds to the neurons squared mean error when training with hop size equal to half of the frame size and the red one stands for the same training but with hopsize equal to 1. We have seen that there are some neurons that do not learn in the way they should, it could be a network problem or a data problem. In order to know this we have done the next test: we have inverted the training data and we have trained our MLP in this way, the result is that now the neurons that do not learn in the correct way are the ones that are on the right side (see figure 7.5). The conclusion of it all is that the network works perfectly but has not been trained with enough data and due to it we have not a complete range of guitar tracks in which the network could learn, therefore it produces those kind of artifacts that should be fixed with an exhaustive training.

7.2.4 Solutions for the Whistle Problem

Now we are going to introduce some solutions for the whistle problem that solve it on the perception field; those solutions have been tested and the results from them are very good, that is, they remove totally the whistle making the audio sample very good in perceptual terms:

1. Getting a low analysis/synthesis hop size: the lower is the hop size the better is the signal SNR and quality. We have tested putting the hop size equal to 1 and the result is amazing, no whistle can be heard because it "sounds" at 44100Hz and it is totally unperceptible. Remember that with 1 as hop size

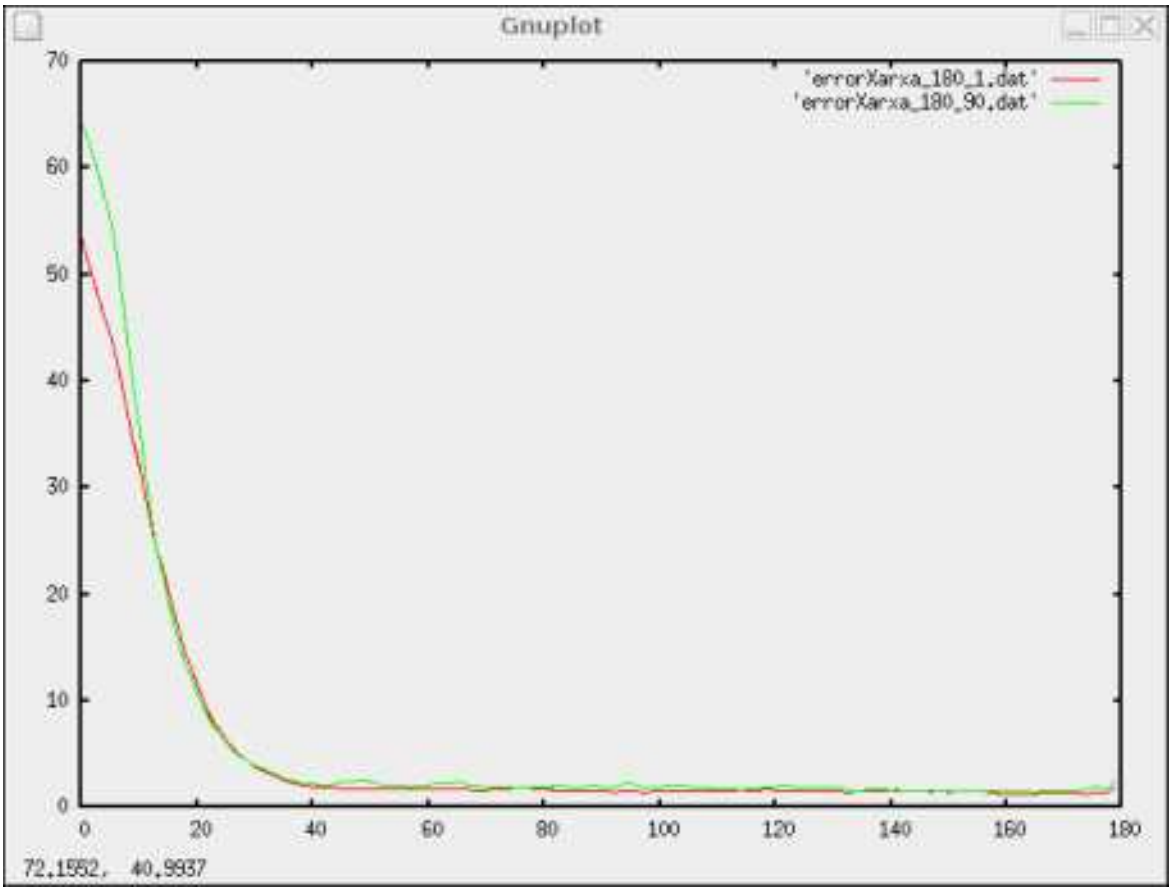

Figure 7.4: Squared Error Measuring. X axe: samples; Y axe: normalized squared error

the smoothness of the transition among frames is very hard, so it attenuates a lot (shifting to a very high frequencies) the whistle heard.

2. Standard denoising algorithm: we have introduced our signal into a standard denoise (the one which comes with Audacity) and considering the whistle as the noise, the algorithm removes it totally and the result is also amazing.

7.3 The Signal Energy

As we said before another important measure is the energy relation between the signal before entering to the network and the signal that it gives. We are interested

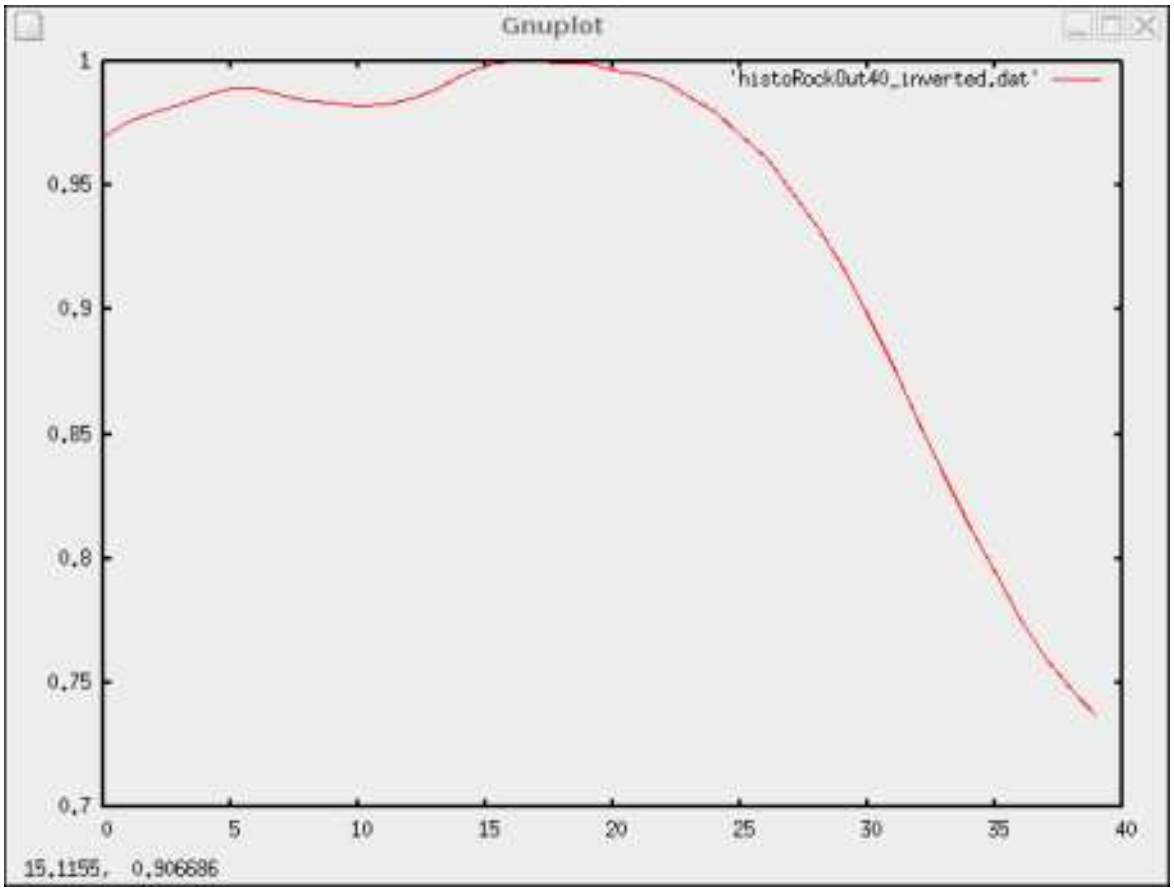

Figure 7.5: Sample Mean Measuring. X axe: samples; Y axe: normalized mean

on this relation to be the minimum as possible and on this sense we are going to do some measures comparing this energy relation among some audios which differ in their frame size and hop size, a priori our new results should concorde with the last ones.

Our main motivation for this experiment is proving that the network does not introduce any kind of gain to the signal, which means that all the distortion comes from the pedal an not from some network artefact. We are going to measure the signal energy in this way:

$Energy = \frac{\sum_{i=1}^{N} audioOut_i^2}{\sum_{i=1}^{N} audioIn_i^2}$ where:

- N: number of samples to be taken.
- audioOut: audio signal generated by the neural network, which should be very

close to audioIn.

- audioIn: audio signal in which the network has been trained, also the desired output.

Let's see now a picture (see figure 7.6) in which the different energy values are shown for each frame size and for each hop size, then we will evaluate these results in order to get some defitive conclusions.
The red line represents the energy of the frames whose hop size is half of the framesize and the green one whose hop size is a quarter of the frame size. As we can observe

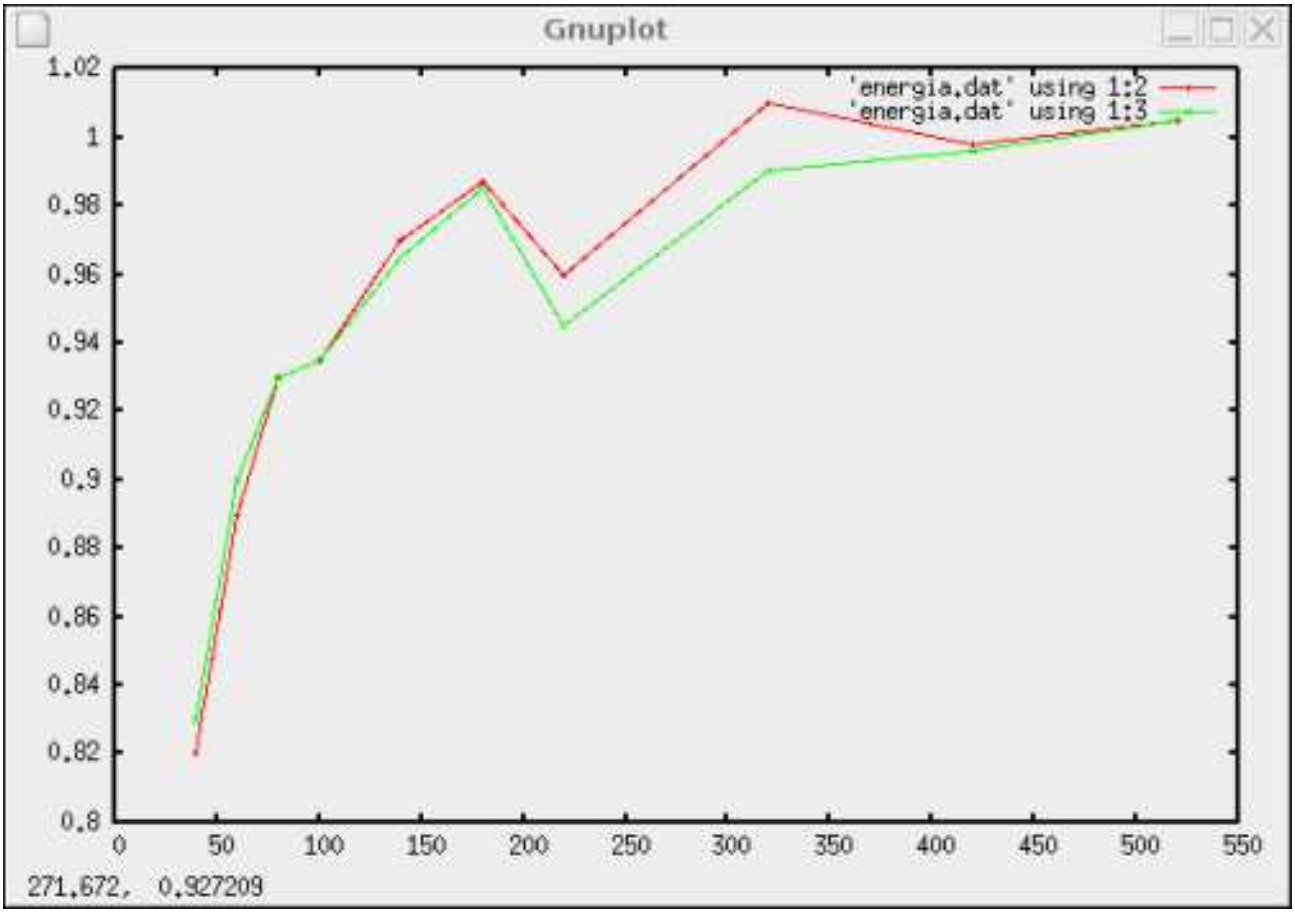

Figure 7.6: Signal Energy Measuring

on figure 7.6 the network does not introduce any kind of gain to our signal, what's more, the energy of the signal given by the network is lower than the input signal

one; therefore we have proved that our network performs well in terms of gain in the sense that it does not add any kind of distortion, which comes totally from the trained pedal.

7.4 The SNR: signal-to-noise ratio

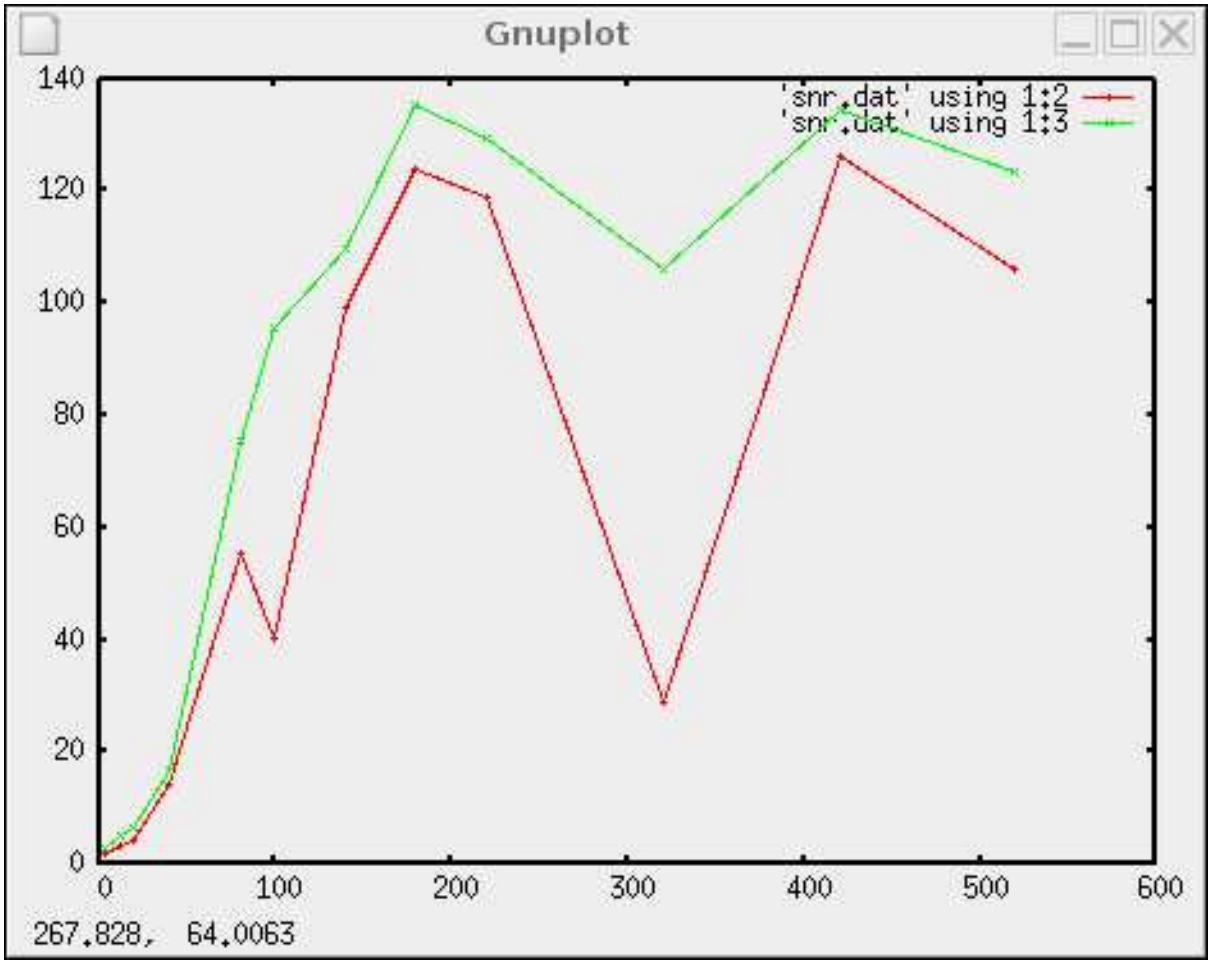

Figure 7.7: Signal-to-Noise Ratio. X axe: frame size; Y axe: SNR.

The last measure that we are going to do on our audios will be the SNR or signal-to-noise ratio; as its name indicates this measure gives us the relation between the noise and the "meaningful" information in our signal. In order to perform the SNR

we are going to use this formula:
$SNR = \frac{\sum_{i=1}^{N} audioExpected_i^2}{\sum_{i=1}^{N} (audioOuti - audioExpectedi)^2}$ where:

- audioOut: audio signal generated by the neural network.
- audioExpected: audio signal in which the network has been trained, it is the desired output.

We are interested on maximizing this measure, that is, minimizing the energy of the error signal (the denominator in our formula). This will mean that there will be the highest possible difference between the meaningful information and the noise in our signal.
On figure 7.7 it can be seen the different SNR for some frame sizes and also for two kind of hop sizes for each frame. The green lines correspond to hopsize as a quarter of the frame size and the red line as half of it.
The result is not what we have expected in the sense that we always thought that the optimum frame size was around 60 samples, but this measure, which is probably the most important, tells us that the best results in terms of SNR are obtained from a framesize of 180 samples, but also obtaining very good results with framesize equal to 420 while having very bad results on 320 samples. It must be said that the best results have always been obtained when using the lower hop size, that is, a quarter of the frame size.
Now we have chosen a framesize we are going to test which hopsize performs a better SNR for this frame size, we are going to do our tests with hop sizes equals to 90, 45, 10, 5 and 1; results can be seen on 7.8. As we can see on the last figure, the lower the hop size is, the higher is the SNR, therefore, on off-line applications it would be recommended to perform a hop size equals to 1.
Remember that with this amount apart from having a very good SNR we have completely removed the noisy whistle we heard, to tell the truth we wanted the whistle not to be percepted and in this sense we have made "sounding" at 44.100 Hz.

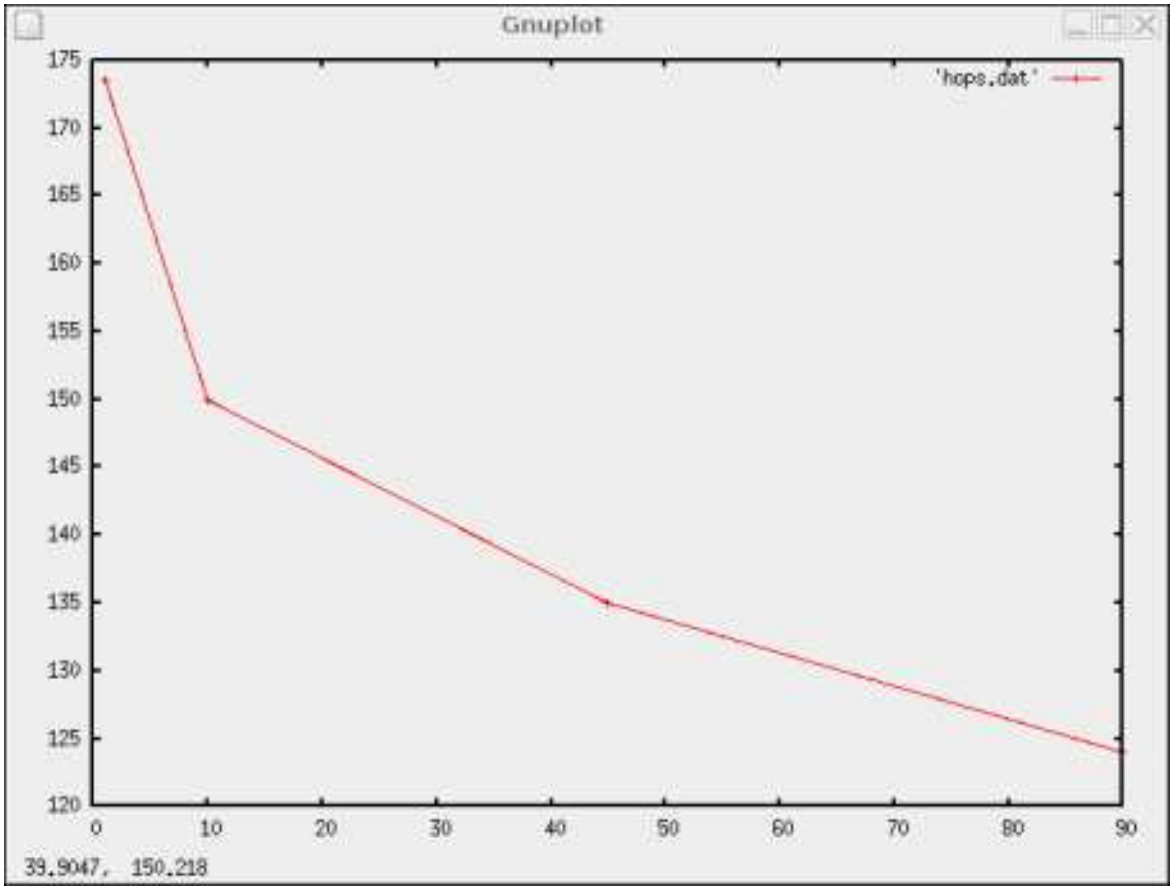

Figure 7.8: Signal-to-Noise Ratio. X axe: hop size; Y axe: SNR.

Chapter 8

System Analysis and Design

On this chapter we will explain what our system perfoms (both in terms of concepts and code), that is, we will explain the classes we have developed as well as their interactions. It must be said that our aim was not to develop a concrete application and on this sense our code has suffered a lot of changes during the process of investigation (doing some concrete experiments, testing the system in some way...); therefore, the current code is only the one in which our system works correctly and we will not include all the tests we have done.

8.1 System Overview

Now we are going to introduce a graphic (figure 8.1) which will show the blocks that compose the system, then we will briefly explain each of them and later we will see the classes that correspond to each one.

- Audio I/O: this block performs all the operations concerning to audio reading and writing, we use it to read the audios for the training phase but also when reading them in order to execute our trained MLP.
- Format Conversion: this block is used for all kind of conversions of the audios to the Torch format for our MLP to read them and thus performing the training or the execution.

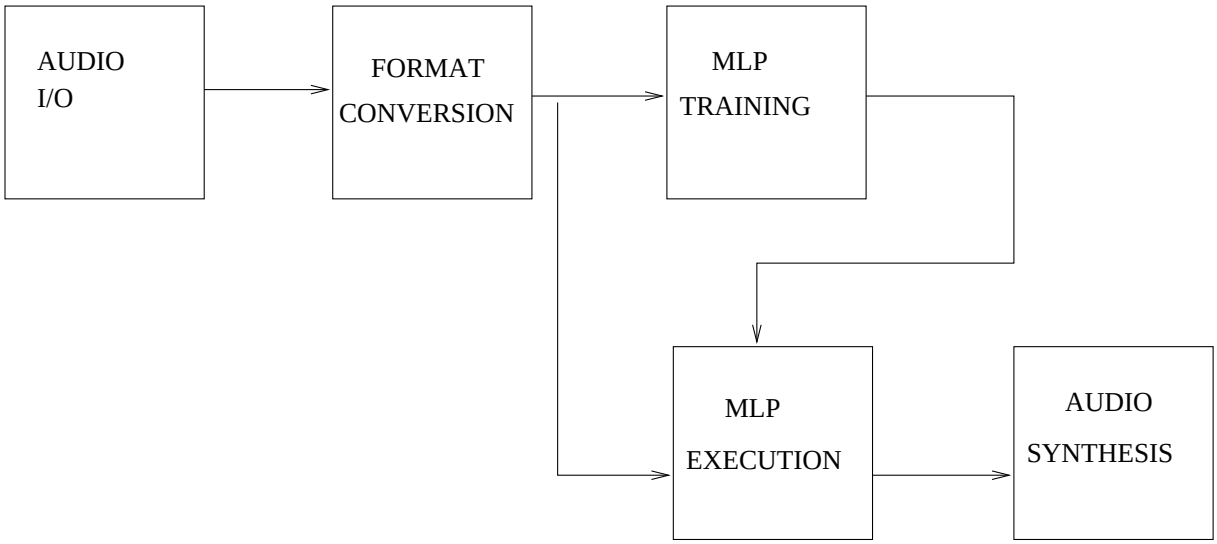

Figure 8.1: System Blocks Diagram

- MLP Training: this block allows us to train our MLP according to the parameters we introduce to it. It returns a training parameters file in which the trained MLP neurons weights are saved.
- MLP Execution: once our MLP is trained we can execute any kind of input on it, and this block is the one that allows us so. The block returns the MLP output according to the training parameters file that we have introduced to it.
- Audio Synthesis: this block performs and standard *Overlap and Add* process which recieves the frames that the trained network has returned. The audio is wrote using the Audio I/O block.

8.2 Class Diagram and Class Overview

First we will include the system static class diagram (see figure 8.2, it will help us to understand the structure of the project and the different blocks that form the overall system. Then, we will will explain in deepness the functionalities of each class, it must be said that we will not mention any kind of Getters and Setters due to its obvious functionality.

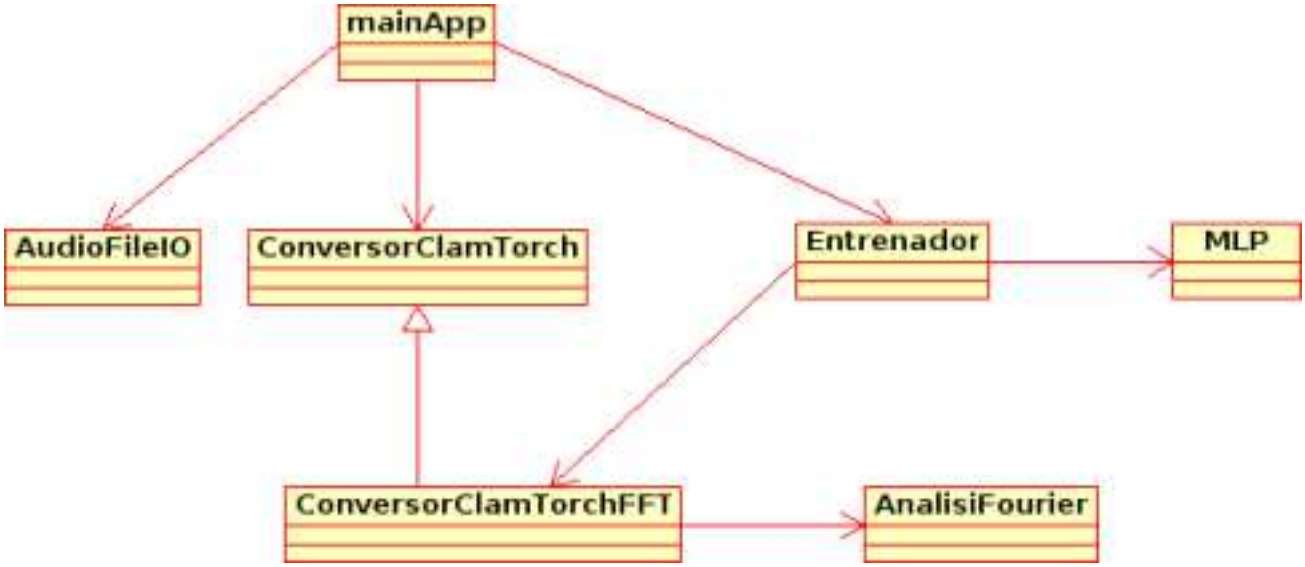

Figure 8.2: System Static Class Diagram

8.2.1 MainApp

This class includes the main method in which all the process are developed, as it can be seen on figure 8.2, this class uses the classes *AudioFileIO*, *Entrenador* and *AudioFileIO* (which will be explained next) and performs all the operations or blocks related to the system overview section.

8.2.2 AudioFileIO

This class performs all the operations needed for reading, manipulating and writing standard WAV files; all the input/output operations use methods which belong to the CLAM framework which offers everything needed to read and write any kind of audio files in WAV format; it has the next methods:

- LoadAudioFile(): loads on the member *audio* the audio file which name is in member *nomFitxer*.
- SaveAudioFile(): saves on file which name is in member *nomFitxerSortida* the audio data which is in member *audio*

Those are the main methods, apart from these, we have total access to the audio data we have saved on member *audio*.

8.2.3 MLP

This class implements and standard Multi-Layer Perceptron, almost all the class is developed using Torch methods which provide us with all the operations for manipulating an MLP. These class methods are:

- Constructor: where we define the number of inputs and outputs, the maximum number of iteracions, the end accuracy, the learning rate and also we specify if data should be normalized or not.
- CarregaDadesProva(char file): loads the test data, which is on file *file*.
- CarregaDades(char file): loads the training data, which is on file *file*.
- CarregaDadesValidacio(char file): loads the validation data, which is on file *file*
- Entrena(): perfoms the MLP training according to the data loaded on the class.
- ExecutaXarxa(): executes the MLP with the test data loaded on the class.
- SalvaParametresLlures(char file): saves the MLP weights (once trained) on the file *file*.

As we can see the class performs all the basic operations which belong to the MLP described on *Tools and Concepts* chapter.

8.2.4 Entrenador

This class is just an interface that allows the user to access to the MLP class without writing any line on Torch framework, it has just the same functionalities so just watching the code it would be very easy to derive the funcionality of each method.

8.2.5 ConversorClamTorch

This class performs all the conversions among the two frameworks used on this project, it is a very important one since it makes easier the programming task; the conversion has been done following everything that we have described on *Concepts on Framework Integration* chapter. This class has the following main methods:

- ClamArray2TorchSeq(CLAM::DataArray in, Torch::Sequence seq): takes the CLAM data of variable i and tranforms it to a Torch sequence putting it on *seq*.

- TorchSeq2ClamArray(Torch::Sequence seq, CLAM::DataArray in): the same as the last method but inverted.

- TorchMemDataSet2File(Torch::DataSet data, char fileIn, char fileOut): saves the data on dataset *data* on the files *fileIn* and *fielOut*, the first for the inputs and the second for the outputs.

- ClamArray2TorchFileFormat(...): takes the data from two CLAM arrays (*inputs* and *outputs*) and puts it in the training format for our MLP. To do this the function it takes as the parameters the number of examples (*Nexamples*), the number of inputs and outputs (*Ninputs*, *Noutputs*), the hop size (*step*) and the name of the resulting file (*trainFile*).

8.2.6 ConversorClamTorchFFT

This class is a child class of the last one an it adds the option of put on the Torch training file the FFT of the data; to perform this the class provides a method very similar to *ClamArray2TorchFileFormat* with the same functionality, taking in account that it saves on the file de Fast Fourier Transform of the data.
Our system final version does not use this function anymore since we have done our system on time domain, however this function opens the possibilities to the programmer for doing some kind of learning on the spectral domain.

8.2.7 AnalisiFourier

This class gives us all the necessary operations to get a complete audio Analysis/Synthesis process, on this sense we have implemented the methods on the easiest possible way in order to have a very simple usage class, let's watch them:

- ConfiguraFFT(int nSamples, int sampleRate): determines our FFT size and the sample rate.
- ConfiguraIFFT(int nSamples, int sampleRate): determines our IFFT size and the sample rate.
- Analitza(CLAM::Audio audio, CLAM::Spectrum spec): performs the Fast Fourier Transform algorithm putting its result on *spec*
- Inversa(CLAM::Spectrum spec, CLAM::Audio audio): performs the Inverse Fast Fourier Transform algorithm putting its result on *audio*.

There is some aspects to comment about this class in order to perform a succesful usage:

- The methods usage should be as follows: fisrt we should configure our FFT, the one should apply the FFT to his data; then in order to reconstruct the audio we should configure the IFFT algorithm and finally we should apply the IFFT algorithm in order to get our signal on time domain.
- We have preferred to limit the user in terms of configuration: the user can not choose the kind of window nor the amount of circular shift. We have chosen standard values for them; concretely, on one hand we chose a BlackmanHarris 92 Db analysis window and a Triangular as the synthesis one and on the other hand we have chosen a circular shift amount that is half of the window size.

8.3 Sequence Diagram

Since our project has very concrete steps we want to enforce them showing the system sequence diagram, which will show the different steps that compose our whole work, this diagram can be seen on figure 8.3.

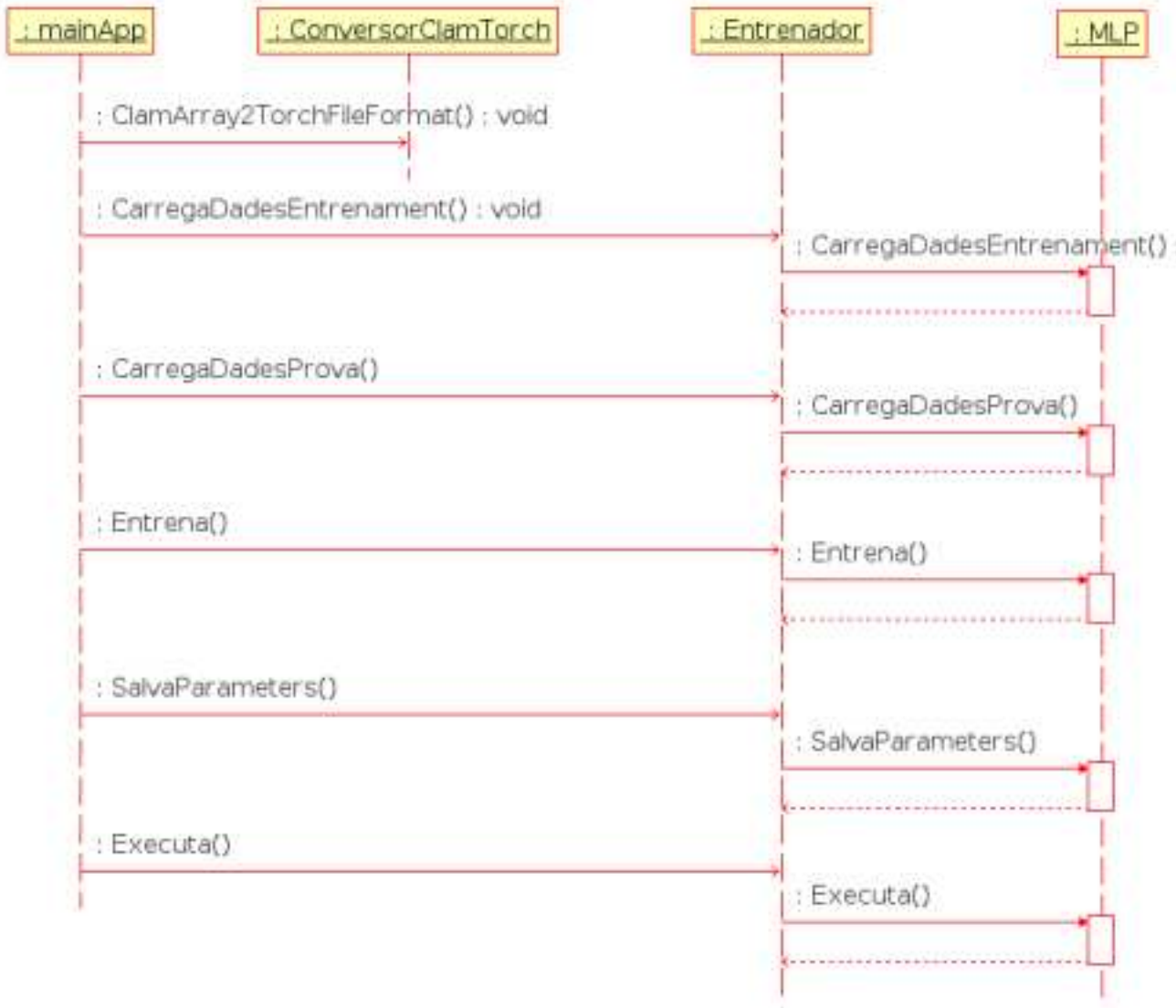

Figure 8.3: Sequence Diagram

Chapter 9

Conclusions and Future Work

To begin with, we would like to say that neural networks are not the most common nor the most orthodox way to model electric guitar effects; however we had some motivations to lead our work onto this field.
For instance, there is a very important personal motivation due to our high interest on neural networks and this project was an attemp to develop something interesting using them. Since music (apart from computers) is our main passion we thought it could be a good idea to put together both disciplines.
Another important motivation was that neural networks have not been extensively used for audio processing, therefore we thought it could be interesting to investigate one new machine learning application on audio processing; not to mention that since neural networks can learn any kind of non-linearity transformation and, taking in account that an effect pedal is just a non-linear transformation, it was very suggestive to try to emulate an overdrive pedal using some kind of neural network.
Before starting to explain our conclusions we want the reader to know that all of them are based on our personal experience while developing this project and, therefore, they are bounded by our parametrization: network type, training data, learning algorithm, etc.
We started our investigation by performing a high-pass filter training on the spectral domain, what's more, this training was focused only on the magnitude information because we supposed that phase was not going to be affected by this filter; it must be said that the results on this training were too bad because the network was not

able to learn the filter.

First of all we trained the network with both real an utopic sweep waves and the network behaviour was linear regarding to the harmonics, so it was a bad idea to consider our signal as a sum of several spectrum which contained just one harmonic.

After trying the sweep, we trained our network with white noise (both real and utopic) having on mind that we were going to train all the possible frequncies, but the result was that the network could only learn spectrum with flat magnitude, thus, when passing an scaled spectrum (a real one) the networks could not perform any kind of filtering.

After trying to perform our training on the spectral domain and due to its very bad results we decided to do it on the time domain according to the Wan and Nelson work (see reference [1]), which was about denoising speech signals using neural networks. The results on this experiment have been relatively good because the network has been able to learn the effect transformation correctly except for a little problem we will comment on later.

We want the reader to know that we have focused our research more on the audio processing aspects than on the neural network, so we have done a little bit of *test and trial* work in the sense that we have done many tests in order, for instance, to proof that the distortion came from the network and did not come from normalization and neither have developed an extensive training corpus in order to get a totally succesful training. Exactly, due to this lack of training corpus we have had a very serious affair with a noisy whistle generated from the network; however it must be said that we have found some very effective ways to get rid of this problem.

We would like to conclude our conclusions by saying that this project has been a first attempt on studying the viability of neural networks usage on this area; we think that the results are promising, not to mention that there is still lots of room for improvement. This project is not limited to neural networks without memory due to the existence of the recurrent neural networks on which possibly pedals such as *delay*, *chorus* and *flanger* could be modeled. Apart from this, it is important to mention that our system can work on real-time and on this field we are now developing a VST plug-in in which the user can play the guitar and get the corresponding pedal transformation from the computer.

Obviously this project does not finish here, there is still two improvements or ampliations that could be very suggestive in order to get a more complete system capable

to be close to the pedal emulation systems available on market nowadays:

- Pedal Parametrization: guitar effect pedals have usually some knobs to control the amount of some tranformation parameters, for example our transformation has three parameters (*overdrive*, *tone* and *volume*); we have modeled our pedal with one of the several combination of these knobs, but what we want for the future is the network to learn how these parameters affect the transformation.
- Other Pedals Modeling: as we have said before, in order to have a complete system, we should perform the learning of other famous guitar effects such as *wah*, *chorus*, *compression*, *flanger*, and so on...

Bibliography

[1] Eric A. Wan and Alex T. Nelson: *Networks for Speech Enhancement*; 2000.

[2] Xavier Amatriain: *An Object-Oriented Metamodel for Digital Signal Processing with a focus on Audio and Music* (PhD Dissertation); October 2004.

[3] Nir Lipovetzsky: *Modelo probabilstico en CLAM. Reconocimiento musical mediante Modelos Ocultos de Markov* (Master Thesis); December 2004.

[4] http://www.iua.upf.es/mtg/clam/

[5] http://www.torch.ch/

[6] http://cslu.ece.ogi.edu/nsel/

[7] http://www.shlrc.mq.edu.au/speech/perception/