Internship Report
Written in August 2022

Automatic effect recognition and configuration for timbre reproduction

Intern:
Alexandre D'Hooge
`dhooge@atiam.fr`

Supervisor:
Gaëtan Hadjeres
`gaetan.hadjeres@sony.com`

March 1st — August 31st

Key-words — audio effects, music information retrieval, differentiable signal processing, sound matching, computer music;

Mots-clés — effets audio, traitement de l'information musicale, traitement du signal différentiable, reproduction sonore, musique assistée par ordinateur.

Abstract

Many musicians use audio effects to shape their sound to the point that these effects become part of their sound identity. However, configuring audio effects often requires expert knowledge to find the correct setting to reach a desired sound. In this report, we present a novel method to automatically recognize the effect present in a reference sound and find parameters that allow to reproduce its timbre. This tool aims at helping artists during their creative process to quickly configure effects to reproduce a chosen sound, making it easier to explore similar sounds afterwards, similarly to what presets offer but in a much more flexible manner.
We implement a classification algorithm to recognize the audio effect used on a guitar sound and reach up to 95 % accuracy on the test set. This classifier is also compiled and can be used as a standalone plugin to analyze a sound and automatically instanciate the correct effect. We also propose a pipeline to generate a synthetic dataset of guitar sounds processed with randomly configured effects at speeds unreachable before. We use that dataset to train different neural networks to automatically retrieve effect's parameters. We demonstrate that a feature-based approach with typical Music Information Retrieval (MIR) features compete with a larger Convolutional Neural Network (CNN) trained on audio spectrograms while being faster to train and requiring far less parameters. Contrary to the existing literature, making the effects we use differentiable does not allow to improve the performance of our networks which already propose fair reproduction of unseen audio effects when trained only in a supervised manner on a loss on the parameters. We further complete our results with an online perceptual experiment that shows that the proposed approach yields sound matches that are much better than using random parameters, suggesting that this technique is indeed promising and that any audio effect could be reproduced by a correctly configured generic effect.

Résumé

Beaucoup de musiciennes et musiciens utilisent des effets audio pour façonner leur son, au point que ces effets peuvent devenir partie intégrante de leur identité sonore. Pourtant, configurer des effets nécessite souvent une maîtrise avancée afin de trouver les réglages permettant d'obtenir le son désiré. Dans ce rapport, nous présentons une nouvelle méthode pour reconnaître automatiquement l'effet utilisé dans un son de référence et trouver les paramètres permettant de reproduire son timbre. Cet outil a pour but d'assister les artistes dans leur processus créatif afin de rapidement configurer des effets pour reproduire le son désiré. Ceci facilitant l'exploration de sons similaires à la référence, comme peuvent le permettre les préréglages parfois disponibles, mais d'une façon bien plus flexible.
Nous implémentons un algorithme de classification permettant de reconnaître l'effet utilisé sur un son de guitare et atteignons jusqu'à 95% de prédictions correctes sur les données de test. Le classifieur est également compilé et peut être utilisé comme une application indépendante pour analyser un son et instancier automatiquement l'effet détecté. Nous proposons également une procédure pour générer un ensemble de sons de guitare traités par des effets configurés aléatoirement à des vitesses auparavant inatteignables. Nous utilisons par la suite ces nouvelles données pour entraîner un réseau de neurones à retrouver automatiquement les paramètres d'un effet. Nous démontrons qu'une approche basée sur l'extraction de descripteurs audio, comme il est courant en traitement de l'information musicale, renvoie des résultats comparables à ceux d'un réseau de neurones convolutionnel entraîné sur des spectrogrammes ; la première approche étant par ailleurs plus rapide et plus légère à implémenter et entraîner que la seconde. Contrairement à la littérature, rendre les effets utilisés différentiables ne permet pas d'améliorer les performances de nos algorithmes qui parviennent déjà à imiter plutôt fidèlement des effets jamais rencontrés auparavant. Nous complétons enfin nos résultats par un test perceptif en ligne qui confirme que l'approche évaluée est une large amélioration sur l'utilisation de paramètres aléatoires, suggérant ainsi que la technique proposée est prometteuse et qu'il est effectivement possible de reproduire n'importe quel effet par un effet générique correctement configuré.

Acknowledgements

I would like to thank my supervisor Gaëtan for supporting my project and for many fruitful discussions. I would also like to thank Alain, Bernardo, Cyran and Georges for all their relevant suggestions and helpful comments on my work. Many thanks go to Hugo who offered me to implement my work as a VST plugin. Thank you to Aura for thoroughly proofreading my work.

I am grateful to the whole music team for their interest in my work, it was a great source of motivation.

Finally, I would like to thank all members of Sony CSL for their kindness and making me feel welcome. It was a pleasure to come to work every day.

Contents

Abstract **I**

Acknowledgements **II**

Table of Contents **III**

Acronyms **V**

List of Figures **VI**

List of Tables **VII**

1 Introduction **1**
1.1 Sony CSL . . . 1
1.2 Audio Effects in Music . . . 1
1.3 Report outline . . . 4

2 Theoretical Background **5**
2.1 Classification . . . 5
2.1.1 k-Nearest Neighbors . . . 6
2.1.2 Support Vector Machines . . . 7
2.1.3 Linear Perceptron . . . 8
2.2 Neural Networks . . . 9
2.2.1 Training . . . 9
2.2.2 Multi-Layer Perceptron . . . 10
2.2.3 Convolutional neural networks . . . 10

3 State of the Art **12**
3.1 Audio effects emulation . . . 12
3.1.1 White-box modelling . . . 12
3.1.2 Black-box modelling . . . 13
3.1.3 Gray-box modelling . . . 13
3.2 Effect recognition and parameters estimation . . . 13
3.2.1 Effect recognition . . . 13
3.2.2 Parameters estimation . . . 14
3.3 Differentiable audio effects . . . 15

4 Audio effect Recognition **16**
4.1 Data . . . 16
4.1.1 Dataset . . . 16
4.1.2 Features . . . 17
4.1.3 Functionals . . . 18
4.2 Implemented Architectures . . . 19
4.3 Results . . . 21
4.3.1 Complete version . . . 21

4.3.2 Simplified Version 22
4.3.3 Energy consumption 22
4.3.4 Feature Importance 23
4.4 Compiled version 23

5 Estimation of effect parameters **25**
5.1 Proposed approach 25
5.2 Data 26
5.2.1 Added features 27
5.3 Implemented architectures 27
5.3.1 Simple regression network 27
5.3.2 Convolutional Neural Networks 28
5.3.3 Training strategies 28
5.3.4 Effect class conditioning 29
5.3.5 Classifier on synthetic dataset 30
5.4 Results 33
5.4.1 Simple MLP 33
5.4.2 Convolutional Neural Network 33
5.4.3 Perceptual experiment 34
5.4.4 Power and Size requirements 37

6 Conclusion **38**

Bibliography **39**

A Audio effects **42**

B Classification **44**

C Regression **47**

Acronyms

***k*-NN**	*k*-Nearest Neighbors
AM	Amplitude Modulation
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CSL	Computer Science Laboratories
DAW	Digital Audio Workstation
FD	Finite Differences
FFT	Fast-Fourier Transform
FiLM	Feature-wise Linear Modulation
Fx	effects
GPU	Graphics Processing Unit
LFO	Low-Frequency Oscillator
MFCC	Mel-Frequency Cepstral Coefficient
MFCCs	Mel-Frequency Cepstral Coefficients
MIR	Music Information Retrieval
MLP	Multi-Layer Perceptron
MRSTFT	Multi-Resolution STFT
MSE	Mean-Squared Error
OvA	One-Versus-All
OvO	One-Versus-One
ReLU	Rectified Linear Unit
RMS	Root-Mean Square
SPSA	Simultaneous Perturbation Stochastic Approximation
STFT	Short-Time Fourier Transform
SVM	Support Vector Machine

List of Figures

1.1 Physical and Digital effects . . . 2
1.2 Summary diagram of the internship objective. . . . 3

2.1 k-NN algorithm . . . 6
2.2 SVM illustration . . . 8
2.3 Kernel of a convolutional layer . . . 10
2.4 Stride in a convolutional layer . . . 11
2.5 Dilation in a convolutional layer . . . 11

4.1 Effect classes in the dataset . . . 16
4.2 Spectral moments extracted from a distorted guitar sound. . . . 18
4.3 Spectral roll-off example for a distorted guitar sound. . . . 19
4.4 Summary diagram of the Classifier network. . . . 20
4.5 JUCE plugin . . . 24

5.1 Proposed pipeline for Distortion emulation . . . 26
5.2 Implementations of the MLP regression network . . . 28
5.3 Principle of Residual networks . . . 28
5.4 Regression CNN summary diagram . . . 31
5.5 Confusion matrix of the complete classifier on the synthetic dataset . . . 31
5.6 Confusion matrix of the simplified classifier on the synthetic dataset . . . 32
5.7 Confusion matrix of the mixed classifier on the synthetic dataset . . . 33
5.8 Screenshot of the online experiment interface . . . 35
5.9 Boxplots of the online perceptual experiment . . . 36

B.1 Confusion matrices of the classification experiments on 11 effects. . . . 45
B.2 Confusion matrices of classification on aggregated classes . . . 46

C.1 Architecture of the AutoFx model . . . 49
C.2 Output of the last FiLM layers depending on conditioning. . . . 50

List of Tables

2.1 Possible outcomes of a binary classification problem. 5

4.1 Preliminary classification results 20
4.2 Aggregated effects class of the original dataset. 22
4.3 Training time and energy consumption of classification models 22

5.1 Size and power requirements of the regression models 37

A.1 Audio effects cheatsheet 43

B.1 Classification features 44

C.1 Regression features. 47
C.2 Regression networks results 48

Chapter 1

Introduction

1.1 Sony CSL

I conducted my internship at **Sony Computer Science Laboratories (CSL) Paris**. These laboratories belong to **Sony France** which is itself a part of **Sony Europe Limited**. Contrary to most laboratories in the industry, Sony CSL Paris is a research laboratory with no incentive to design products that could be sold afterwards. This allows the researchers to focus on topics interesting to the community without constraints on releasing commercial software.
I was an intern in the **Music Team** under the supervision of Dr. Gaëtan Hadjeres for a project tailored to my personal interests. I proposed to design a tool to help artists automatically configure their sound when practising or composing a new song, inspired by the difficulties I personally encounter as a guitarist. This idea was completely in line with what the Music team now aims to achieve which is to help the artists composing more easily by offering them new tools to develop their creativity. Examples of past and ongoing releases and collaborations are available on the official website of the team: https://cslmusicteam.sony.fr/.

During my internship I was fairly autonomous and free to try the approaches that felt interesting and promising. I made a few presentations to the team to share my progress and got the opportunity to exchange with artists on the tool I was designing. I was also able to collaborate with other members of the team to combine my work with theirs or so that they could help me design an interface for my project. This internship was a great experience that allowed me to thrive and confirm my wish to pursue my career in research.

1.2 Audio Effects in Music

The timbre of a sound can be defined as anything that is not its *pitch* or its *loudness*. Reproducing the timbre of another sound is a situation that can arise frequently when manipulating audio. One may want to change the voice of a speech recording to resemble the one of a famous actor or a sound engineer could need to modify a recording done with older tools to match the characteristics of new takes. Those tasks ultimately come down to *timbre reproduction* and they can sometimes be done using audio effects.

In this report, I call *audio effect* any process that is used to alter a sound in a creative way. Those effects (Fx) can present themselves as hardware processing units[1] such as guitar effect pedals. They can also be digital and available as plug-ins that can be *standalone* – no other software is required to use the effect – or included in a Digital Audio Workstation (DAW)[2] (see Figure 1.1).

[1]It is relevant to note that hardware does not imply analog processing and that even effects in physical hardware can be digital.

[2]A DAW is a software used by musicians/sound engineers to record or edit songs and that usually allows to add effects on tracks, control effect parameters through time or manipulate MIDI clips. See for example Ableton, FL Studio (Proprietary) or Ardour (Open-source).

Figure 1.1: A voice processing effect pedal (left) and a digital Chorus plugin (right).

Inspired by the existing literature, I classify audio Fx into three broad categories:

- **Non-linear:** an effect that includes a non-linearity to add harmonics to the input sound. The general principle of such effects is usually to multiply the input signal by a *gain* (often called *Drive*) before applying a saturating function on the signal's amplitude to forbid it exceeding a given threshold. This *saturation* step enriches the spectral content of the signal by adding high frequency harmonics. It can be done digitally with hard-clipping functions, soft-clipping functions (hyperbolic tangent for instance) or using hardware equipment such as a vacuum tube or diodes. Examples of such effects are *Overdrive, Distortion, Fuzz*, available under many different forms and with huge variety. Another non-linear effect well-known to sound engineers is *Compression*, which is usually not used as a way to alter timbre – even though it can be done [1];

- **Modulation:** Fx that use signal modulation to enhance the input signal. It can be Amplitude Modulation (AM) for effects such as *Tremolo* or Frequency Modulation (FM) for *Chorus, Flanger or Vibrato*. Though not based on Frequency Modulation, the *Phaser* effect can also be included in this category. It is perceptually close to the *flanger* effect but it is based on iterative filtering of the input signal by all-pass filters;

- **Ambient:** Fx that recreate the feeling of being in a given physical space. This includes in particular *Reverb* which recreates the reverberation of a room. The effect can be implemented digitally from the impulse response of a room or using theoretical formulas. It can even be an analog emulation like *Spring reverb* where an electric signal (the sound recorded by a microphone) is converted by magnetic transducers into mechanical vibrations of a spring which will increase the decay of the signal and add many reflections.
  Another typical ambient effect is *Delay* or *Echo* which duplicates the input signal and add it back to the original after waiting for a chosen duration and usually after reducing its volume.
  I also include in ambient Fx *Equalization* which allows altering the repartition of spectral content in a sound and can be used to completely change the perception of a sound.

A short description of several audio effects can be found Table A.1, along with their use cases and the way they are usually implemented.

Audio Fx are used almost everywhere in current music. Modern instruments allow for simple use of Fx because they record electrical signals, electric guitar players for instance often use effect pedals to completely transform their sound. This gives access to a huge variety of sounds, from the mellow sounds of psychedelic rock to the harsh distorted sounds of metal music. Effect pedals are so important that guitarists can spend a lot of time turning them on and off on stage, to the point that a genre called *Shoegaze* has been invented.

However, guitarists are not the only musicians to use Fx. Vocalists can use them too for improving their performance. Most produced vocals now use *reverb* at some point to blend the vocals with the rest of the music. Some vocal effects can also be used to create a signature sound, one of the most recognizable effect being Antares' *Autotune*.
Keyboardists can also be heavy Fx users since many synthesizer sounds are obtained through several layers of Fx to completely reshape the original sound produced.
Besides, audio Fx are well known to sound engineers that use them when mixing tracks for reshaping sounds or balancing them. Fx are also present in live music to make sure that all instruments are correctly heard in the concert venue, for instance using compression to restrain the attack of loud drum sounds that might otherwise drown the other instruments' sounds.

The ubiquity of Fx means that most beginner musicians and producers must learn how to use them. Having a home-studio setup to compose and records tracks is now fairly accessible and a plethora of quality audio Fx is available freely on the internet. However, understanding how to choose the parameters of those effects is a lifelong learning process that may require complex theoretical knowledge [2]. Such difficulties can hinder the creative process of artists who may not want to spend a lot of their time tweaking effects instead of composing/playing/recording.

For this reason, a tool that automatically configures Fx according to a reference sound could help artists focus on creativity and explore whole new sounds way more easily. This is the goal I set myself for this internship, it is summarized by the diagram presented Figure 1.2: the artist chooses a reference sound that has a timbre they like due to some processing effects; the designed tool extracts the effect(s) used and find optimal parameters for a generic Fx rack to reproduce the perceptual characteristics of the reference sound; the artist then connect their instrument to their computer through a sound interface and forwards the dry/unprocessed[3] signal through the newly configured effect or effects rack, reproducing the reference sound in no time.

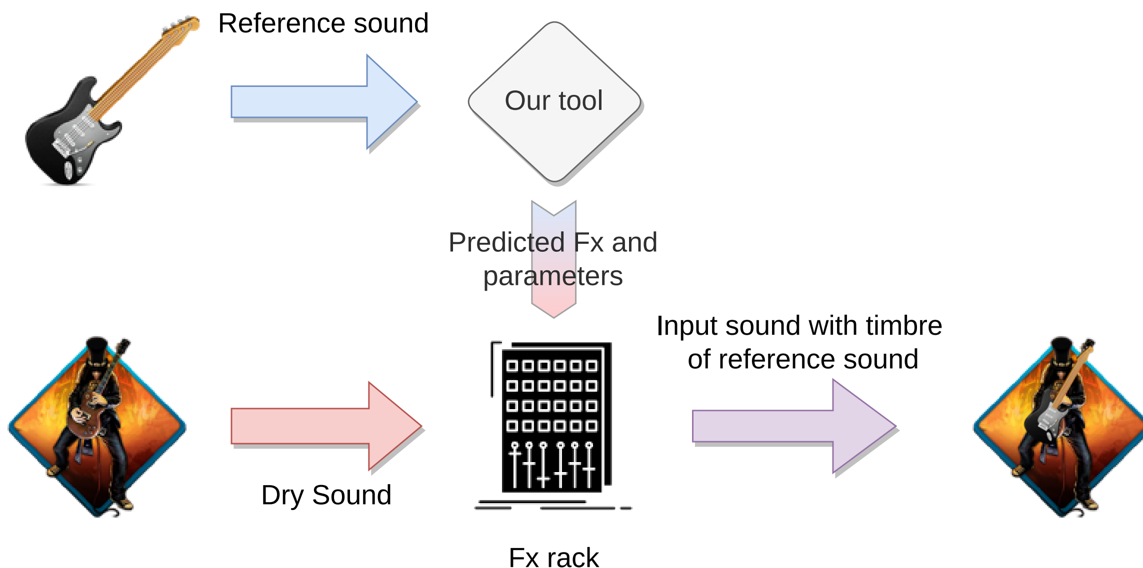


Figure 1.2: Summary diagram of the internship objective.

[3]In musical jargon, a "dry" sound is synonymous to a clean sound, i.e. a sound that have not been processed in any way. The counterpart being a "wet" sound, hence many effects having a "Dry/Wet" knob.

1.3 Report outline

The solution we implement is two-fold. The first part is a *classification algorithm* to recognize the effect used in a audio signal while the second is a *regression algorithm* to find parameters that allow to reproduce the reference sound timbre. This separation is due to the final objective that is to obtain a *standalone plugin* that could be used by artists in their creative process and is inspired by what is usually done by human experts to do a similar task. The first algorithm recognizes the effect used and informs the user who then has the choice to stop the processing here and implement their own effects or let the second algorithm configure effects to reproduce the reference timbre.

In this report, we thus begin with a theoretical background on classification algorithms and neural networks that can be used for classification or regression.
We then present the state of the art on the tasks of effects reproduction, recognition of audio effects and retrieval of effects parameters. We will in particular adress how audio effects can be associated to neural networks.
The next chapter focuses on the algorithms we implemented for the classification of audio effects and their recognition in audio samples. This work is conducted on guitar sounds because it is the instrument that originally inspired the goal of that internship and because of data availability. We discuss the results obtained and explain the simplifying assumptions that are made to combine the classification algorithm with the regression network.
After recognizing the effect present, our goal is to retrieve its parameters. The corresponding chapter describes the implemented algorithms for this task and present the difficulties encountered. Quantitative results are transcribed and the answers to an online perceptual experiment are reported.
Finally, we recall what failed and succeeded in this internship, what results are of interest to the research community and what could be done in the future to improve this work.

The code of this internship is publicly available at:

https://github.com/adhooge/AutoFX

Sound examples are available on the accompanying website:

https://adhooge.github.io/AutoFX/

Chapter 2

Theoretical Background

This work implements neural networks and other machine learning algorithms for classification or regression. The following section is a short introduction to the theoretical notions behind the implemented architectures and will hopefully give the reader enough understanding of the field's basis.

2.1 Classification

Classification problems can arise in many different fields and contexts. For instance in biology, it can be interesting to predict if a person is subject to a specific illness based on a set of measures like weight, height, age... For sorting purposes, one might also want to design an algorithm that automatically recognizes if the house pet in a picture is a dog, a cat, a bird or a rat.

Those two examples already illustrate the variety that can exist in classification tasks. In the first example, only two outcomes are possible: whether the person is sick or not, this is called **Binary Classification**; in the second example, four possible classes exist and even more could be considered, the problem being thus called **Multiclass Classification**. Besides, it is likely that classes will not be represented equally in a random panel. For house pet recognition, more people own dogs or cats than birds or rats and data is thus likely to be unbalanced. This is not necessarily an issue but it should be known before implementing a classification algorithm, for instance to keep that ratio when splitting the data into smaller sets.
More importantly, when aiming at detecting cancer for example, an algorithm that always predict an absence of cancer would be right around 95% of the time since in a completely random panel of persons approximately 5% have cancer [1]. Taken out of context, a 95% accuracy is an impressing score. However, in such a task, **False Negatives** (a person not diagnosed with cancer while they have one) are really dangerous and should happen as rarely as possible. For this reason, several metrics exist to assess the performance of classification algorithms and are presented hereafter.

Metrics

We begin with only considering a problem of Binary Classification. In that case, the possible outcomes are recalled Table 2.1.

		Predicted class	
		1	0
Actual class	1	True Positive (TP)	False Negative (FN)
	0	False Positive (FP)	True Negative (TN)

Table 2.1: Possible outcomes of a binary classification problem.

[1] 3.8 million French people live with a diagnosed cancer https://www.fondation-arc.org/cancer/le-cancer-en-chiffres-france-et-monde

Using these notations, we can define:

- the **Precision**:

$$\mathcal{P} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2.1}$$

- the **Recall**:

$$\mathcal{R} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2.2}$$

- the **F-score**:

$$F = 2 \times \frac{\mathcal{P} \times \mathcal{R}}{\mathcal{P} + \mathcal{R}} \tag{2.3}$$

All those metrics are between 0 and 1 with 1 being the best score. Precision assesses the capacity of the classifier to avoid False-Positives while Recall measures if the classifier misses any actual positive sample. The F-score is the harmonic mean of Precision and Recall, it is used when both metrics are important and need to be combined in a single value.
Using those metrics allows to mitigate results of a classifier and assess what is relevant depending on the task at hand. With the previous cancer-detection example, Recall is the most important score since we want people with cancer to be correctly diagnosed. Ideally, Precision should also be high to avoid false alarms but in that case a confirming test could be done to compensate for a low precision.
In a *Multiclass classification* problem, those metrics can be used on each class independently or can be averaged across classes if a value for the complete classifier is necessary.
In the following sections, now that we have seen how to evaluate the performance of a classifier, we present common classification algorithms and their theoretical principles.

2.1.1 k-Nearest Neighbors

The k-Nearest Neighbors (k-NN) algorithm is a simple technique for classifying samples of data. Based on training data consisting of pairs (data, class) that are stored by the classifier, each new sample is assigned a class based on its k nearest neighbors.
To identify those *nearest* neighbors, a distance d needs to be defined, it can be a simple absolute distance or more complex functions depending on the data. The neighbors can then be weighted to classify the new sample, either **uniformly** *i.e.* all neighbors weigh the same or by distance where each neighbor weighs $1/d$.
When the data is sparse, an alternative can be to define a radius instead of a number k of neighbors to find. When using a radius, all neighbors in this radius are used to classify the new sample.The main steps of the k-NN algorithm are summarized Figure 2.1. It should be noted that this algorithm requires high computational resources and that its performance is highly dependent on the chosen distance, making it a good fit to specific problems where the data distribution is at least partially known beforehand.

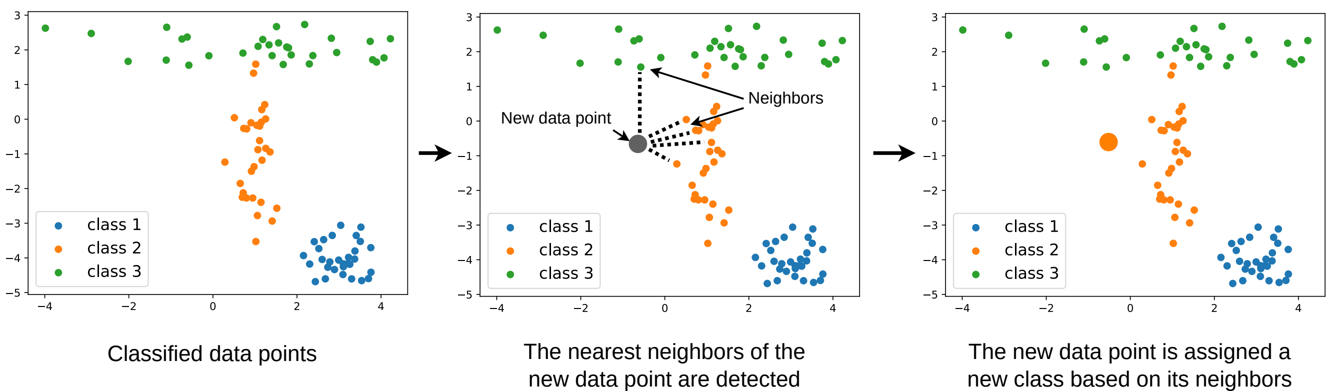


Figure 2.1: Steps of the k-NN algorithm to classify new data samples.

2.1.2 Support Vector Machines

A Support Vector Machine (SVM) is another classification algorithm that aims at finding the best boundary between two sets of samples, best referring in that case to the boundary that maximizes the distance between the classes' frontiers defined by the so-called *support vectors*. Let us consider a binary classification example for simpler analysis.
We consider a dataset of n training samples $(\vec{x}_1, c_1), (\vec{x}_2, c_2), \dots, (\vec{x}_n, c_n)$ where the $\vec{x}_i$ are vectors of coordinates and c_i is the assigned class of each data point, either $+1$ or -1. The algorithm objective is to find the *maximum-margin hyperplane* to split the data points according to their class. A hyperplane is entirely defined by its normal vector $\vec{w}$ as the set of points that satisfies:

$$\langle \vec{w}, \vec{x} \rangle - b = 0 \tag{2.4}$$

where $\langle ., . \rangle$ denotes the scalar product and $\frac{b}{||\vec{w}||}$ is the offset of the hyperplane to the origin with $||\vec{w}|| = \sqrt{\langle \vec{w}, \vec{w} \rangle}$. The algorithm then must define two hyperplanes by the following equation:

$$\langle \vec{w}, \vec{x} \rangle - b = \pm 1 \tag{2.5}$$

Any data point on or above the $+1$ hyperplane belongs to the class $+1$ and any data point on or below the -1 hyperplane belongs to the class -1, which can be written as:

$$\langle \vec{w}, \vec{x}_i \rangle - b \geq +1 \qquad \text{if } c_i = +1 \tag{2.6}$$
$$\langle \vec{w}, \vec{x}_i \rangle - b \leq -1 \qquad \text{if } c_i = -1 \tag{2.7}$$

Let $\vec{x}_+$ and $\vec{x}_-$ verify $\langle \vec{w}, \vec{x}_\pm \rangle - b = \pm 1$, the margin between the two hyperplanes can be obtained as:

$$\text{margin} = \left\langle \vec{x}_+ - \vec{x}_-, \frac{\vec{w}}{||\vec{w}||} \right\rangle \tag{2.8}$$
$$= \frac{\langle \vec{x}_+, \vec{w} \rangle - \langle \vec{x}_-, \vec{w} \rangle}{||\vec{w}||} \tag{2.9}$$
$$= \frac{1 + b - (-1 + b)}{||\vec{w}||} \tag{2.10}$$
$$= \frac{2}{||\vec{w}||} \tag{2.11}$$

To maximize the margin, it is thus necessary to minimize $||\vec{w}||$ which is the optimization objective used to fit a SVM. It is important to maximize this margin because, in the case of linearly separable data, there exists an infinity of hyperplanes that can correctly separate the data in its two classes but new data points are more likely to be misclassified if the decision boundary is closer to one class than the other. All notations and important notions are summarized Figure 2.2.

If the data to classify is not linearly separable or when there is some overlap between classes, improvements such as the *kernel trick* can be used to maintain the SVM efficiency. I advise the interested readers to look into more complete resources[2] for explanations of those techniques.

From binary to multiclass classification

A SVM is designed for binary classification problems. However, it can be generalized to multiclass classification problems using One-Versus-One (OvO) or One-Versus-All (OvA) problem splitting which choice depends on the problem at hand.

[2] For instance: https://ocw.mit.edu/courses/6-034-artificial-intelligence-fall-2010/resources/lecture-16-learning-support-vector-machines/

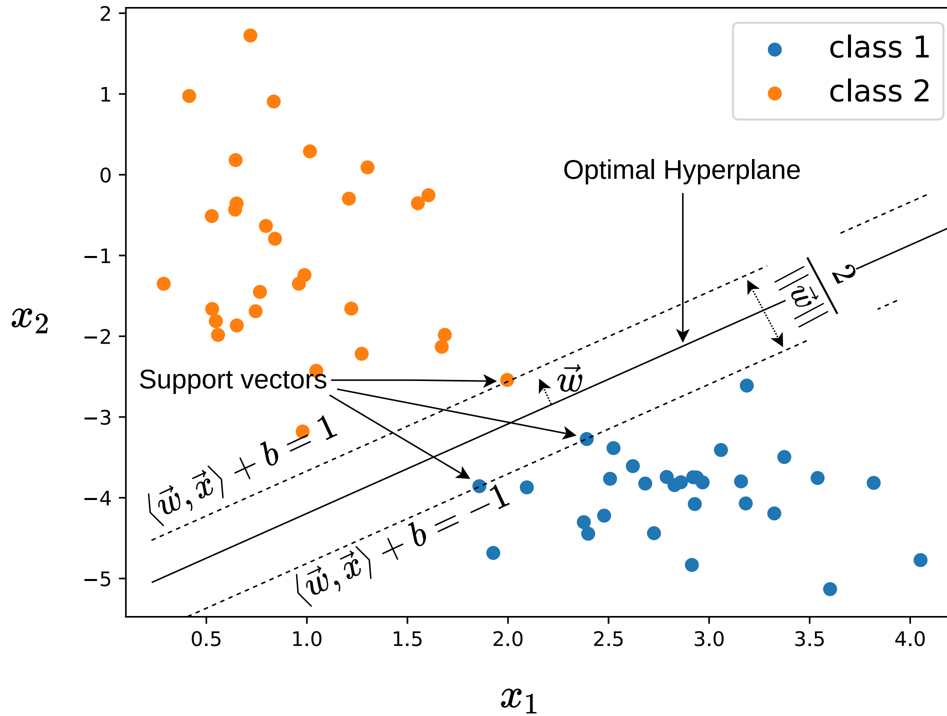


Figure 2.2: Data points separated by an optimal hyperplane (plain line) that maximizes the width $\frac{2}{||\vec{w}||}$ between the hyperplanes going through the *support vectors* (dashed lines). The coordinates vector of each datapoint is $\vec{x}_i = \left(x_1^{(i)}, x_2^{(i)}\right)$

.

In the case of a OvO approach, one binary classifiers is trained for each possible pair of classes and the final class is the one that received the most votes. If there are N classes, this technique requires $N(N-1)/2$ classifiers making it costly when the number of classes increases.
In a OvA approach, N binary classifiers are trained to answer the question *Does this sample belong to class c_i or does it actually belong to any other class?* with $1 \leq i \leq N$. The data sample is then directly assigned to the class with the maximum probability. This technique requires less classifiers than OvO but might not be efficient in all situations because the comparison between one class and all the others can be "unfair".

2.1.3 Linear Perceptron

A Linear perceptron is another algorithm that can be used for binary classification tasks. It aims at learning a function

$$f(\mathbf{x}) = \begin{cases} +1 & \text{if } \mathbf{w}.\mathbf{x} + b > 0 \\ -1 & \text{else} \end{cases} \tag{2.12}$$

where $\mathbf{x}$ is the input vector, $\mathbf{w}$ is a weight vector that is learned during training and b is a bias coefficient. $\mathbf{w}$ and b are updated to minimize the classification error of the M input samples:

$$\varepsilon = \sum_{i=1}^{M} ||f(\mathbf{x}_i) - c_i||_d \tag{2.13}$$

where c_i is the actual class of the sample and $||.||_d$ is the chosen distance metric.
This algorithm can be generalized to multiclass classification in the case where the output is a vector and the assigned class is the *argmax* of that vector. In that case, b becomes a vector and $\mathbf{w}$ a matrix.

2.2 Neural Networks

A linear perceptron is a very simple type of neural network. Neural networks need to be *trained* on a task to *learn* how to complete it. In this section, we explain how neural networks are trained before describing more complex neural network architectures.

2.2.1 Training

Loss functions

A neural network of any architecture needs to be trained to progress towards the desired minimum. For this reason, the first step is to define a **loss function**. This function will quantify how wrong or how right the network is by comparing its predictions to the ground truth. For a classifier network for instance, we want to compare the predicted class of an input sample to its actual class. We could use Accuracy, Precision or Recall as a loss function but it is expected of a loss function to drop towards zero as the network gets better. For this reason, the loss usually used for classification problems is the **Cross-entropy loss** defined as:

$$\mathcal{L}_{\text{cross-entropy}} = -\sum_{i=1}^{k} t_i \ln y_i \tag{2.14}$$

where $\mathbf{t} = (t_1, \ldots, t_k)$ is the target **one-hot vector** where all coordinates are 0 except the correct class that is a 1 and $\mathbf{y} = (y_1, \ldots, y_k)$ is the predicted output by the neural network where y_i is the probability of the input sample to belong to class i.
In other reconstruction or regression problem, a simple **Mean-Square Error Loss** (MSE loss) can be used to compare how close the reconstruction $\hat{\mathbf{x}}$ is to the original input $\mathbf{x}$ according to the formula:

$$\mathcal{L}_{\text{MSE}} = \sum_{i=1}^{n} |x_i - \hat{x}_i|^2 \tag{2.15}$$

Backpropagation

The loss function is the first step for training a neural network by **backpropagation**. This technique consists in updating each weight $w_{k,\ell}$ accordingly to the partial derivative of the loss function with respect to the weight:

$$w_{k,\ell}^{(i+1)} = w_{k,\ell}^{(i)} - \eta \frac{\partial \mathcal{L}}{\partial w_{k,\ell}} \tag{2.16}$$

the quantity η is a tuning parameter called the **Learning rate** and will scale how much the network is updated after each training step, the derivative being usually obtained *via* the **chain rule**. This update is similarly applied to any coefficient that needs to be optimized. This shows the main difficulty when it comes to neural networks which is that they should implement differentiable processing for backpropagating gradients. If any operation is not differentiable, it will stop the backpropagation of the gradients because the chain rule cannot be carried on until the end.

Activation functions

The output values of a layer are only constrained by the weights of that layer but sometimes it is necessary to return a value in a given range. For instance, when a network is expected to predict the probability of a variable, its output must be between 0 and 1. It could be attained by adding optimisation constraints during the training procedure but the most efficient way is to pass the output through a function that has an image set restricted to $[0, 1]$ like a **Logistic function** (sometimes called **Sigmoid function**):

$$\text{logistic}(x) = \frac{1}{1 + e^{-x}} \tag{2.17}$$

Many different activation functions exist[3] and they can also be added between layers to modify the coefficients values before sending them to the next layer. One common activation function between layers is the Rectified Linear Unit (ReLU) or small variations of it:

$$\mathrm{ReLU}(x) = \max(0, x) \tag{2.18}$$

This activation function has the specificity to stop the backpropagation of gradients whenever a value is negative and it is plebiscited because it is fast to compute and has no upper-bound.

2.2.2 Multi-Layer Perceptron

A more powerful version of the Linear Perceptron is the Multi-Layer Perceptron (MLP) that contains at least one *hidden layer i.e.* a layer that acts neither on the input or the output but on an internal variable that is forwarded to the next layer. That algorithm can learn complex non-linear functions that can be used for non-linearly separable classification or regression problems. If a MLP has h hidden layers, it has h weight vectors $\mathbf{w}_i$ and bias coefficients b_i where $1 \leq i \leq h$ so that the output of a single layer is:

$$\ell_i(\mathbf{x}) = g_i(\mathbf{w_i}.x + b_i) \tag{2.19}$$

where g_i is an **activation function** used to map the output of the layer i to another space.
The output of the entire network is thus:

$$\hat{y}(\mathbf{x}) = f(\mathbf{w_h}.g_{h-1}(\mathbf{w_{h-1}}.g_{h-2}(\cdots g_1(\mathbf{w_1}\mathbf{x} + b_1)\cdots) + b_{h-2}) + b_{h-1}) + b_h) \tag{2.20}$$

where $\hat{y}(\mathbf{x})$ is the network estimation of the desired output $y(\mathbf{x})$ associated to the input $\mathbf{x}$. f is the final **activation function** of the network.
A neural network is optimized to reduce the prediction error between $\hat{y}$ and y quantified by a **Loss function**. To do so, the weight are all updated by **backpropagation**.

2.2.3 Convolutional neural networks

The simple fully-connected linear networks (another name given to MLP) perform well on a number of tasks but usually fail to adress more complex problems such as audio or image recognition. For tasks requiring somewhat higher-level analysis, a Convolutional Neural Network (CNN) can be used. The main building block of a CNN is a **Convolutional layer** which consists in applying a **filter** on an area of the input data. The size of that area is defined by the so-called **kernel-size** and will be map to a single value of the output data, as illustrated Figure 2.3. The kernel can be a simple average filter but it can also be more complex with for instance learnable weights in neural networks.

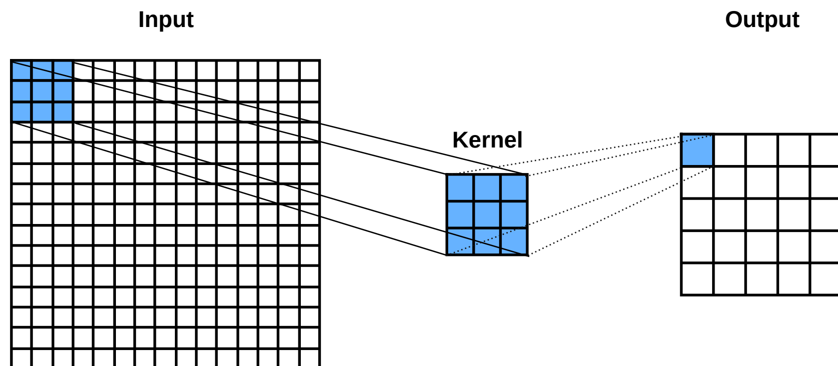


Figure 2.3: Principle of a convolutional layer in the case of a 3×3 kernel.

[3] See for instance https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity

Several control parameters exist to modify the output produced. For instance, the stride determines how the filter will move from one area to the next, as shown Figure 2.4. Changing the stride controls the overlap between each iteration of the filter and can even be set to skip entirely some values of the input (if the stride is higher than the kernel size). Since the stride impacts the size of the output, it is often used as a way to downsample the input data. It should also be noted that the input can be padded to make sure that beginning and end values are not treated differently than middle values.

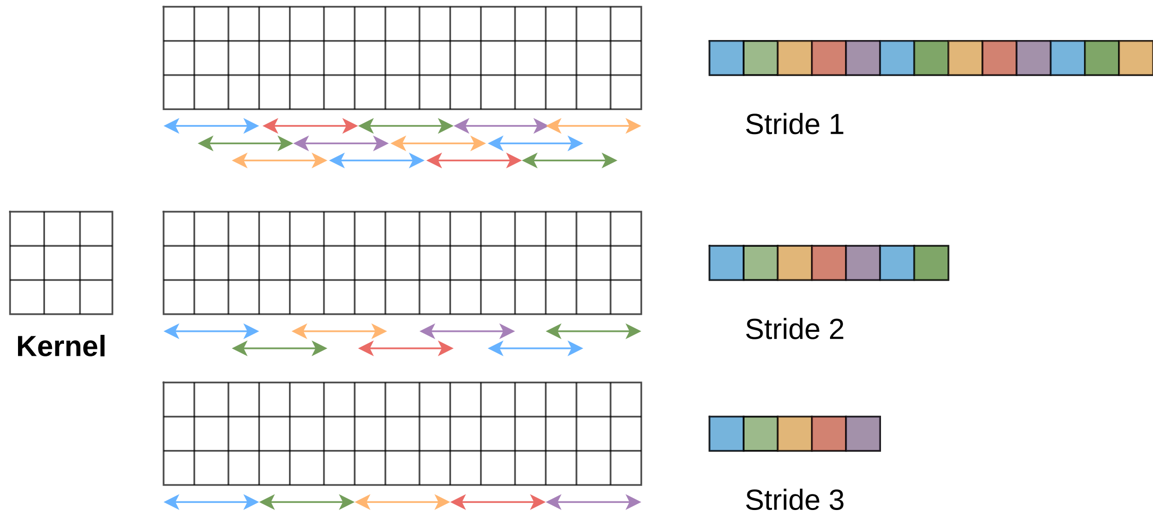


Figure 2.4: Outputs produced by convolutional layers with different strides. A colored arrow represents the area on which the kernel is applied to obtain the corresponding output cell.

Finally, another parameter called dilation can be used to modify how the convolution kernel is applied, see Figure 2.5. This parameter allows to increase the **receptive field** of a CNN which is the input area represented by a single output cell.

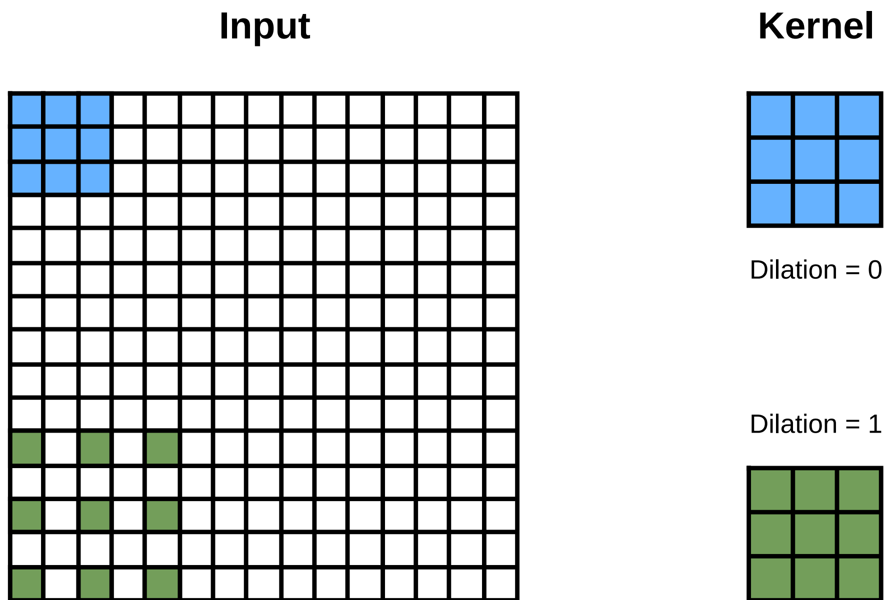


Figure 2.5: The dilation factor modifies how the kernel is applied to the input by skipping some cells.

In a CNN, the convolutional layers filters have fixed parameters (kernel size, dilation, padding, stride) but the weights are learnable. Besides, to increase dimensionality, it is common to add *channels* to the output by multiplying the number of filters and thus the number of learnable parameters. In that case, the input is processed as many times as there are channels but with other filter to obtain different *feature maps* that might capture different aspects of the input data.

Chapter 3

State of the Art

In this section, we will begin by discussing how audio effects are usually emulated in the literature. We will then present how audio effects can be automatically recognized from audio before tackling the issue of retrieving parameters of those effects. Finally we will present how audio effects can be made differentiable to include them in neural networks.

3.1 Audio effects emulation

Reproducing audio effects in a digital framework is an important research field that works on filling the gap in the industry between analog implementations and digital equivalents. This is in fact related to the never-ending conflict that opposes artists in favor of analog audio processing and artists using Analog-to-Digital Converters (ADC) to use digital hardware and software. Without taking any side, it should be noted that obtaining digital emulation of audio effects – but also synthesizers – can be useful for keeping a trace of legendary circuits that are no longer produced.

When it comes to digitally reproducing an audio effect, all approaches can usually be splitted into three categories that are: **White-box, Black-box and Gray-box modelling**. Each one of those can then be differentiable or not depending on the implementation.

3.1.1 White-box modelling

This technique is the oldest and consists in reproducing the functioning of an effect from theoretical knowledge. When the physical equations or the transfer function of a processing unit are known, they can be reimplemented in a digital framework and should thus guarantee the reproduction of the audio effect. This is of course limited to some extent because, especially when focusing on circuits with non-linear elements such as diodes or vacuum tubes, solving huge and complex equation systems can be untractable. Researchers thus have to make simplifying assumptions to reduce the complexity of their model, which also reduces its fidelity to the reference effect.

However, when a good theoretical model can be obtained, this approach yields good results as it can be observed in [3] where the researchers successfully reproduce the response of the *Red Llama* overdrive effect based on the analysis of its electronics.
Even though it is not a modulation of an existing effect, a similar approach from [4] shows that the non-linearity existing in a *bistable system* – a system with two equilibrium points – can be used as a distortion effect when implementing its physical equations.
Using a modal approach, [5] reproduces the behavior of a spring reverb. The author however notes that, even though the effect itself can be used in real-time, retrieving its parameters is computationally costly and must be done beforehand. This is not necessarily a problem but this observation supports the idea that working with complex physical equations can lead to important computational load.

Akin to white-box modelling, the theoretical knowledge of physical systems and phenomena can be used to propose new approaches to obtain a desired effect. [6] proposes a generalization of the Moog ladder filter with *state-variable filters* that allows for more control and can also model other circuits such as the Octave CAT filter. Similarly, nonlinear biquad filters can be used for analog modelling, as it is studied in [7].

3.1.2 Black-box modelling

Black-box modelling, as its name suggests, is a solution where the effect emulator is a black box *i.e.* its internal functioning is not known. The only constraint for a black-box model is that processing a sound with the original system or its black-box emulation should give the same output. Such approaches are more common since the advance of machine learning because neural networks allow reproducing complex systems that would be difficult to model with white-box modelling. It should however be noted that neural networks are usually used at the expense of explainability.
Black-box models can be used to emulate *tube amplifiers*, as it is proposed in [8] where the authors focus on adding control to the black-box model for improved usability. In [9], the authors give a thorough analysis of the existing literature of deep learning methods for modelling audio effects and propose a new architecture for the emulation of non-linear effects and the *Leslie Speaker Cabinet*, a modulation effect obtained by the rotation of a speaker inside a cabinet.

It is interesting to observe that neural networks can be used to add audio effects to sounds but also the opposite: removing them. For instance, [10] presents a neural network that can remove distortion effects on musical signals. Such possibilities are particularly interesting for timbre transfer tasks or sound morphing where the goal is to match the audio characteristics of a reference sound with an input sound. Considering the situation where the input sound is processed with an effect and the reference sound is already processed with another effect, it might be easier to use an intermediate clean sound instead of trying to morph directly the input sound to the reference style.

3.1.3 Gray-box modelling

Mixing both previous approach, gray-box modelling[1] consists in combining knowledge of the actual system, often obtained through measurements, with modelling techniques not necessarily representing the actual physical principle of the system. For instance, [11] models distortion circuits using a feedforward neural network with a predefined receptive field N representing the number of previous samples that are used to predict the next sample n. That receptive field is determined based on linear impulse responses of the effects, obtained with a swept-sine technique.

Gray-box modelling can also be done without any neural network: [12] uses a Wiener-Hammerstein model to emulates guitar amplifiers, its parameters being tuned during an optimisation phase to match the real system's responses to sine sweeps. In [13], the authors use a Chebyshev non-linear model fitted with a synchronized swept sine method and show that it can efficiently reproduce the processing of an overdrive pedal.

3.2 Effect recognition and parameters estimation

3.2.1 Effect recognition

When the effect is known beforehand, it is possible to choose the best approach to emulate it. However, if the audio effect is unknown, it must be identified before carrying on with its emulation. The specific case of recognizing guitar audio effects have been studied in two different papers [14, 15] by Stein *et al.* For this task, a new dataset of recordings of guitar and bass single notes processed with 11 different effects has been assembled. The authors extract carefully designed audio features that are then used to classify the samples with a Support Vector Machine (SVM). A high accuracy is obtained, either on audio with a single effect or with multiple cascaded effects as it is often the case in reality.

[1] The use of that term is not systematic in the literature and can sometimes be included in black-box modelling.

A similar approach had been proposed in 2008 in [16] enabling automatic classification of guitar sounds, suggesting that fitting an SVM on audio features is well suited for audio classification tasks. Finally, should the situation arise where the processing unit is available but needs to be classified, [17] propose several input-output measurements that can be used to extract features that directly identify effect classes. Similarly, [18] directly builds upon [14] and demonstrates which features are most important when it comes to recognizing guitar effects. In particular, this paper studies the use of a *Bag of Audio Words* approach, inspired by Neural Language Processing.

3.2.2 Parameters estimation

After identifying the effect used on a processed sound, it can be useful to retrieve the parameters that were configured to obtain that sound. This topic has also been studied thoroughly in the literature like in [19] where the authors aims at reproducing the audio mastering process of a musical track — the mastering is the last step before releasing a song, it consists in minor corrections in loudness and equalization to ensure the track will sound correctly on any listening device. The implemented network predicts coefficients based on the input magnitude spectrum to apply dynamic range compression that emulates professional mastering. That prediction of parameters for mastering is a way to allow beginners to quickly obtain correctly mastered tracks without having to practice and understand complex techniques while still enabling learning and improvements from experts because the parameters are meaningful and can be modified afterwards. Similarly, in [20], the authors design an algorithm to extract features from a reference sound which are then used as input to a regression model for predicting the parameters of a dynamic range compressor. The overall system is trained on a feature loss and the features of the original input sound are also computed and given to the model for better adaptability.

Some papers focus particularly on equalization, such as [21] where the author uses a neural network to predict the parameters of a parametric equalizer. The network is trained on a spectral loss instead of the usual parametric loss, a choice that appears to give perceptually closer results to the reference sound. Besides, this paper uses a differentiable implementation of biquad filters that permits training the network in an end-to-end fashion. Shallow[2] neural networks can also be used to ease the process of fitting the gain response of a graphic equalizer more efficiently than traditional optimization methods, as studied in [22, 23].

Not unlike the objective of this internship, [24] design a neural network that recognizes digital implementations of effect pedals and retrieve their parameters from audio spectrograms. The authors focus on non-linear effects and show that a *convolutional neural network* can retrieve the two parameters of the studied effects with little error. [25] adresses almost the same problem with the difference that it does not focus specifically on non-linear effects but rather tries to retrieve the parameters of one effect of each "main category": distortion for non-linear effects, delay for ambient effects and tremolo for modulation effects. Besides, the authors choose to manually design features for each effect and directly use a shallow network to process them instead of using a bigger convolutional neural network. Interestingly, it appears that this approach yields comparable results to [24], showing that adding *a priori* knowledge to an algorithm can greatly reduce its size and computational load.

Automatically retrieving the parameters of audio effects is closely related to finding synthesizer configurations to obtain a desired sound. Such a task can indeed be complicated since most synthesizers have several dozens of parameters and the sound space exploration is often restricted to using presets, which are long to create. Neural networks can be used to quickly retrieve parameters for sound matching, and can be more efficient than exploration algorithms like genetic algorithms, as it is presented in [26]. Focusing on the idea of improving user experience, [27] proposes a helper tool called *Synthassist* to find a sound by vocal query. The user imitates the desired sound and, through features analysis and user feedback, the system returns parameters for the synthesizer to produce the desired sound.

[2] A network is called *shallow* when it has few hidden layers, as opposed to *deep* neural networks

This proposition can be linked to [28] where the authors offer to greatly simplify the process of exploring the capabilities of a commercial synthesizer. Through *Variational Auto-Encoders* and *Normalizing Flows*, audio examples generated by the instrument are mapped to a latent space which is itself linked to another space representing the way parameters affect the sound. Such representations allow to easily find parameters to obtain a desired sound, for instance by vocal query; or to suggest new sounds resembling a first one based on their proximity in the latent spaces.

3.3 Differentiable audio effects

Audio effects have many existing implementations but are usually not differentiable, meaning that it is not possible to mathematically predict how a change in the input and the parameters will affect the output. While it is common and not a problem to musicians who will quickly understand the effect's variations after a few listening experiments, it is an issue to neural networks that need quantitative information on how to change the parameters' prediction to minimize the final error between two audio files. To enable such training, the audio effects have to be made differentiable. As it is summarized in [29], several approaches exist to do so, the first one being to manually implements a differentiable version of the effect. This is for instance what is done in [30] where the authors propose a tool to realize automatic DJ transitions between tracks. Such a task usually requires (at least) an *equalizer* and *faders*[3] and the researchers have therefore implemented those processors in a differentiable manner based on their knowledge of the signal processing principles involved. This approach, like white-box modelling for effects emulation, is powerful but requires thorough knowledge of the studied systems, making it usually harder to implement. Another paper using that technique is [31]: this paper introduces a differentiable implementation of an additive synthesizer that is trained to reproduce the sound of a reference recording. The parameters' estimation network is trained on a combination of a parametric loss and a spectral loss between the reference sound and the produced output. Besides, the authors use two different datasets to improve their system performance: an *in-domain* dataset consisting of sounds produced by the synthesizer itself and an *out-of-domain* dataset containing sounds produced by other synthesizers. The first dataset allows the estimator network to learn how the additive synthesizer's parameters affect the audio produced while the second dataset is used to help the system generalize to unseen sounds which it might not be able to actually produce, in which case the closest match is to be attained.
Instead of manually implementing a processor that is exactly differentiable, another possibility is to approximate the gradients of the effect being used. This is the approach chosen in [1] where the effect is considered as a black box and its gradient is obtained by **Simultaneous Perturbation Stochastic Approximation (SPSA)** which consists in randomly adding a perturbation to the input parameters of the effect and computing the gradient of the output with respect to these parameters. This technique appears to be an efficient alternative to **Finite Differences**, reducing computational load without compromising the quality of the estimate too much. Using this approach, the authors can train end-to-end models to complete tasks such as *tube amplifier emulation, non-speech sound removal or automatic music mastering.* In particular, it should be noted that the proposed system still permits user control because the predicted parameters are inputs to actual audio plugins that can be used in a DAW.
Finally, the last possibility to make an effect differentiable is to use a **neural proxy**, *i.e.* a neural network that is pre-trained to match the response of a chosen effect, just like what can be done for *black-box modelling*. Because the effect is reproduced by a neural network, its processing is differentiable and the effect can thus be part of a larger network that needs to be trained on an audio loss. For instance, Steinmetz *et al* present in [32] a framework for automatic multitrack mixing which predicts parameters for a neural proxy of a typical mixing effects chain: *Equalizer*, *Compressor* and *Reverb*. In order to use a custom loss function compatible with stereo audio, the effect chain must be differentiable so that the model can learn how the effects modify the sound. To do so, they train a neural network to emulate the processing of a real effect chain with added conditioning in order to have a final model that can be controlled more easily through common parameters even though it is a blackbox.

[3]For DJs and sound engineers, a fader is the sliding knob that controls the volume of a track.

Chapter 4

Audio effect Recognition

In this section we present our work on task of recognizing audio effects from audio. We begin with presenting the dataset used and the features considered and then compare several classification algorithms. We finally evaluate the results of our models on different subsets and variations of the original data.

4.1 Data

4.1.1 Dataset

The classification task is performed on the Fraunhofer IDMT-SMT Audio Effects dataset[1] presented in [14]. This dataset, specifically created for the classification task in the accompanying paper, consists of recordings of bass and electric guitars playing each single note on the fretboard from the open string (that can be considered as the 0th fret) to the 12th fret. Those recordings of 2 seconds each (with approximately 0.5 second of silence at the beginning) have a 44.1 kHz sample rate and have been repeated three times with different settings. Each recording is then processed through an audio effect unit belonging to one of the Effect class presented Figure 4.1 with three different parameter settings. These effects are digital, the data augmentation process being done with a DAW.

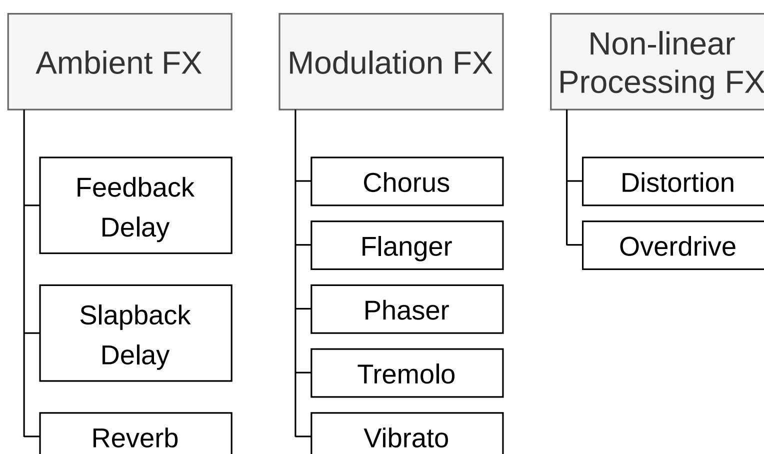


Figure 4.1: Effect classes in the dataset. Figure reproduced from [14].

Sound examples from the dataset are available on the accompanying webpage:

https://adhooge.github.io/AutoFX/

This dataset also includes polyphonic guitar sounds obtained by combining up to five single note recordings (in accordance to common chord fingerings) for further data augmentation.

[1]The dataset is freely available online: https://www.idmt.fraunhofer.de/en/publications/datasets/audio_effects.html

4.1.2 Features

For the task of audio Fx recognition and classification, I have implemented the framework presented in [14]. The classification is feature-based so, to that end, I compute the Short-Time Fourier Transform (STFT) of each sound and compute features from the magnitude spectrum.
Let N be the size of the Fourier transform, M the number of frames and F_s the sampling rate. The amplitude of the k-th frequency bin at frame m is hereby noted $A_m[k]$ with $0 \leq m < M$ and $0 \leq k \leq \frac{N}{2}$ where N is chosen to be a power of 2 to benefit from the Fast-Fourier Transform (FFT) algorithm.

Considering the spectrum as a probability distribution which values are the frequencies

$$f[k] = \frac{k.F_s}{N} \tag{4.1}$$

and probabilities are the normalized amplitudes

$$a_m[k] = \frac{A_m[k]}{\sum_{k=0}^{(N+1)/2} A_m[k]} \tag{4.2}$$

I obtain the following features for each frame m:

- *Spectral Centroid:*

$$\mu_m = \frac{2}{N+1} \sum_{k=0}^{N/2} f[k] a_m[k] \tag{4.3}$$

- *Spectral Spread:*

$$\sigma_m^2 = \frac{2}{N+1} \sum_{k=0}^{N/2} (f[k] - \mu_m)^2 a_m[k] \tag{4.4}$$

- *Spectral Skewness:*

$$\mathcal{M}_{3,m} = \frac{2}{N+1} \sum_{k=0}^{N/2} (f[k] - \mu_m)^3 a_m[k] \tag{4.5}$$

- *Spectral Kurtosis:*

$$\mathcal{M}_{4,m} = \frac{2}{N+1} \sum_{k=0}^{N/2} (f[k] - \mu_m)^4 a_m[k] \tag{4.6}$$

Example values of those features extracted on a distorted guitar sound are presented Figure 4.2. Those features capture the energy repartition and the overall shape of the magnitude spectrum, which can help identify different sounds.

I also compute the *Spectral Slope* which is the slope of the linear regression of the spectrum, defined by:

$$\text{slope}_m = \frac{\frac{N+1}{2} \sum_{k=0}^{(N+1)/2} f[k] a_m[k] - \sum_{k=0}^{(N+1)/2} f[k] \sum_{k=0}^{(N+1)/2} a_m[k]}{\frac{N+1}{2} \sum_{k=0}^{(N+1)/2} f^2[k] - \left(\sum_{k=0}^{(N+1)/2} f[k]\right)^2} \tag{4.7}$$

Another feature that can be used to study the energy repartition in a spectrum is the *spectral roll-off*. It is defined as the frequency so that 95% of the signal's energy is contained below that frequency. Intuitively, this feature could help recognize distortion sounds because their added harmonics should increase the energy in high frequencies, therefore increasing the *spectral roll-off*. An example result is shown Figure 4.3.

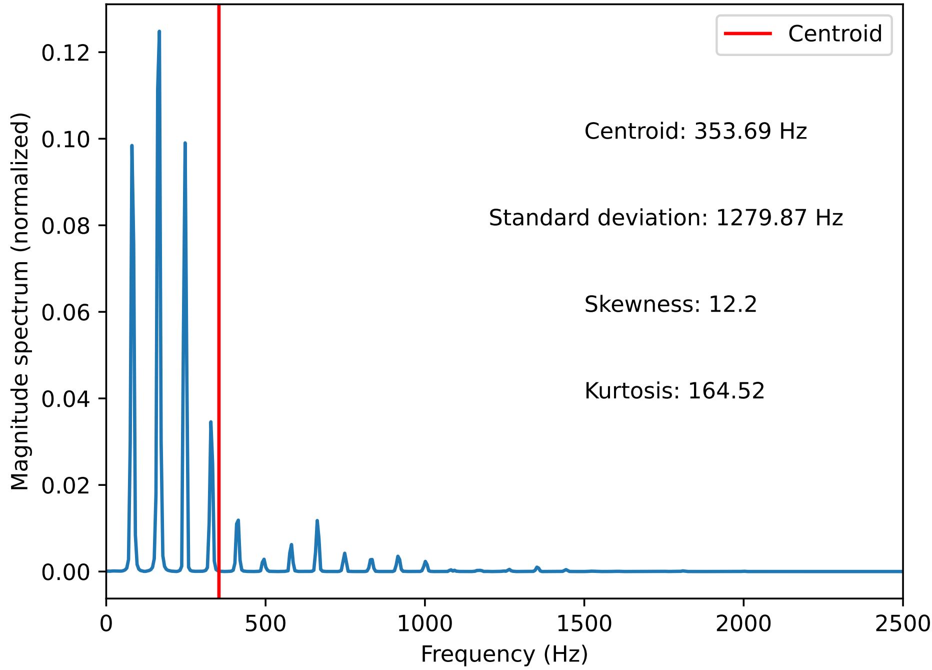


Figure 4.2: Spectral moments extracted from a distorted guitar sound.

The amount of frame-to-frame energy fluctuation in time, called the *spectral flux*, is also relevant to compute and can be defined by the following formula:

$$\phi_m = \sum_k \sqrt[1/q]{|A_m[k] - A_{m-1}[k]|^q} \tag{4.8}$$

where q is an arbitrary power factor. This feature can for instance help identify *tremolo* effects that will affect the amplitude faster than the natural decay of the notes.

I also compute the *Spectral flatness* of the spectrum, that quantifies its noisiness. It is defined as the geometric mean of the spectrum over its arithmetic mean.
Finally, those features are accompanied by the first 10 Mel-Frequency Cepstral Coefficients (MFCCs) obtained by a 128-bands *Mel-Spectrogram*. These coefficients are related to the inner periodicity of the magnitude spectrum and should be related to perceptual characteristics because of the use of the Mel scale. That scale is based on psychoacoustics measurements and link a mel-frequency m to a hertzian frequency f according to the following formulas:

$$m = 1127 \ln\left(1 + \frac{f}{700}\right) \tag{4.9}$$

$$f = 700 \times \left(\exp\left(\frac{m}{1127}\right) - 1\right) \tag{4.10}$$

Those features and others that can help recognizing timbre are described in further details in [33, 34].

4.1.3 Functionals

The previous features are obtained as frame-wise vectors. They could be used directly as input to the classifier but this would require a very large network which would be difficult to fit. Besides, since there is only one correct answer for the entire sound, using frame-wise values is not necessarily relevant. Instead, it is more efficient to use functionals, *i.e.* functions to reduce the frame-wise vectors to scalars. Those can be very simple such as `max`, `min` or `average`.

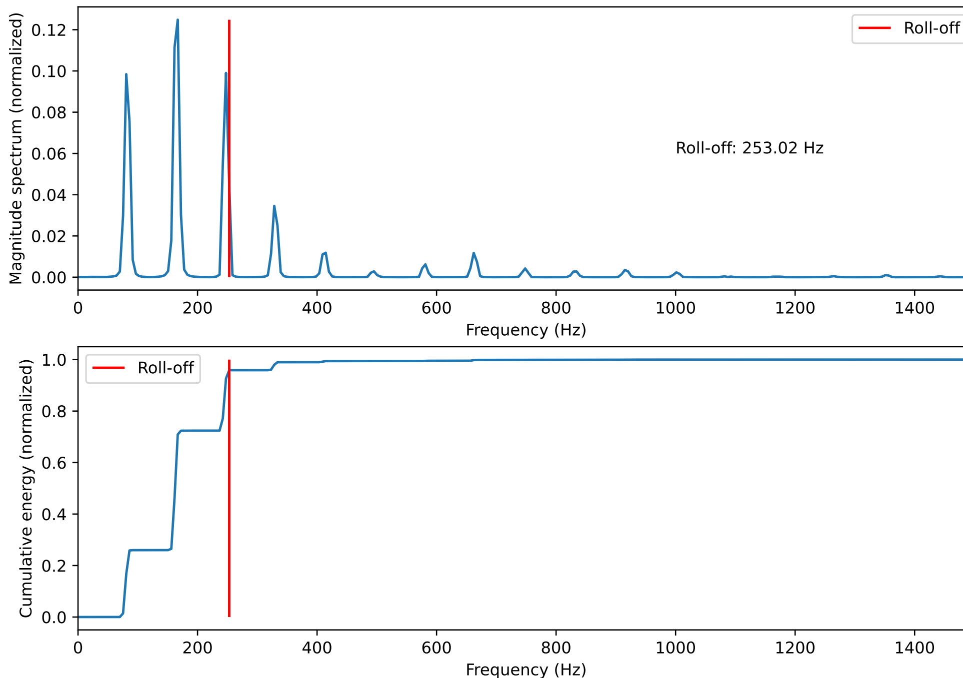


Figure 4.3: Spectral roll-off example for a distorted guitar sound.

They can also be more complex to capture temporal variation for instance doing *linear* or *quadratic regression* or retrieving the *maximum FFT frequency bin* of a feature.
For effect recognition, I take the first four moments – **mean, variance, skewness, kurtosis** – of each feature considering its time evolution as different probabilistic realizations of a random variable. I also compute the minimum and maximum values of each feature.
For the MFCC, I only take the frame-wise mean and maximum of each coefficient.

To capture more information, the delta coefficients $\delta f_m = f_{m+1} - f_m$ of each feature f for $0 \leq m \leq M - 1$are also computed before applying functionals. Besides, to eliminate the dependence to pitch of the spectral moments (the higher a note, the higher its spectral centroid, in most cases), they are also duplicated and normalized by the pitch of the note before going through the functionals.

Applying those functionals on the features yields a vector of 163 values for each data sample, a summary of that feature vector is given Table B.1. Computing those features takes approximately 35 ms for a 1.5 s (almost 43× real-time) sound with a sampling rate of 22 050 Hz.

4.2 Implemented Architectures

First experiments were done using the `scikit-learn` framework for quick implementation and testing. Building upon what is done in [14], I implemented a k-Nearest Neighbors classifier, a Support Vector Machine and a Multi-Layer Perceptron. Those experiments were used as a proof of concept and guided the choice of the final solution for the proposed application. Main results on the monophonic guitar sounds (Guitar Mono, GM) and the complete dataset (Full) are summarized Table 4.1.

These results are not as good as what is obtained in [14] but we use less features than in that paper and do not perform any feature reduction before training. Since this was only a preliminary test, we did not focus on further improving its accuracy. That first experiment already shows that a k-NN is not accurate enough even though it is fitted extremely fast. The SVM and the MLP perform similarly with a slight advantage to the SVM algorithm that is also fitted faster on the dataset.

		k-NN	SVM	MLP
Guitar Mono	Accuracy (%)	67.1	91.7	89.6
	Fitting time	3 ms	6.4 s	12 s
	Inference time	3.6 ms	0.32 ms	0.074 ms
	Model size	21.6 MB	6.9 MB	429 kB
Full dataset	Accuracy (%)	62.7	92.1	90.7
	Fitting time	7 ms	75 s	93 s
	Inference time	9.5 ms	2.5 ms	0.087 ms
	Model size	57.8 MB	20.2 MB	434 kB

Table 4.1: Summary results of preliminary classification experiments on a left out test set.

However, it appears that the MLP is the fastest when it comes to inference time, while also being the lightest when saved (using the Python's library `pickle`). This is likely due to the fact that a k-NN or a SVM must store data points to classify new samples. This consequently increases both the required memory space for saving such a model and the inference time because the stored data samples have to be compared to the new input. On the other hand, training a neural network like the MLP initially costs more time and energy but, once it is trained, only the resulting weights need to be saved, making the final model very lightweight and fast. Finally, this experiment also shows that, except for the k-NN model, the accuracy increases slightly when using the bigger dataset. It could be due to the fact that the SVM model benefits from better support vectors while the MLP takes advantage from having more training data.

The final objective being to implement the proposed tool as a DAW plugin, it was important to be able to compile the trained model for later use. Though this can be done in `scikit-learn` through the ONNX standard[2], we reimplemented the classifier in `pytorch` for easier compatibility with the rest of our work and because compiling models to use them in C/C++ is quite straightforward in `pytorch` using **TorchScript**. Besides, since performance were similar between SVMs and MLPs and because it is way easier to implement a perceptron than a SVM in a neural network framework – even though it can be done [35] – I reimplemented the classifier as a Multi-Layer Perceptron with one hidden layer in PyTorch. A summary diagram of the network is shown Figure 4.4.

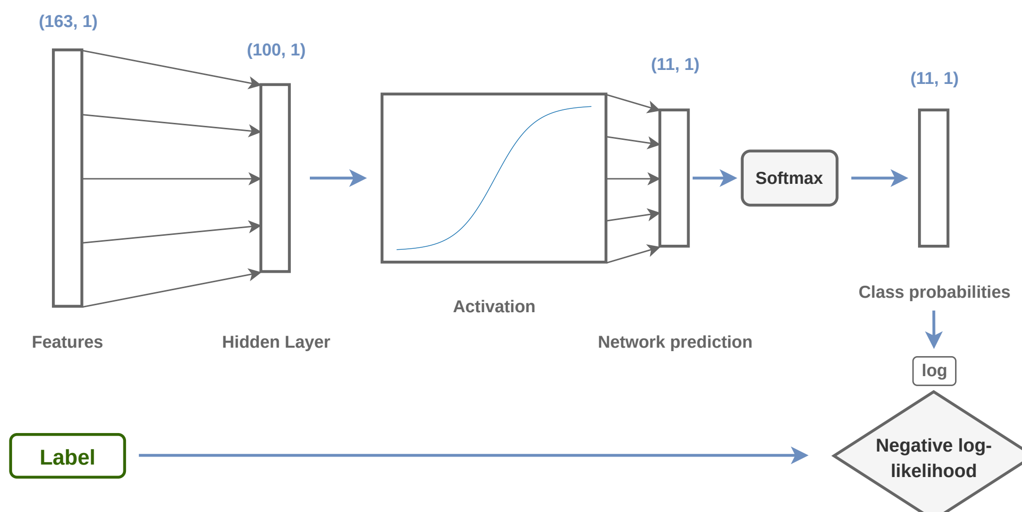


Figure 4.4: Summary diagram of the Classifier network.

[2] `onnx.ai`

The model is trained on the Audio-effects dataset. All audio are cut to their onset using a fixed energy threshold attack detection method [33], normalized in amplitude and resampled to 22 050 Hz. Before training, all features are scaled to have zero-mean and unity standard deviation. The scaler is kept for later inference since all inputs need to be scaled in the same way for the classifier to function properly.
The network uses a *Sigmoid* activation and an Adam optimizer with $(\beta_1, \beta_2) = (0.9, 0.999)$ and a learning rate $\eta = 0.002$. Batch size is 256 and I use a **Negative Log-Likelihood loss**. The output layer of my network is a Softmax layer which, given a n-dimensional vector $\mathbf{x}$, will rescale it so that all values are between 0 and 1 and sum to 1 according to the following equation:

$$\text{Softmax}(\mathbf{x})_i = \frac{\exp x_i}{\sum_{j=1}^{n} \exp x_j} \quad \text{for } 1 \leq i \leq n \tag{4.11}$$

This rescaling allows considering the output directly as a probability vector of the sound belonging to each class.
Then, I compute the log of the probability vector to apply the Negative Log-Likelihood function defined as:

$$\mathcal{L}_{NLL}(\mathbf{y}, \log \mathbf{p}) = -\frac{1}{N} \sum_{n=1}^{N} \log \mathbf{p}_{n,y_n} \tag{4.12}$$

Where $\mathbf{y}$ is the target vector containing the class index for each input sample, $\mathbf{p}$ is the probability matrix returned by the network and N is the *batch size*. This definition holds when using an mean reduction over batches.
Combining the *softmax* layer to the NLL-loss, we obtain a *log-loss* or *cross-entropy* loss defined as:

$$\mathcal{L}_{\log}(\mathbf{y}, \mathbf{p}) = -\frac{1}{N} \sum_{n=1}^{N} \sum_{c=1}^{C} \mathbf{y}_{n,c} \log(\mathbf{p}_{n,c}) \tag{4.13}$$

where C is the total number of classes.

For training, validation and testing, I use a 0.8-0.1-0.1 split of the entire dataset which corresponds to approximately 500 sounds per class in *validation* and *test* and around 4500 sounds per class in *Train*. The parameters were optimized using **9-fold cross-validation**. This means that the remaining data after removing the test dataset is split into 9 random blocks, one is kept as a validation dataset and the network is trained on the eight remaining blocks. After doing a complete training and evaluating the network's performance on the validation set, the process is repeated using another block as the validation set until 9 iterations have been done. This technique is a way to present more robust results because it allows to mitigates the assumption that the network might have gotten "lucky" because the training set was particularly helpful or the validation set somehow "easy" to classify. Repeating this experiment with different hyper-parameters allow to choose the best values for each of them.

4.3 Results

4.3.1 Complete version

The first implementation of the classifier was similar to what is done in [14] and would classify samples in the 11 effect categories that are present in the dataset. Implementations have been done on either the complete dataset or only monophonic guitar sounds. Training could last up to 100 epochs with a possible early-stopping if the validation loss did not decrease for 10 epochs. Results for both experiments on the test set are shown Figure B.1 as *confusion matrices* where each sample is shown on a row corresponding to its actual label and a column corresponding to the predicted label. Therefore, the higher the values on the diagonal of the confusion matrix, the better the classifier is.

The results show that a high accuracy is attained in both experiments and that modulation effects are the hardest to classify, some confusion happening between *Chorus, Flanger, Phaser and Vibrato*. This is not surprising since those effects can sound similar and are based on similar physical principles.

4.3.2 Simplified Version

It has been shown that the classifier performs well on the entire dataset. However, the rest of that internship adresses a reduced problem due to time and difficulty constraints. Indeed, further work is restricted to *monophonic guitar* sounds and to aggregated effects class due to implementation choices that are explained in subsection 5.3.5. That aggregation is summarized in Table 4.2

Fx class	Aggregated Fx class
Dry	Dry
Feedback Delay Slapback Delay	Delay
Reverb	Reverb
Chorus Phaser Flanger Vibrato	Modulation
Tremolo	Tremolo
Overdrive Distortion	Distortion

Table 4.2: Aggregated effects class of the original dataset.

As one would expect, better results are obtained on this simpler problem, as can be observed on the confusion matrices Figure B.2. An accuracy of up to 97% is reached on the monophonic guitar set.

4.3.3 Energy consumption

To refine the analysis of the models' performance, I monitor their energetical consumption using two Python libraries: `CarbonTracker`[36] and `pyRAPL`[37]. These libraries allow tracking the energy used by the Central Processing Unit (CPU) and the Graphics Processing Unit (GPU) while running code. The energy consumption for training each model is given Table 4.3. Training was conducted on a NVIDIA GeForce GTX 1080 Ti GPU.

Model	Training time (s)	Energy consumption (Wh)
Complete GM	120	5.2
Complete Full	255	13
Simplified GM	96	3.8
Simplified Full	208	10

Table 4.3: Training time and energy consumption for training the studied models. Complete refers to training on all 11 effect classes while simplified refers to using aggregated classes.

Those models are very light and energy-efficient since their consumption during training is comparable to a low-energy LED light[3] lit up for an hour. However, this analysis does not take into account the fact that the features of the training dataset have to be computed, which also requires energy.

[3] A LED light usually has a power of $\sim 10\,\mathrm{W}$, `https://izi-by-edf.fr/blog/led-rentabilite/` (in French)

Indeed, `pyRAPL` measurements suggests that processing the monophonic guitar set requires 13 Wh, or 48 Wh for the full dataset (even though those values are probably higher than reality because they include the consumption of all software running on the computer like the Operating System or the Integrated Development Environment - IDE - used). The computation of the features indeed does increase the overall consumption of the models up to 6 times but the features only need to be computed once while the training is likely to happen multiple times for hyper-parameters tuning for instance.

4.3.4 Feature Importance

Even if the classification results are ultimately what really matters, it is interesting to study which features have the most impact on the classification task. To assess such impact, I perform *Random Feature Permutation* which is a technique suggested in [38] consisting of evaluating a fitted model on a test set while randomly exchanging values of a feature between samples. Doing so is supposedly better than removing the feature entirely or setting it to an arbitrary meaningless value because it allows keeping the fitted model unchanged while ensuring that the values still fill the expected range. Even though the exact values differ between the evaluated models: Full dataset or Guitar Mono, Aggregated classes of original classes; the observed trends are similar and commented hereafter.

On average, permuting a feature leads to an accuracy drop between 1% and 2% and similar drop in each class precision or recall. However, a deeper analysis shows that some features appear responsible to 15% up to 20% of the model's accuracy. This is the case of the *averaged spectral flux*, the *averaged spectral flatness* or the *kurtosis of the δ spectral flux*.

When looking at Precision and Recall for more details, it appears that all classes do not react equally when a feature is shuffled. The Distortion and Overdrive classes – or the corresponding aggregated class – are fairly robust to feature permutation and drop at worst by $\sim 5\%$ when the *averaged spectral flux* is shuffled. Other classes are more sensitive and shuffling their most important feature lead to metrics dropping by at least 30% up to 55%. While the feature may vary from one effect to another, analyzing the top values regarding metrics loss shows that the *spectral flux* (its average in particular) and the *δ spectral flux* (especially the kurtosis and skewness) are highly important. The *spectral slope* and its frame-wise variation also appear meaningful though to a lesser end, along with the *averaged spectral centroid* and the first three MFCCs.
Interestingly, shuffling some features increases the accuracy of the model. This should not be related to the random shuffling since those values stay negative when averaging metrics over 10 repetitions of the permutation procedure. It so appears that shuffling the *spectral skewness normalized by pitch* could increase the accuracy by 0.1%.

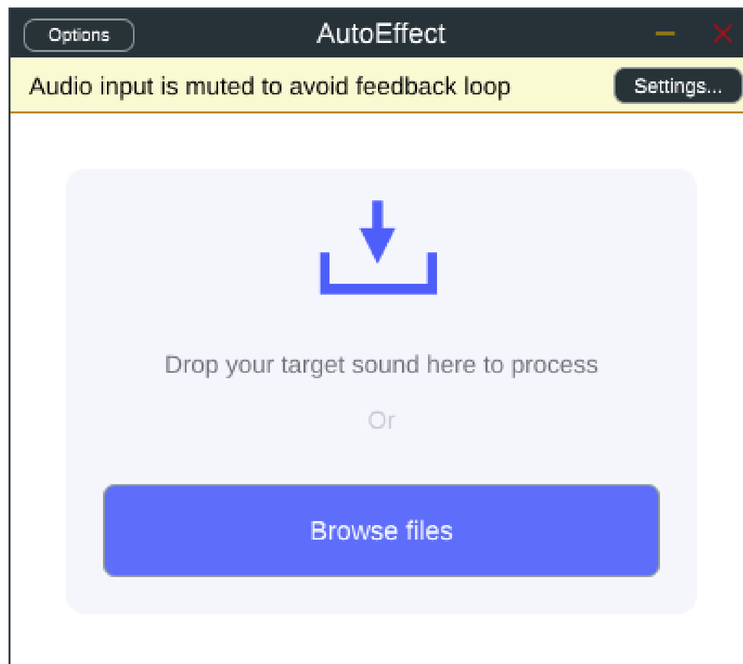
Because these observations are made on the test set, it is not possible to directly predict how removing a feature before training would impact the results. However, it appears that some features are clearly more important than others in this task, that the most relevant features for an effect class are not necessarily identical to those of another class and finally that some features could be removed with no performance drop or maybe even an increase on some metrics.

4.4 Compiled version

Since the final objective of that internship is to obtain a plugin implementing the proposed tool, we needed to compile the classifier model to be able to integrate it to a C++ program. This required reimplementing all feature extraction functions in Pytorch and ensuring they are compatible with **Torchscript**. The feature computation was at first done using `librosa` [39] which cannot be compiled using Pytorch's utility. Once the implementation could be compiled, we chose a pre-trained model to load in a JUCE[4] program.

[4] JUCE is a framework for designing audio plugins https://juce.com/

The JUCE plugin was designed and coded by Hugo PRAT and allows to drag-and-drop a sound file which is then processed by the classifier, its output being used to instanciate the detected effect, as shown Figure 4.5.



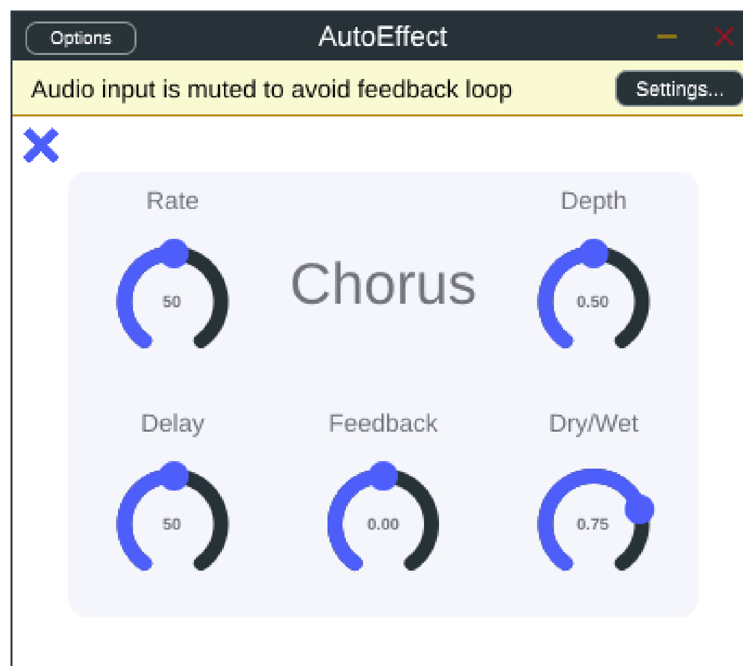


Figure 4.5: Home screen of the JUCE plugin (top) and the resulting interface after a sound has been processed (bottom).

Chapter 5

Estimation of effect parameters

5.1 Proposed approach

This internship aims at designing a tool that estimates effects' parameters to reproduce the timbre of a reference sound. However, the main difficulty is that the effect actually used in the reference sound is supposedly unknown and in particular not available for upcoming processing. This prior hypothesis motivates the fact that I choose specific plugins that aims at emulating any other effect unit of the same type. This directly yields the following assumption: the chosen solution can emulate any effect belonging to its effect class. Such an assumption can be criticized but is necessary to reduce the task's complexity and will be assessed by a perceptual experiment in subsection 5.4.3.

The chosen effect proxies are taken from the `pedalboard` library [40] that implements **JUCE** audio processing plugins (among other features) in Python. The study is conducted on a reduced set of effects to reduce the task's complexity but one effect for each main effect category is implemented to evaluate the generalization capabilities of the chosen approach.

The first plugin used is `pedalboard.Chorus` that obviously implements a Chorus effect but not only, as the documentation states:

> To get classic chorus sounds try to use a centre delay time around 7-8 ms with a low feedback volume and a low depth. This effect can also be used as a flanger with a lower centre delay time and a lot of feedback, and as a vibrato effect if the mix value is 1.

For this reason, this single plugin is used to emulate Chorus, Flanger, Vibrato and Phaser effects – even though it is not mentioned in the documentation Phaser have been added because of its perceptual similarity to Flanger. The plugin can be controlled by five parameters:

- `rate_hz`: the rate of the oscillator modulating the delayed signal, from 0.1 Hz to 10 Hz;
- `delay_ms`: center delay in milliseconds of the modulated signal, from 0 ms to 20 ms;
- `feedback`: volume of the delayed signal $\in [0, 1]$;
- `depth`: amplitude of the modulation signal $\in [0, 1]$;
- `mix`: from 0 for full dry to 1 for full wet.

Both Slapback and Feedback Delay are reproduced using `pedalboard.Delay` which features three control parameters:

- `delay_seconds`: the time delay between each echo, from 0 to 1 s;
- `feedback`: the volume ratio between each echo and the previous one $\in [0, 1]$;
- `mix`: the processed/dry signal ratio of the output signal $\in [0, 1]$.

It is expected that such a plugin will have difficulties emulating delays with hard slapbacks (a single repetition that quickly fades out) because it is not designed to do so. We still use that approach in the lack of a better one.

Finally **Overdrive and Distortion** are emulated using a combination of plugins. The library `pedalboard` includes a `pedalboard.Distortion` plugin, but it is simply a hyperbolic tangent *waveshaper* with a `drive_db` parameter controlling the input gain. Though efficient for a simple distortion effect, such an implementation fails to reproduce the wide variety of distortion effects that exists. Inspired by the circuit analysis of famous distortion pedals [41] I propose to add two filters after the waveshaper, as shown Figure 5.1.

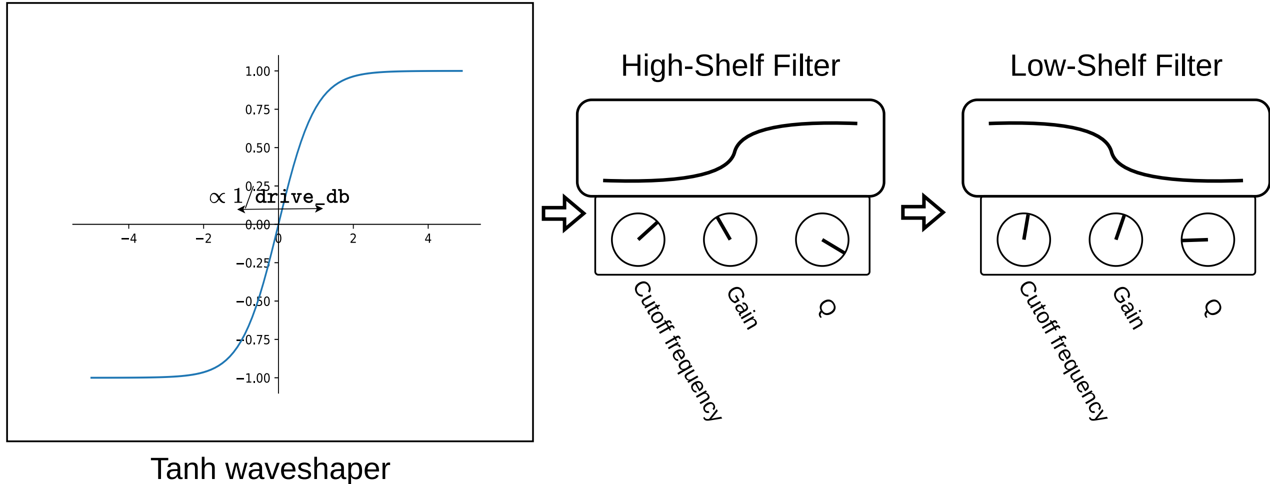


Figure 5.1: Proposed pipeline for Distortion emulation

The filters are shelf filters which do not completely cut high or low frequencies but rather alter the frequencies amplitude with a controllable resonance around a cutoff frequency switching from one gain value to another. This pipeline seems to successfully reproduce a variety of distortion sounds[1].

5.2 Data

To the best of our knowledge, no dataset exist for the task of retrieving audio effects parameters. To that end, I have designed a pipeline to generate data samples consisting of:

- a clean sound of a single note;
- a set of uniformly sampled random parameters;
- the corresponding processed sound.

For each parameter of each effect, a parameter range is manually defined to produce sounds that are "realistic" *i.e.* similar to what could be obtained from real processing units. For training and prediction purposes, all parameters are normalized to the range $[0, 1]$ and values are rounded to the hundredth since a finer resolution may only be useful to expert users. This first implementation focuses on monophonic guitar sounds for simplicity. All clean monophonic guitar sounds from the IDMT-SMT dataset (I include sounds only processed by an Equalization or Amp simulation) are used to produce processed sounds through data augmentation. This amounts to 1872 recordings for a total duration of approx. 47 minutes without silence. It is then very fast and easy to process the clean dataset with the data augmentation pipeline thanks to the `pedalboard` library: the entire clean set is processed in approx. 9 s which is more than 300× real-time. For each effect, the clean dataset is processed entirely 20 times, leading to 37440 files per Fx, or 15.6 hours of audio.

[1]Synthetic sound examples are available on https://adhooge.github.io/AutoFX/

It should be noted that each sample has random parameters so the dataset includes 37440 theoretically different set of parameters. We consider more important to explore a wide range of parameter values than having many different inputs going through an effect unit with the same parameters. The chosen approach could lead the network to make mistakes when a given parameter set is applied to notes of pitch or timbre not seen during training but results suggest that the chosen method for data augmentation is enough. For size and memory constraints, all files are downsampled from 44.1 kHz to 22 050 Hz, leading to a final dataset of 8.5 GB.

5.2.1 Added features

Inspired by [25] where the authors successfully retrieve effect parameters from a limited set of handcrafted features, we also extract features from the input samples to use them as added information in the implemented models.
To help the evaluation of modulation effects, I extract the **pitch-curve** of the input sound *i.e.* the detected fundamental frequency in frames of 0.01 s using `torchaudio`. I also compute the **unwrapped phase of the maximum frequency bin** from the spectrogram of the data sample, along with its frame-wise **Root-Mean Square (RMS) energy**. I accompany those features of their frame-to-frame fluctuations to study their evolution through time before applying several *functionals*:

- I extract the two highest values of the Fourier Transform of the padded feature vectors along with the corresponding frequency bins, this allows to detect if a feature is periodic, the second maximum being added for robustness in case a spurious detection occured;
- I compute the *Standard Deviation* and the *Skewness* of the RMS features as they have been observed to help retrieving delay parameters.

To further help retrieving parameters of *delay* effects, I retrieve five onsets and their corresponding activations using the *Superflux* algorithm presented in [42]. This method suppresses vibrato by tracking frequency peaks with a maximum filterbank instead of simply considering the framewise energy difference of each frequency bin.
Finally, to help retrieving distortion parameters, I compute the temporal average of the 10 first MFCCs. This amounts to 48 features that are summarized Table C.1.

For compatibility with the rest of my work, all feature computations are reimplemented in PyTorch and made compatible with parallel computing to limit any slowdown during training. Implementation details are available on the accompanying repository: `https://github.com/adhooge/AutoFX`.

5.3 Implemented architectures

5.3.1 Simple regression network

Since classification is done on audio features with a MLP, we also use a simple MLP as a baseline for our regression task. Experiments are done with a varying number of hidden layers and different configurations to assess how well such a simple model can perform. The tested implementations are shown Figure 5.2.

The features are those already computed for the classification task as described in subsection 4.1.2, additional features are the ones described in subsection 5.2.1 while the conditioning encodes additional information on the input sound, as it is explained subsection 5.3.4. The predicted parameters are represented as a single vector with 15 values containing the 5 *Chorus* parameters, the 3 *Delay* parameters and the 7 *Distortion* parameters. During training, the gradient is only backpropagated for the parameters of the actual effect class so that the network is not trained to zero the parameters of the unused effects. By doing so, we ensure that the computational power of the network is completely directed towards the task at hand.

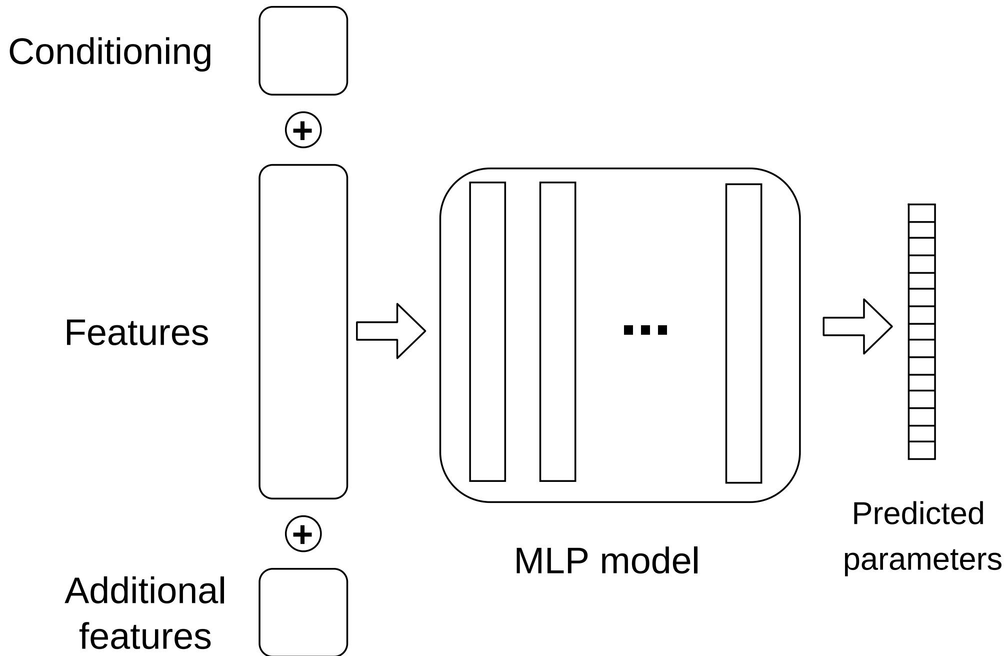


Figure 5.2: Tested implementations of a simple MLP network for regression. The additional features presented subsection 5.2.1 and optional conditioning can be added. The number and the size of the hidden layers can also be changed.

5.3.2 Convolutional Neural Networks

The Convolutional Neural Network (CNN) I implemented uses residual learning and is thus often called a Residual Network (Resnet) in the litterature. It has been proposed in [43] and enables easier training especially for very deep neural networks . The motivation to such residual learning, illustrated Figure 5.3, is assuming that by adding an identity connection between layers in the network it will focus on learning residual information instead of reproducing an identity function if necessary. We made the choice of using a Resnet instead of a simple CNN because it trained slightly faster at no supplementary cost.

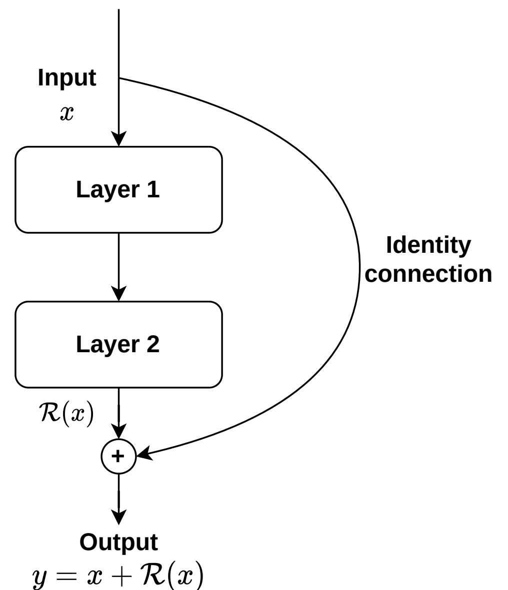


Figure 5.3: Principle of Residual networks. Here the identity connection skips a layer because this is what we have implemented and what is done in the original paper [43] but it could occur directly after each layer.

5.3.3 Training strategies

Supervised Learning

A first training approach is to implement **Supervised Learning**. Supervised learning refers to training procedures where a *correct* answer, or label, exists and is known beforehand. It is a form of training that is often preferred when available because having an actual ground-truth for the model's predictions allow to use a loss and a training objective that is directly meaningful to the task at hand. However, supervised learning requires a dataset with annotations that are most of the time written by human experts, making them costly and usually smaller than datasets without annotations.
Fortunately, in our case, the data is generated synthetically so it can be automatically labelled at no additional cost and without any size constraint.

For this reason, that training strategy is the main one we use in our work, for training on **In-Domain** sounds, *i.e.* sounds that are obtained from the effects being studied (similarly to what is done in [31]). Training on such sounds ensure that there actually is a *right answer* for the predicted parameters and that the system can reproduce the reference sound exactly.
In that situation, the loss we use is a Mean-Squared Error (MSE) **loss function** to ensure that the network minimizes the distance between its predictions and the actual parameters. We also monitor the absolute distance between each prediction and parameter value to assess if the quality of the regression depends on the parameters.

Unsupervised Learning

Consequently, **Unsupervised Learning** refers to a training procedure where no ground-truth exists for the input data samples. This situation is common when working with huge datasets and is usually the first phase of training a model, to benefit from the amount of data, before *finetuning* on a smaller dataset in a supervised manner to refine the model to the task studied. Here, we reverse the original procedure, using unsupervised learning on another dataset to finetune our model and hopefully enable better generalization capabilities.
The second dataset used is actually the same used for the classification task (see subsection 4.1.1) because it contains processed guitar notes for which the original clean sound is available. We call those sounds **Out-of-Domain** sounds because they have not been produced by the effects we implement, which implies that it might be impossible to reproduce their timbre. This terminology and training strategy is also inspired by [31] where the authors begin by training their model in a supervised manner on In-Domain sounds with a parameter loss (because the ground-truth values for the parameters are available). Afterwards, they gradually introduce a spectral loss to free themselves from requiring labels and finally train in an unsupervised manner on an out-of-domain dataset to make the model generalize to unseen sounds similar to what could be queried by an actual user.
In my case, the spectral loss I use is a Multi-Resolution STFT (MRSTFT) loss which was firstly introduced in [44] and included in the Python's library `auraloss` [45]. This loss has been introduced because changing the parameters of a STFT (like the window size, the number of frequency bins or the hop size) yields different spectrograms with varying time and frequency resolutions. For this reason, when comparing two sounds, it is interesting to see how the comparison holds at different scales. This is the principle of the MRSTFT loss that averages contributions of spectrogram losses

$$\mathcal{L}_{\text{mag}}(\mathbf{x}, \hat{\mathbf{x}}) = \frac{1}{N} \, ||\log|\text{STFT}(\mathbf{x})| - \log|\text{STFT}(\hat{\mathbf{x}})|||_1 \tag{5.1}$$

where N is the FFT size and $||.||_1$ is the L_1 norm. The MRSTFT loss is thus defined by:

$$\mathcal{L}_{\text{MRSTFT}}(\mathbf{x}, \hat{\mathbf{x}}) = \frac{1}{M} \sum_{m=1}^{M} \mathcal{L}_{\text{mag}}^{(m)}(\mathbf{x}, \hat{\mathbf{x}}) \tag{5.2}$$

with M the number of losses computed. In our implementation, a loss on the phase-spectrograms is also computed and the FFT sizes used are $[64, 128, 256, 512, 1024, 2048]$ with 25% overlap between frames, no zero-padding and a Hann window.

5.3.4 Effect class conditioning

In our work, the effect is classified before its parameters are inferred in order to reduce the quantity of information the Regressor Network must learn. Though the Regressor Network could be trained without any knowledge shared from the Classifier, improved performance can be attained through conditioning
One solution to generalize could be to train one network per Fx considered but this has several downsides. First of all, it is time and energy consuming because several networks would have to be trained for extensive durations. But, more importantly, the obtained networks would not even be guaranteed to work better than a single network trained on all Fx since **Multi-Task Learning** has been shown to improve results in a variety of use cases [46].

For this reason, we keep the original Regressor Network and use the Fx Class information as a conditioning input added to alter the produced output.

One simple way to add conditioning to a neural network can be to concatenate the new information to the original input, this is for instance how conditioning is added to the MLP used for regression. However, while this approach makes sense when the conditioning is added to a vector of features of rather short size, simply concatenating new information to input samples is usually not recommended. Indeed, for the CNN models, the input is a batch of audio spectrograms. In that case, one can see how blindly concatenating conditioning data to the input would be rather inefficient. The conditioning is a high-level information that should weigh more in the model's processing than the temporal values of frequency bins. Furthermore, applying a convolutional kernel on both a spectrogram excerpt and conditioning values will yield unpredictable results.

For this reason, we use Feature-wise Linear Modulation (FiLM) layers. Perez *et al.* introduced this technique in [47] for visual tasks where a convolutional neural network extracts features that are conditioned by a question-analysing recurrent network.
Let $\mathbf{C}$ be the conditioning input, it can be of any shape relevant to the task at hand. In our case, it will be a vector of probabilities from the classifier network representing how likely the sample is to belong to each effect class. The FiLM layer aims at deriving from $\mathbf{C}$ two arrays γ and β whose shapes are identical the output of the FiLM-ed layer of the original network. Let $\mathbf{F}$ be the feature-map obtained after a layer of the FiLM-ed network, it is modified by the FiLM layer like so:

$$\text{FiLM}(\mathbf{F}|\gamma, \beta) = \gamma \circ \mathbf{F} + \beta \tag{5.3}$$

or

$$\text{FiLM}(\mathbf{F}|C) = \gamma(\mathbf{C}) \circ \mathbf{F} + \beta(\mathbf{C}) \tag{5.4}$$

where $\circ$ denotes the element-wise product. The principle is to apply an affine transformation on the feature-map extracted by a network to zero some features, change their relative scaling or maybe even change their sign. Intuitively, the original network is expected to extract some features relevant to the task at hand from the input data and the FiLM layers are used to mitigate those features and modify them accordingly to conditioning information.

The overall implementation of the CNN, including the two possible training procedures and the added conditioning, is summarized Figure 5.4.

5.3.5 Classifier on synthetic dataset

On real sounds, the Fx class is known beforehand. However, on synthetic sounds — *i.e.* clean sounds processed by `pedalboard` — there is no properly defined Fx class. This is due to the fact that the effects we use to generate the synthetic data can emulate several different effect classes. For instance, the *custom distortion pipeline* we implement should reproduce *Overdrive* or *Distortion* sounds depending on the settings, while the `pedalboard.Chorus` plugin is used to produce *Chorus, Flanger, Vibrato* and *Phaser* sounds. To distinguish between those specific effects, we could manually define parameter ranges that are associated to specific effect classes, but this approach is difficult to implement because there is no clear transition from one effect to another.
To circumvent that problem, we use the pre-trained classifier to obtain the conditioning values for each sample. This is in accordance with the overall pipeline we want to implement where the sound is firstly classified before computing effect parameters.
Consequently, we began with directly classifying synthetic sounds with the classifier trained on all effects (but only monophonic guitar sounds since the synthetic dataset is only guitar mono) as it is presented subsection 4.3.1. The results, summarized Figure 5.5, show that many sounds were misclassified, causing the algorithm to instanciate the wrong effect.

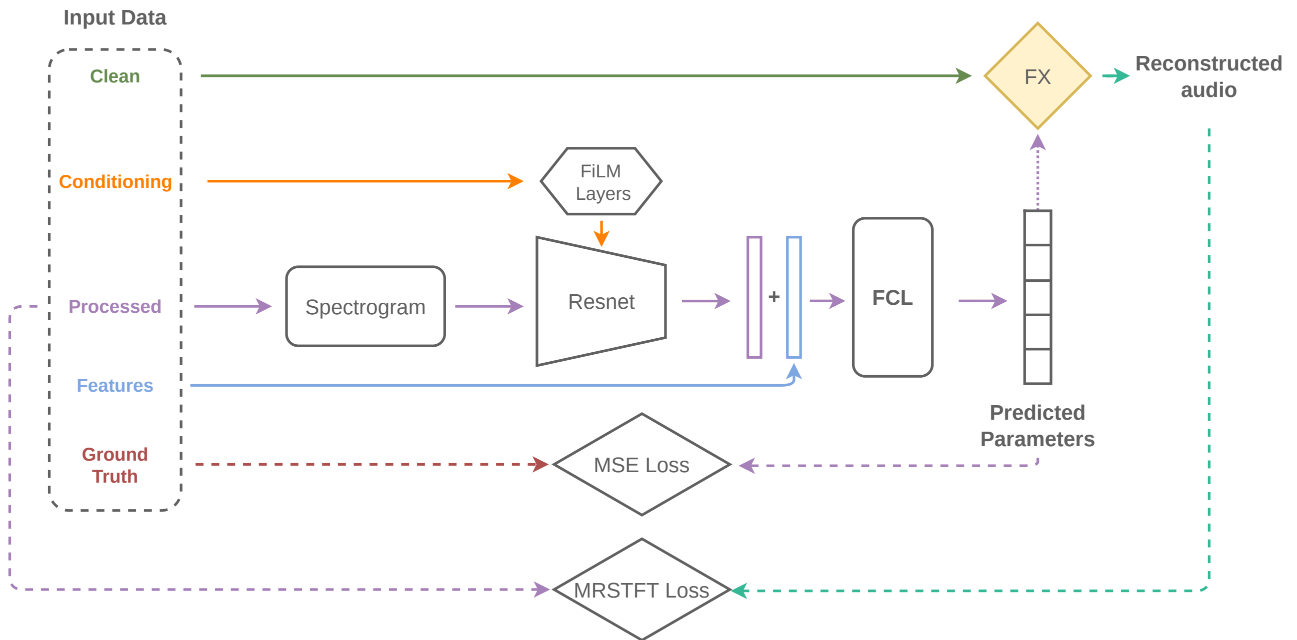


Figure 5.4: Summary diagram of the regression network's implementation with a ResNet. FCL stands for Fully-Connected Linear, a simple linear layer like the ones used in MLPs.

True label \ Predicted label	Dry	Feedback Delay	Slapback Delay	Reverb	Chorus	Flanger	Phaser	Tremolo	Vibrato	Distortion	Overdrive
Slapback Delay	2731	4253	6330	896	280	3833	16631	556	69	1844	17
Chorus	1510	5704	33	9768	1042	8127	4578	184	4828	1663	3
Distortion	3789	154	16	982	2181	6490	7971	306	271	15224	56

Accuracy=0.201

Figure 5.5: Confusion matrix of the classifier trained on all effects applied to the synthetic dataset. The synthetic effects are considered as a single class so Slapback Delay represents all Delay sounds, Chorus represents Modulation and Distortion includes all the sounds produced with the Custom Distortion pipeline.

To improve the classifier performance, we simplify the problem using *aggregated classes* in accordance to the way the synthetic data is generated. The aggregated classes are shown Table 4.2 and the confusion matrix of the classifier pre-trained on aggregated classes and applied to the synthetic dataset is shown Figure 5.6.

True label \ Predicted label	Modulation	Delay	Distortion	Reverb	Tremolo	Dry
Modulation	20569	4293	241	10194	115	2028
Delay	10131	20296	100	661	356	5896
Distortion	30787	338	481	941	282	4611
Reverb	0	0	0	0	0	0
Tremolo	0	0	0	0	0	0
Dry	0	0	0	0	0	0

Accuracy=0.368

Figure 5.6: Classification results of the classifier trained on aggregated classes (see subsection 4.3.2) on the synthetic datasets. The three bottom rows are empty because the synthetic dataset does not contain those effects.

Even in that simplified situation, many sounds are misclassified. Those results suggest that the classifier trained on the IDMT-SMT dataset fails to generalize to unseen effects. Indeed, while it is plausible that some samples are classified as *Dry* because the effects can be configured in a way that almost does not change the input signal, a majority of *Distortion* sounds are classified as *Modulation* while almost half *Delay* and *Modulation* samples are also misclassified. Nevertheless, those results are of interest because they suggest that the dataset presented in [14] might not include enough variety to allow generalization, or that the effects considered have specific characteristics that identify them and are not seen on other effects. The features of the IDMT dataset could also be restricted to a range that does not represent the reality of audio effects, causing new sounds to have inconsistent feature values when rescaled like the training sounds. This is an important observation because that dataset is the reference for the classification of guitar effects but in the absence of another similar datasets, the classifiers are usually not tested on completely new effects.

To still have conditioning for training the regression network, I train a new classifier on aggregated classes and on an unbalanced dataset that include synthetic sounds. Doing so solves the misclassification issue on synthetic sounds, as it can be seen Figure 5.7. The accuracy reaches 96.7% on the synthetic dataset and only a few misclassifications occur.

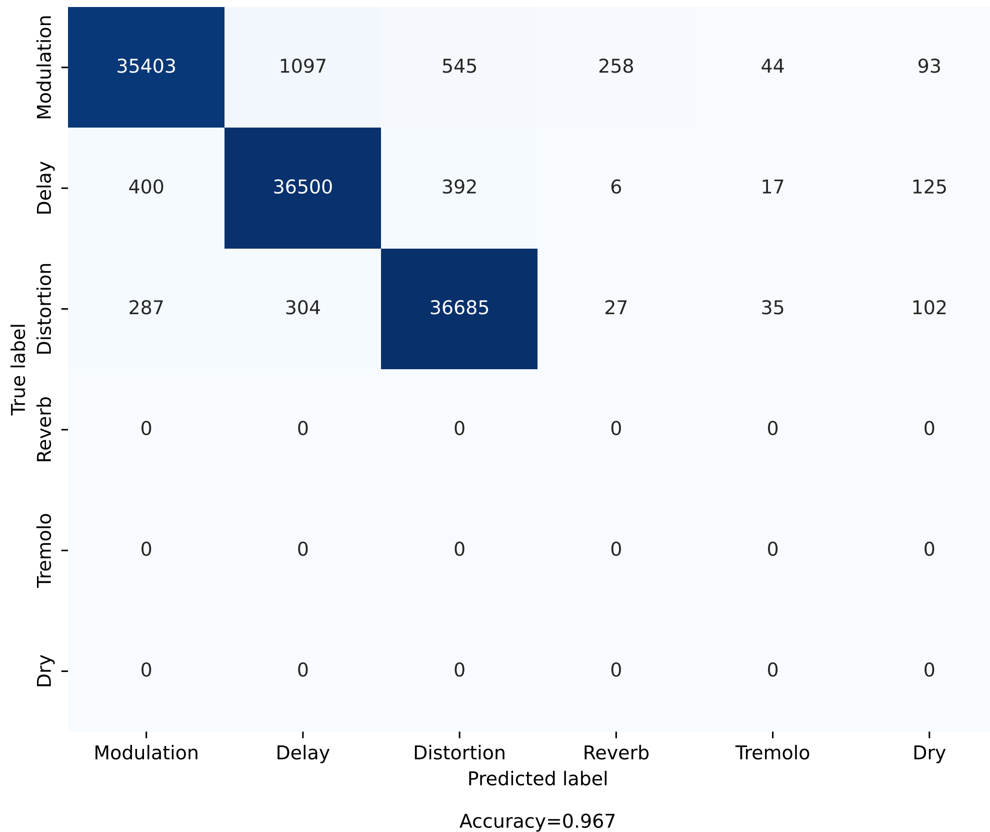


Figure 5.7: Confusion matrix of the classifier trained on the mix dataset for the synthetic data. The three bottom rows are empty because the synthetic dataset does not contain those effects.

5.4 Results

5.4.1 Simple MLP

The first experiments were conducted with the MLP network to have baseline results on the regression task under scrutiny. Several variations of the network have been tested, with a varying number of hidden layers, added conditioning or added features. All models were trained with a batch size of 64, a hidden layer size of 100, ReLU activation between layers and a final Sigmoid activation. The training ended when the validation loss stopped decreasing for 3 epochs.
We observed that the number of hidden layers does not really change the performance of the models but a model with 5 hidden layers trained slightly faster than models with 1 or 10 hidden layers so we set the number of hidden layers to 5 for further experiments.
Enabling conditioning and additional features neither had much impact on the training or validation losses. We decided to keep all three variations of the MLP to assess their performance in a perceptual experiment.

5.4.2 Convolutional Neural Network

The main network under study is a large *Resnet* dubbed **AutoFx**. Its architecture is summarized in Figure C.1. The training was done with a batch size of 64, an Adam optimizer [48] with a learning rate $\eta = 0.0001$ and $(\beta_1, \beta_2) = (0.9, 0.999)$ and the input spectrograms were obtained with an FFT size of 1024 with 25% overlap and a Hann window. The network is trained for 20 epochs with an MSE Loss on the parameters prediction.

Two main trainings are conducted: one with additional features and one without. A summary of their performance on the test set with the simple MLP for comparison is shown Table C.2.
The AutoFX networks have the best performance, in particular the version with added features which has the lowest MSE Loss. When it comes to individual absolute distance, all parameters are predicted with less than 0.1 error (except for `Q-hi` with one MLP model) which is the usual precision for tuning physical effect pedals. Interestingly, no model performs best on all parameters. The best model has a low prediction distance for all parameters and the best score for 7 out of 15 parameters but some parameters are better predicted by other models. Another noticeable result is that the lowest distance on the parameters ranges from 0.0065 (`rate`) to 0.066 (`Q-hi`) which suggests, as it was expected, that some parameters are harder to predict than others. Audio examples of reconstructions from the test set are available on the accompanying webpage [2].
Out of curiosity, we also observed the coefficients generated by the FiLM layer on the last convolutional layer of the network when changing the conditioning. The γ and β values are represented Figure C.2. While its impossible to deduce anything directly concerning the input audio with these representations, it appears that the conditioning has an impact on the FiLM layers and consequently on the implemented network. We observe that the proposed γ and β matrices are dependent on the conditioning effect: a *delay* conditioning leads to a positive bias and an almost entirely negative γ while this is the opposite for *modulation* and *distortion*. Besides, one can see that the conditioning matrices of modulation and distortion do not have their maximum values on the same coefficients, which confirm that the FiLM layers exploit the conditioning information.

Finetuning attempts

Inspired by [9], we used SPSA to make the `pedalboard`'s effects differentiable to be able to train the implemented model end-to-end on a spectral loss. Similarly to what is done in [31], the *supervised* training on the parameter loss was gradually changed to *unsupervised* training on a spectral loss. This approach would allow to train the network on *out-of-domain* sounds once the spectral loss has completely replaced the parameter loss, allowing for better generalization.
However, we tested many different ways to finetune the model: gradually introduce the spectral loss while reducing the parameter loss; switch from parameter loss to spectral loss with different hyperparameters; train on the spectral loss while freezing some layers... None of the tested approaches worked for finetuning the model. The network would always end up unlearning what it could do before adding the spectral loss, only returning clean sounds as it appeared to be a local minimum of the spectral loss.
Further tests in the simple case of finding the parameters to match a single audio file suggest that the differentiation technique used for the effects is not suited to our problem. Indeed, some parameters have more impact than others on the output sound of the effects and SPSA is highly sensitive to such issues. Because all parameters are perturbed at the same time to approximate the gradient, the sound can be mostly affected by a main parameter and the computed gradient will not reflect the impact of smaller parameters. This might have been avoided by using a FD algorithm but, as it has been said before, it requires more computational power and would thus slow the training down. We did not have the time to implement that approximation scheme to test this hypothesis.

5.4.3 Perceptual experiment

This internship has been the opportunity for me to implement my first online perceptual experiment. We used the `WebMUSHRA` repository[3] [49] that proposes a framework to quickly implement listening tests through `.yaml` configuration files and a PHP web server. We used it to implement a MUSHRA-like listening test on a personal webserver, a screenshot from the experiment is shown Figure 5.8.

During this experiment, the participants are proposed a reference sound processed with an audio effect from the IDMT-SMT dataset. It is thus an *Out-of-Domain* sound, never seen by the networks and that might be impossible to reproduce.

[2] https://adhooge.github.io/AutoFX/
[3] https://github.com/audiolabs/webMUSHRA

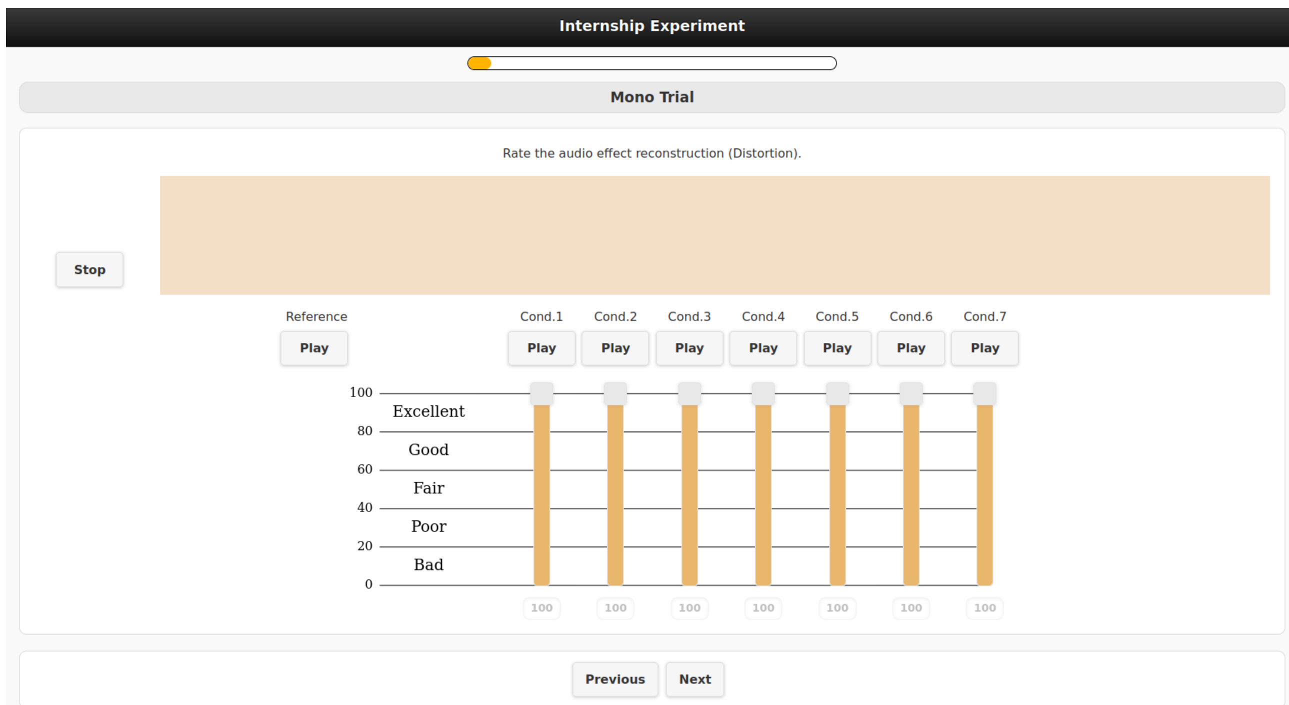


Figure 5.8: Screenshot of the online experiment interface. The user can listen to the reference sound and the proposed reconstructions as many times as necessary and rate them from 0 (Bad) to 100 (Excellent).

The parameters to reproduce that effect are predicted by each model and used to process the clean sound (available from the dataset), the obtained sounds are then proposed to the participants who are expected to rate the quality of the reconstruction from 0 (Bad) to 100 (Excellent). As it is common in perceptual experiments, and mandatory in MUSHRA listening tests, the reference is also included in the sounds to rate. This is to ensure that the task is correctly understood and to provide a higher bound to compare other ratings to. I also added sounds reconstructed with random parameters uniformly sampled in the predefined ranges to use as a lower bound.
The participants were asked to rate 10 sounds for each effect class: Modulation, Delay and Distortion. For each sound, 5 reconstructions were proposed from the following models[4]:

- 163NC: MLP model trained on the 163 classification features with No Conditioning;
- 163C: MLP model trained on the 163 classification features with added Conditioning;
- 211C: MLP model trained on 211 features (the classification features and the added regression features) with Conditioning;
- AutoFX: CNN model with FiLM conditioning but no additional features;
- AutoFX-F: CNN model with FiLM conditioning and added features.

The test pages were randomized as were the reconstruction sounds to ensure fairness during the rating process. 17 persons aged from 20 to 54 years old, most with a background in music either through their work or because they practice an instrument participated in the online experiments. A summary of the ratings is shown Figure 5.9. All proposed models perform better than using random parameters and some perform quite well compared to the reference that is almost always rated at 100. Indeed, most models have a median rating around 60 with the best model being around 65, the lower bound being 30, the median rating of the random samples.

[4]Sounds used in the experiment are available on the accompanying website.

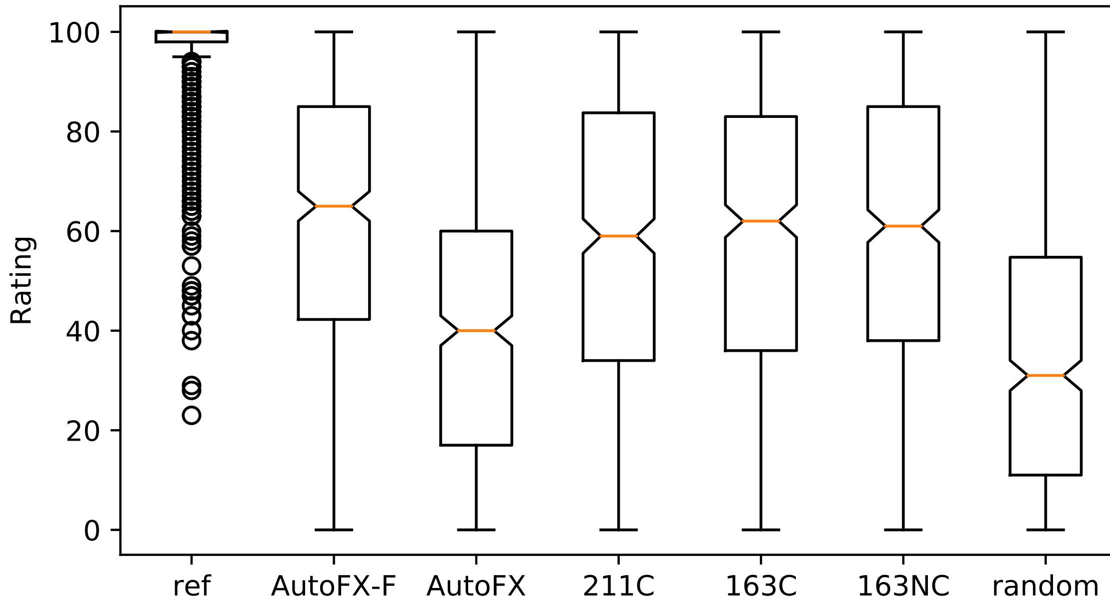


Figure 5.9: Boxplots of the online perceptual experiment. The orange line is the median rating for each category while the box itself represents the first and third quartiles. The whiskers extend the box by 1.5 times the Inter-Quartile Range (the size of the box) on each side and the empty circles are *outliers*, points that are outside the range covered by the whiskers.

Those results are interesting as they show that well-defined features with a light network can perform better than a deep network with much more parameters. However, it also appears that the best network is the biggest, a compromise then have to be find between ease of training, size of the trained model and inference time.

This experiment was the first of its kind I implemented, though efficient and interesting, some issues have been brought to my attention that will be useful for future experiments. First of all, many participants told me that the experiment was very long and actually much longer than my estimation. For future experiments we should reduce the number of samples to rate and probably also limit the number of reconstructions, 5 models to evaluate at once being to much. Several participants also reached out to ask for clarifications on the rating procedure, we should thus work on making the experiment more understandable with maybe a few examples to explain what is expected. Finally and even though it is not something we can change easily, some participants wanted to do the experiment on their phone (with headphones) but the website proposed by webMUSHRA is not responsive and cannot be used on a phone, some sounds being unreachable. Nevertheless, this was an enriching experience that I plan on using for future work to make better experiments.

5.4.4 Power and Size requirements

To further compare the implemented architectures, similarly to what we did for the classification models, we measured the energy consumption required to train the models on an Nvidia GeForce GTX 1080 Ti GPU, the size of the models and the inference time with an Intel i7 CPU. Those results are summarized Table 5.1.

Model	Training time	Energy Consumption	Model size	Number of parameters	Inference time
AutoFX	100 min	0.560 kWh	44.1 MB	3.7M	21.4 ms
MLP-163	19 min	0.0649 kWh	852 kB	58.9k	0.18 ms
MLP-211	12 min	0.0401 kWh	911 kB	63.7k	0.18 ms

Table 5.1: Size and power requirements of the regression models. MLP-163 and MLP-211 refer to the versions of the MLP network with conditioning and either 163 or 211 features.

As one can see and as it could be expected, the AutoFX model is the longer to train and the greedier regarding size and energy. For comparison, it takes approximately 10 Wh to charge a smartphone so the MLP models' trainings require as much energy as charging a smartphone 4 to 6 times while training the AutoFX model is equivalent to charging the same smartphone 56 times. While these consumptions are very low compared to recent neural networks that are trained for weeks on GPU clusters, it is interesting to see how the AutoFX model requires more energy for performance improvements that are noticeable but not outstanding. Besides, it is also slower at inference time and the saved model is heavier than its MLP counterparts. To be fair, the inference of the model itself is not the longer operation when analyzing a new sound: computing its additional features will indeed take approximately 50 ms. Nevertheless, processing a new sound under 0.1 s is still very fast and would not be noticeable by the user, which is reassuring for the objective of implementing the models as a plugin in DAWs.

Chapter 6

Conclusion

In this report, we presented this internship's work that was to develop a tool to assist musicians in their creative process by automatically recognizing audio effects and retrieving their parameters. After a short explanation of the theoretical notions involved, we commented the existing literature on the topic of timbre reproduction and audio effects emulation. We then introduced our work firstly with the architectures we implemented for the recognition and classification of audio effects on guitar sounds. We analyzed the performance of the tested algorithms on different sets and with different configurations before evaluating the energetical needs of our models. Next, we moved on to the task of retrieving effect parameters and shared the experiments we conducted, the difficulties we encountered and commented the results of our models both quantitatively and qualitatively through an online perceptual experiment we designed.

Even though the initial objective of the internship is not complete, we got the opportunity to test several approaches and the obtained results are promising and suggest that the chosen method could work. Guitar effects can be identified and classified by extracting audio features and using a simple classification network. We also successfully realized a compiled version of that network, which is the first to designing a plugin that could actually be used by artists. Our work also sheds lights on the limits of the IDMT-SMT Audio effects dataset and suggests that a classifier performing well on that dataset would not necessarily work on new audio effects.
Furthermore, our pipeline to generate synthetic data demonstrates the potential of Spotify's `pedalboard` library for audio effects related tasks.
We also show that finding the parameters of an effect from an audio spectrogram or a set of audio features is feasible with good accuracy with rather small neural networks. Interestingly, our perceptual experiment suggests that generic audio effects can be used to emulate any other effect of the same type as long as they are correctly configured. The sound matching will not be perfect but could be sufficient to assist artists who want to obtain a specific sound.

Had we gotten more time, we could have tested new differentiation techniques of audio effects to finetune the AutoFX model. It would also be interesting to implement the last effects present in the dataset to see if the observations hold to more and more complex datasets. Another improvement would be to generalize to other instruments than monophonic guitars or to work on melodies instead of single notes to get closer to real use cases. Besides, we considered audio processed by a single effect, which is also far from real-life situations. Future work could focus on generalizing the approach to chain of effects. To the best of our knowledge, this situation has already been studied for effects classification [15] but never for parameters estimation.
Finally, the regression network is compileable so it would be possible to finish the AutoFX plugin and release a first version of it quite easily.

Bibliography

[1] M. A. Martinez Ramirez *et al.*, "Differentiable Signal Processing With Black-Box Audio Effects", in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2021)*, pp. 66–70, Jun. 2021.

[2] M. Senior, "Introduction", in M. Senior (ed.), *Mixing Secrets for the Small Studio*, pp. ix–x, Focal Press, Boston, Jan. 2011. URL https://www.sciencedirect.com/science/article/pii/B9780240815800000223.

[3] L. Köper and M. Holters, "Taming the Red Llama—Modeling a CMOS-based Overdrive Circuit", in *Proceedings of the 23rd International Conference on Digital Audio Effects (DAFx2020)*, 2020.

[4] A. Ramirez *et al.*, "Bistable Digital Audio Effect", in *Proceedings of the 23rd International Conference on Digital Audio Effects (DAFx2020)*, 2020.

[5] M. Van Walstijn, "Numerical Calculation of Modal Spring Reverb Parameters", *Proceedings of the 23rd International Conference on Digital Audio Effects (DAFx2020)*, pp. 38–45, Sep. 2020.

[6] K. J. Werner and R. McClellan, "Moog Ladder Filter Generalizations based on State Variable Filters", in *Proceedings of the 23rd International Conference on Digital Audio Effects (DAFx2020)*, 2020.

[7] J. Chowdhury, "Stable Structures For Nonlinear Biquad Filters", in *Proceedings of the 23rd International Conference on Digital Audio Effects (DAFx2020)*, 2020.

[8] E.-P. Damskägg *et al.*, "Deep Learning for Tube Amplifier Emulation", in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP2019)*, pp. 471–475, May 2019.

[9] M. A. Martínez Ramírez, E. Benetos and J. D. Reiss, "Deep Learning for Black-Box Modeling of Audio Effects", *Applied Sciences*, 10(2), p. 638, Jan. 2020.

[10] J. Imort *et al.*, "Removing Distortion Effects in Music Using Deep Neural Networks", 2022.

[11] E.-P. Damskägg, L. Juvela and V. Välimäki, "Real-Time Modeling of Audio Distortion Circuits with Deep Learning", in *Proceedings of the 16th Sound & Music Computing Conference (SMC 2019)*, pp. 332–339, 2019.

[12] F. Eichas and U. Zölzer, "Gray-Box Modeling of Guitar Amplifiers", *Journal of the Audio Engineering Society*, 66(12), pp. 1006–1015, Dec. 2018.

[13] A. Novak *et al.*, "Chebyshev Model and Synchronized Swept Sine Method in Nonlinear Audio Effect Modeling", in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, 2010.

[14] M. Stein, "Automatic detection of multiple, cascaded audio effects in guitar recordings", in *13th Int. Conference on Digital Audio Effects (DAFx-10)*, 2010.

[15] M. Stein *et al.*, "Automatic Detection of Audio Effects in Guitar and Bass Recordings", in *Audio Engineering Society Convention 128*, Audio Engineering Society, May 2010.

[16] K. Dosenbach, W. Fohl and A. Meisel, "Identification of Individual Guitar Sounds by Support Vector Machines", in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, 2008.

[17] F. Eichas, M. Fink and U. Zölzer, "Feature Design for the Classification of Audio Effect Units by Input/Output measurements", in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, vol. 18, 2015.

[18] M. Schmitt and B. Schuller, "Recognising Guitar Effects - Which Acoustic Features Really Matter?", *Gesellschaft für Informatik, Bonn*, 2017.

[19] S. I. Mimilakis *et al.*, "Deep Neural Networks for Dynamic Range Compression in Mastering Applications", in *Audio Engineering Society Convention 140*, Audio Engineering Society, May 2016.

[20] D. Sheng and G. Fazekas, "Automatic Control of the Dynamic Range Compressor Using a Regression Model and a Reference Sound", in *Proceedings of the 20th International Conference on Digital Audio Effects (DAFx-17)*, 2017.

[21] S. Nercessian, "Neural Parametric Equalizer Matching Using Differentiable Biquads", in *Proceedings of the International Conference on Digital Audio Effects (DAFx)*, 2020.

[22] V. Välimäki and J. Rämö, "Neurally Controlled Graphic Equalizer", *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 27(12), pp. 2140–2149, Dec. 2019.

[23] J. Rämö *et al.*, "Neural Third-Octave Graphic Equalizer", in *Proceedings of the 22nd International Conference on Digital Audio Effects (DAFx-19)*, p. 6, 2019.

[24] M. Comunità, D. Stowell and J. D. Reiss, "Guitar Effects Recognition and Parameter Estimation With Convolutional Neural Networks", *Journal of the Audio Engineering Society*, 69(7/8), pp. 594–604, Jul. 2021. Publisher: Audio Engineering Society.

[25] H. Jürgens, R. Hinrichs and J. Ostermann, "Recognizing Guitar Effects and Their Parameter Settings", in *Proceedings of the 23rd International Conference on Digital Audio Effects (DAFx2020)*, 2020.

[26] M. J. Yee-King, L. Fedden and M. d'Inverno, "Automatic Programming of VST Sound Synthesizers Using Deep Networks and Other Techniques", *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2(2), pp. 150–159, Apr. 2018.

[27] M. Cartwright and B. Pardo, "SynthAssist: an audio synthesizer programmed with vocal imitation", in *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 741–742, ACM, Nov. 2014.

[28] P. Esling *et al.*, "Universal audio synthesizer control with normalizing flows", in *Proceedings of the International Conference on Digital Audio Effects (DAFx 2019)*, Sep. 2019.

[29] C. J. Steinmetz, N. J. Bryan and J. D. Reiss, "Style Transfer of Audio Effects with Differentiable Signal Processing", 2022.

[30] B.-Y. Chen *et al.*, "Automatic DJ Transitions with Differentiable Audio Effects and Generative Adversarial Networks", in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP2022)*, pp. 466–470, May 2022.

[31] N. Masuda and D. Saito, "Synthesizer Sound Matching with Differentiable DSP", in *International Society for Music Information Retrieval 2021 (ISMIR 2021)*, Nov. 2021.

[32] C. J. Steinmetz *et al.*, "Automatic Multitrack Mixing With A Differentiable Mixing Console Of Neural Audio Effects", in *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 71–75, Jun. 2021. ISSN: 2379-190X.

[33] G. Peeters, “A large set of audio features for sound description (similarity and classification) in the CUIDADO project”, Tech. Rep. 1, 2004.

[34] T. H. Park, *Towards automatic musical instrument timbre recognition*, Ph.D. thesis, Jan. 2004.

[35] Y. Tang, “Deep Learning using Linear Support Vector Machines”, in *International Conference on Machine Learning 2013: Challenges in Representation Learning Workshop*, 2013.

[36] L. F. W. Anthony, B. Kanding and R. Selvan, “Carbontracker: Tracking and Predicting the Carbon Footprint of Training Deep Learning Models”, in *ICML Workshop on "Challenges in Deploying and monitoring Machine Learning Systems"*,, 2020.

[37] “pyRAPL/pyRAPL code repository”, URL https://github.com/powerapi-ng/pyRAPL.

[38] L. Breiman, “Random Forests”, *Machine Learning*, 45(1), pp. 5–32, 2001.

[39] B. McFee *et al.*, “librosa/librosa: 0.9.2”, Jun. 2022. URL https://zenodo.org/record/6759664.

[40] “spotify/pedalboard”, Nov. 2021. URL https://github.com/spotify/pedalboard.

[41] “ElectroSmash - Boss DS1 Distortion Analysis”, URL https://www.electrosmash.com/boss-ds1-analysis.

[42] S. Böck and G. Widmer, “Maximum Filter Vibrato Suppression for Onset Detection”, in *Proceedings of the 16th International Conference on Digital Audio Effects (DAFx)*, 2013.

[43] K. He *et al.*, “Deep Residual Learning for Image Recognition”, in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, IEEE, Jun. 2016.

[44] R. Yamamoto, E. Song and J.-M. Kim, “Parallel Wavegan: A Fast Waveform Generation Model Based on Generative Adversarial Networks with Multi-Resolution Spectrogram”, in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP2020)*, pp. 6199–6203, IEEE, May 2020.

[45] C. J. Steinmetz and J. D. Reiss, “auraloss: Audio focused loss functions in PyTorch”, in *Digital Music Research Network One-day Workshop (DMRN+15)*, 2020.

[46] S. Ruder, “An Overview of Multi-Task Learning in Deep Neural Networks”, Jun. 2017. Self-published.

[47] E. Perez *et al.*, “FiLM: Visual Reasoning with a General Conditioning Layer”, *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018.

[48] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization”, in *International Conference on Learning Representations*, arXiv, 2015.

[49] M. Schoeffler *et al.*, “webMUSHRA — A Comprehensive Framework for Web-based Listening Tests”, *Journal of Open Research Software*, 6(1), Feb. 2018.

Appendix A

Audio effects

Effect	Typical use cases	Implementation details
Distortion	Add harmonics to a sound to make it sound richer. Typical of electric guitars in rock and metal music	The input sound is amplified and passed to a processing unit (digital or analog) that saturates beyond a given threshold. The flattening of the waveform increases the quantity of high frequency harmonics, similarly to what happens to a *Sine* being gradually transformed into a *Square signal*
Overdrive	Like Distortion but softer, typical of electric guitars in Blues and Rock music	Like Distortion but with a smoother saturation unit and less preamplification
Fuzz	Like Distortion but harsher, typical of electric guitars in 60's rock. This effect was for instance widely used by Jimi HENDRIX	Like Distortion but with a more brutal saturation, usually obtained by a transistor in analog effects
Compression	An effect to reduce the dynamics of a recording. Dynamics refer to the changes in loudness that can occur in a piece of music, for instance a violin going from *piano* to *forte* will cause great dynamics of the recorded loudness. It is also typical to use a compressor on drum recordings to reduce the attack's intensity of drum hits	A compressor has two processing ranges: below a chosen loudness threshold, the input signal is left as it is or even amplified; above that threshold, the signal's loudness is reduced. The user can usually choose how much the signal is *compressed* on the loud end with a *ratio* control. Besides, to guarantee smoothness of the processed sound, the compressors are usually dynamic with an *attack* and a *release* control that respectively determine how much time the processor takes before beginning the compression when the threshold is reached or how long the compression lasts after going below the threshold
Reverb	Add resonance to a recording to simulate the reflections of the sound on the walls, floor and ceiling. This usually adds *presence* to a recording, making it sound less artificial because it feels like the singer/musician is in an actual room.	It can be obtained by convolution of the input signal with the impulse response of a room. It can also be done by simulating reflections or using mechanical components such as a spring

Effect	**Typical use cases**	**Implementation details**
Delay	Add echoes to a sound to simulate another physical space, to add rhythmic repetitions or increase the impact of a word or a melody	The original signal is duplicated a chosen number of times at a controllable volume (called *feedback*) which are played after a chosen waiting time (the *delay*).
Chorus	Roughly imitates the sound that could be obtained from several musicians playing the same part at once with slight timing variations, making the input sound seems *wider*. It is naturally occuring in instruments with *sympathetic strings*	The input signal is duplicated, slightly delayed (by a few dozen milliseconds) and pitch-shifted by a Low-Frequency Oscillator (LFO) before adding it back to the original signal. This causes beating between close frequencies, producing the effect heard
Flanger	Similar to a filter sweeping through frequencies, it can be used to add *movement* to a sound	The original signal is copied and the copy is delayed with a delay controlled by a LFO before being mixed back with the input signal
Vibrato	An effect that can actually be done directly on several instruments such as the *singing voice*, *violins* and *guitars* which consists of making the pitch vary slightly around a center note, for instance by bending the played string. It can be used to add expressivity to a note	Exactly like Flanger but the original signal is discarded, only keeping the delayed version
Tremolo	The sound's volume is periodically reduced, adding a rhythmic component sometimes compared to an "underwater" sound	This effect can be obtained either by completely turning off the output signal at regular intervals or using Amplitude Modulation (AM) where the input sound is considered as the *carrier signal* and an LFO produces the *message signal* that will command the amplitude variations
Phaser	Similar to Flanger	The input signal is copied and processed by several all-pass filters *i.e.* filters that do not change the magnitude but affect the phase. Adding this modified signal to the original input will create notches in the frequency spectrum due to phase cancellation
Equalization	This effect can be used by sound engineers to ensure that different instrument tracks do not overlap too much in the frequency domain so that they can all be heard clearly. Equalization can also be used to increase or decrease the volume of low, medium and high frequencies to affect the timbre of an input sound from sharp high-frequency tones to low cavernous ones	The signal is processed by a filterbank that will modify the volume of each frequency band according to the selected filtering parameters: gain, resonance, cut-off frequency or center frequency...

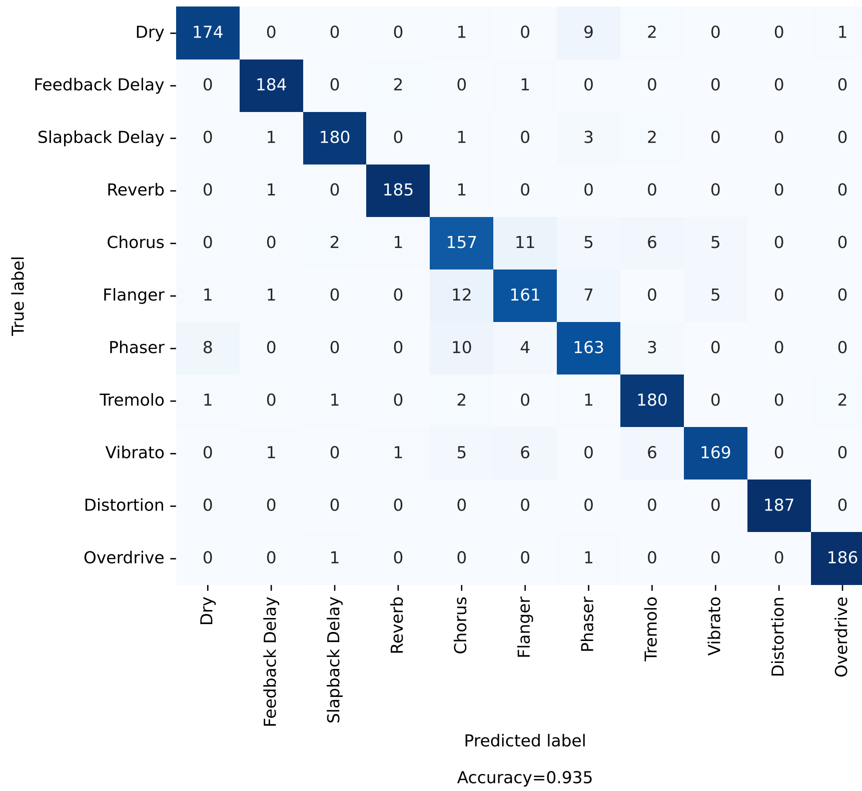
Table A.1: Summary table of audio effects, their use cases and the way they work.

Appendix B

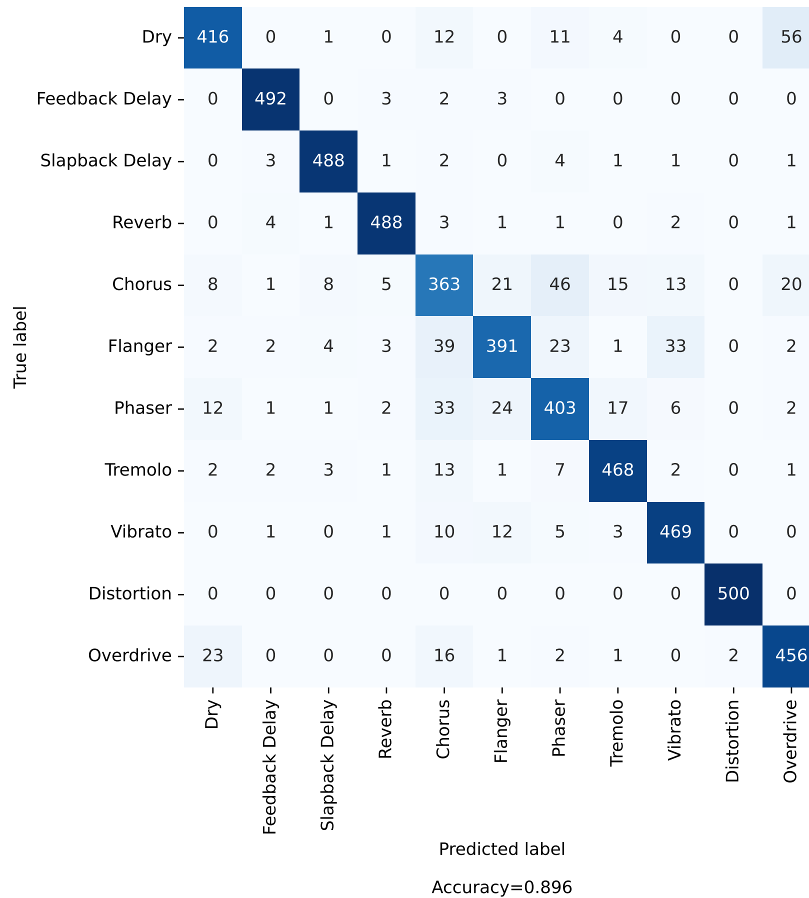
Classification

Feature	Applied functionnals
Spectral Centroid	min, max, mean, variance, skewness, kurtosis
Spectral Spread	min, max, mean, variance, skewness, kurtosis
Spectral Skewness	min, max, mean, variance, skewness, kurtosis
Spectral Kurtosis	min, max, mean, variance, skewness, kurtosis
Spectral Slope	min, max, mean, variance, skewness, kurtosis
Spectral Roll-off	min, max, mean, variance, skewness, kurtosis
Spectral Flux	min, max, mean, variance, skewness, kurtosis
Spectral Flatness	min, max, mean, variance, skewness, kurtosis
δSpectral Centroid	min, max, mean, variance, skewness, kurtosis
δSpectral Spread	min, max, mean, variance, skewness, kurtosis
δSpectral Skewness	min, max, mean, variance, skewness, kurtosis
δSpectral Kurtosis	min, max, mean, variance, skewness, kurtosis
δSpectral Slope	min, max, mean, variance, skewness, kurtosis
δSpectral Roll-off	min, max, mean, variance, skewness, kurtosis
δSpectral Flux	max, mean, variance, skewness, kurtosis
δSpectral Flatness	min, max, mean, variance, skewness, kurtosis
10 first MFCCs	max, mean
Spectral Centroid / pitch	min, max, mean, variance, skewness, kurtosis
Spectral Spread / pitch	min, max, mean, variance, skewness, kurtosis
Spectral Skewness / pitch	min, max, mean, variance, skewness, kurtosis
Spectral Kurtosis / pitch	min, max, mean, variance, skewness, kurtosis
Spectral Slope / pitch	min, max, mean, variance, skewness, kurtosis
Spectral Roll-off / pitch	min, max, mean, variance, skewness, kurtosis
Spectral Flux / pitch	min, max, mean, variance, skewness, kurtosis
Spectral Flatness / pitch	min, max, mean, variance, skewness, kurtosis

Table B.1: Classification features. The minimum of δSpectral flux is discarded because it is always zero.

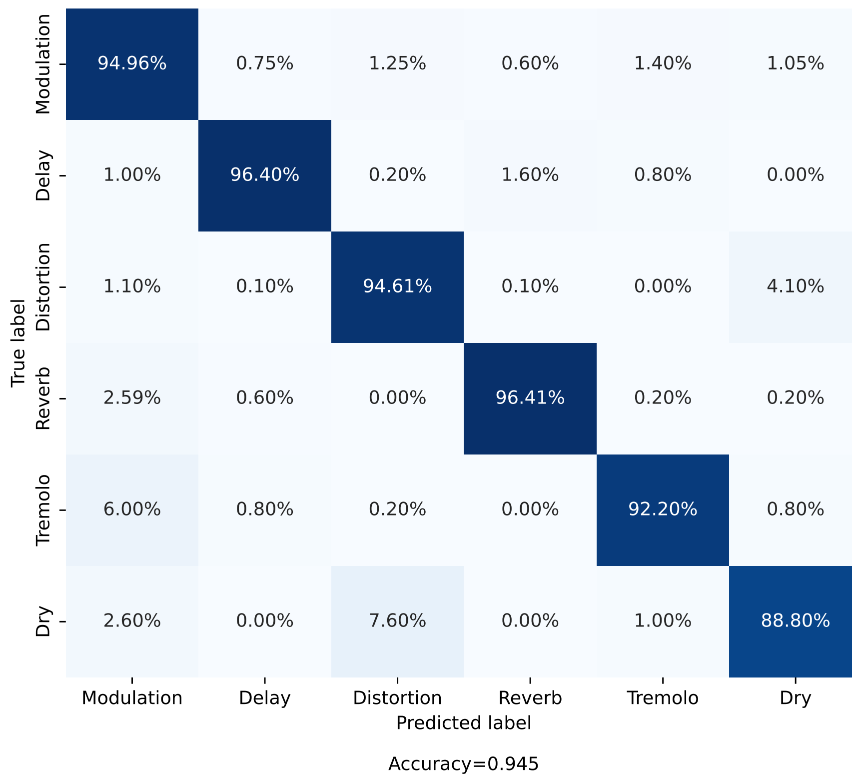


(a) Guitar Monophonic

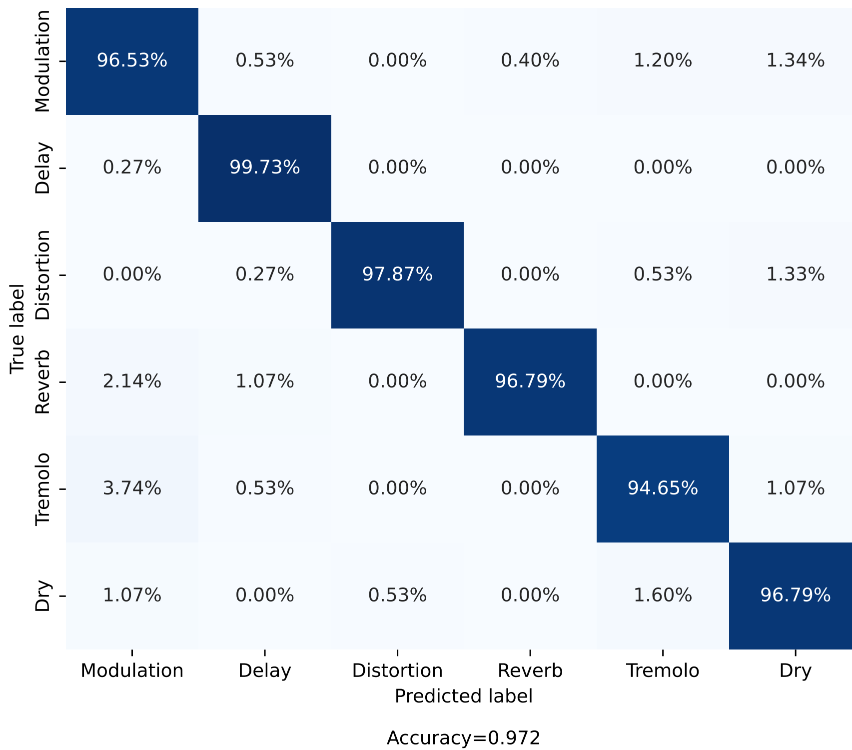


(b) Full dataset

Figure B.1: Confusion matrices of the classification experiments on 11 effects.



(a) Full dataset



(b) Monophonic guitar

Figure B.2: Confusion matrices on the test set with aggregated classes. Since the aggregated dataset is no longer balanced, rows are normalized by the number of actual samples for each class.

Appendix C

Regression

Feature	Applied functionals
Pitch	2 FFT max bins, 2 FFT max bins values
Unwrapped phase of the maximum frequency bin	2 FFT max bins, 2 FFT max bins values
RMS energy	2 FFT max bins, 2 FFT max bins values, variance, skewness
δPitch	2 FFT max bins, 2 FFT max bins values
δUnwrapped phase of the maximum frequency bin	2 FFT max bins, 2 FFT max bins values
δRMS energy	2 FFT max bins, 2 FFT max bins values, variance, skewness
5 Onsets	timestamps, activations
10 MFCCs	mean

Table C.1: Regression features.

	Total	Modulation					Delay			Distortion						
Model	MSE Loss	`rate`	`depth`	`centre-delay`	`feedback`	`mix`	`delay`	`feedback`	`mix`	`drive-db`	`cutoff-hi`	`gain-hi`	`Q-hi`	`cutoff-lo`	`gain-lo`	`Q-lo`
AutoFX	8.42×10^{-3}	0.012	0.032	0.036	0.031	0.025	**0.011**	0.050	0.024	0.063	0.054	0.033	0.079	0.070	0.030	0.053
AutoFX-feat	$\mathbf{6.97 \times 10^{-3}}$	**0.0065**	0.026	0.037	0.028	**0.022**	0.014	**0.035**	0.019	0.019	**0.044**	**0.014**	0.074	0.052	**0.024**	**0.051**
163NC	0.0103	0.025	0.040	0.050	0.050	0.036	0.033	0.046	**0.010**	0.018	0.090	0.031	**0.066**	0.045	0.030	0.068
163C	0.0100	0.024	0.049	0.046	0.049	0.027	0.033	0.039	0.012	**0.0092**	0.080	0.047	0.072	**0.036**	0.032	0.064
211C	9.83×10^{-3}	0.013	**0.021**	**0.030**	**0.015**	0.032	0.021	0.063	0.020	0.019	0.098	0.041	0.11	0.054	0.030	0.081

Table C.2: Results of the regression networks on the test set. Apart from the MSE Loss, all values are absolute distances between the predicted values and the true labels. For all measures, lower is better. *163NC* refers to the MLP network with the 163 classification features and No Conditioning, *163C* is the same network but with added conditioning and *211C* is the MLP trained on all the classification features plus the additional regression features (for a total of 211) with added conditioning.

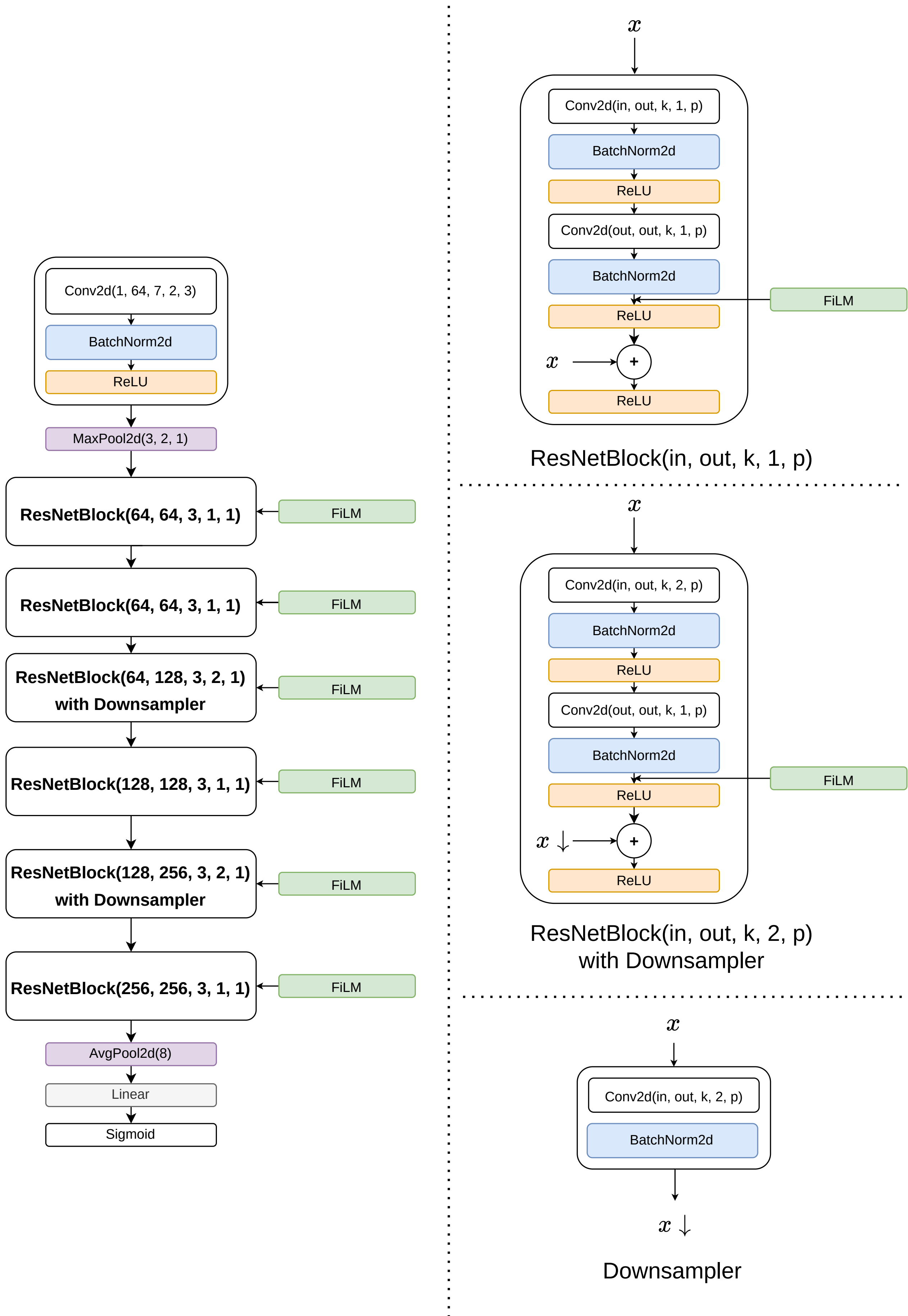


Figure C.1: Architecture of the AutoFx Model (left) and building blocks of said model (right). The parameters in `Conv2d` are, from left to right: number of input channels, number of output channels, kernel size, stride and padding. Dilation is always set to 1.

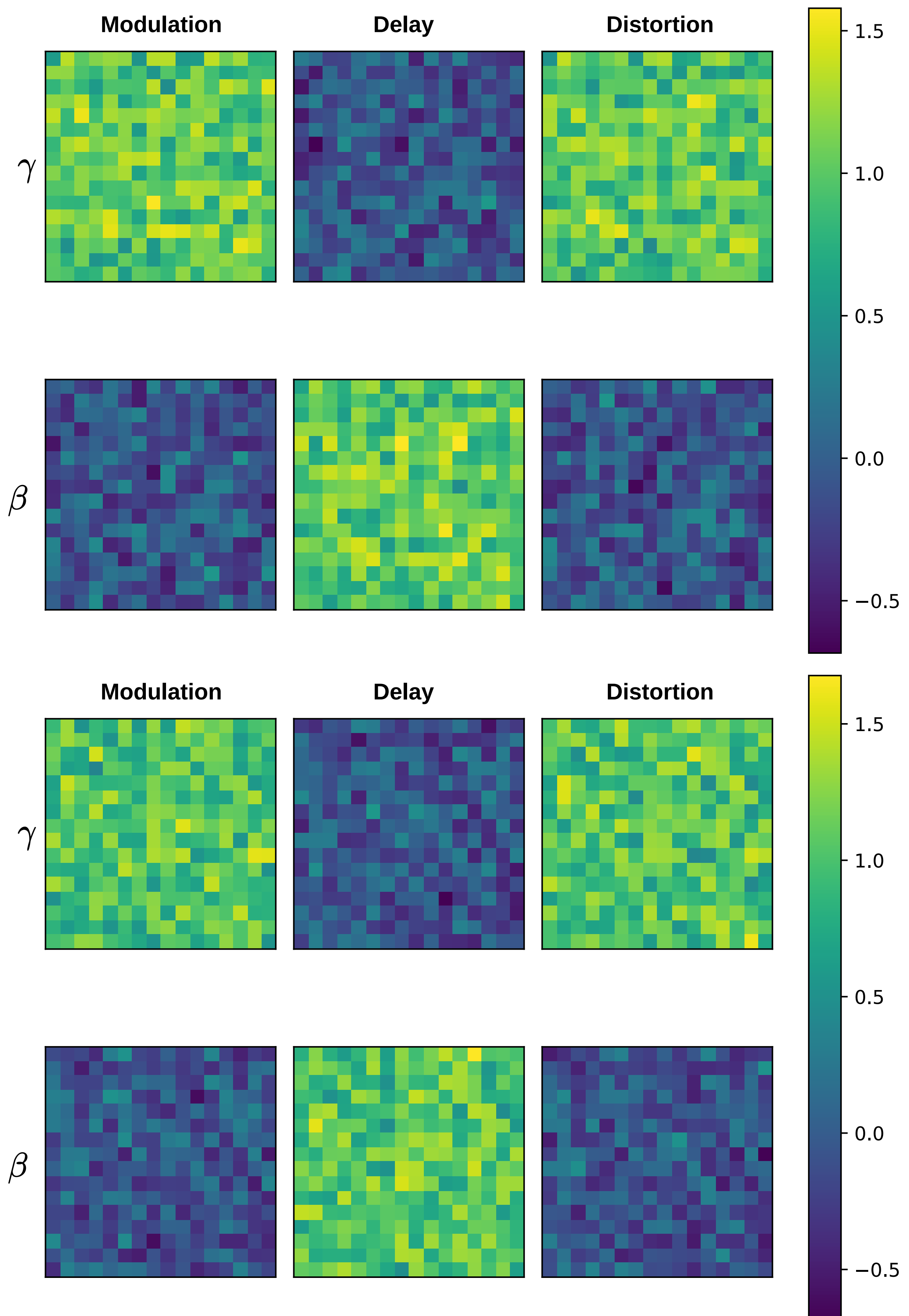


Figure C.2: Output of the last FiLM layers depending on conditioning.