**UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA**

**FACULTATEA DE MATEMATICǍ ŞI INFORMATICǍ**

**SPECIALIZAREA INGINGERIE SOFTWARE**

**LUCRARE DE DISERTAŢIE**

**Dezvoltarea unui algoritm îmbunătățit pentru detecția și atenuarea distorsiunilor de sunet din înregistrări de pe discuri de vinil**

**Conducător ştiinţific**

Lect. Univ. Dr. STERCA ADRIAN

**Absolvent**

DRIMBA ALEXANDRU

**2020**

**BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA**

**FACULTY OF MATHEMATICS AND COMPUTER SCIENCE**

**SPECIALIZATION SOFTWARE ENGINEERING**

**DISSERTATION THESIS**

**Developing an improved algorithm for detection and attenuation of sound distortion in vinyl recordings**

**Supervisor**

Lect. Univ. Dr. STERCA ADRIAN

**Author**

DRIMBA ALEXANDRU

**2020**

**Abstract**

The aim of this thesis is to significantly improve the efficiency of one of our already existing projects by using faster and improved algorithms or approaches. The previous work accomplishes the detection and attenuation of sound artifacts by using signal filtering, neural networks (for detecting noise and distortion in an audio signal) and linear prediction (for reconstructing the distorted signal sequences).

…

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

Contents

1. Introduction. Problem statement and motivation …………………………………………… 2
   1. Introduction. Mechanical analog storage and its specific sound distortions ……….. 2
   2. Purpose of this work ….. ……………………………………………………………. 4
2. Previous work .……………………………………………………………………………… 5
   1. A Java application for basic signal processing ……………………………………… 5
   2. Implementation details and proposed improvements ………………………………
3. A faster equivalent to the FIR filtering: Overlap-Add Short-Time Fourier Transform ………
   1. FIR filters ……………………………………………………………………………
   2. The Discrete and Fast Fourier Transforms ………………………………………….
   3. Linear filtering in the frequency domain: OLA STFT ………………………………
4. A message-passing Java-Python IPC protocol over stdin/stdout ……………………………
   1. The DLX STX ETX encoding ……………………………………………………….
   2. Message format ………………………………………………………………………
   3. List of supported messages ……………………………………………………………
5. Application updates. Experimental tests and results ………………………………………
   1. FFT filtering using OLA STFT ………………………………………………………
      1. Improvements to the existing FFT implementation …………………………
      2. Unexpected discrepancies and the proposed fix ……………………………
      3. Performance results of OLA STFT filtering vs FIR filtering ………………
   2. The new IPC protocol and Python classifier ……………………………………………
      1. Message processing, throughput and robustness ……………………………
      2. Generating the training data …………………………………………………
      3. Training the classifier. Accuracy and performance results …………………
   3. Other implementation improvements …………………………………………………
   4. Bug fixes ………………………………………………………………………………

6. Conclusions and Future Work ………………………………………………………………

Bibliography ...……………………………………………………………………………………

Chapter 1

Introduction. Problem statement and motivation

1.1. Introduction

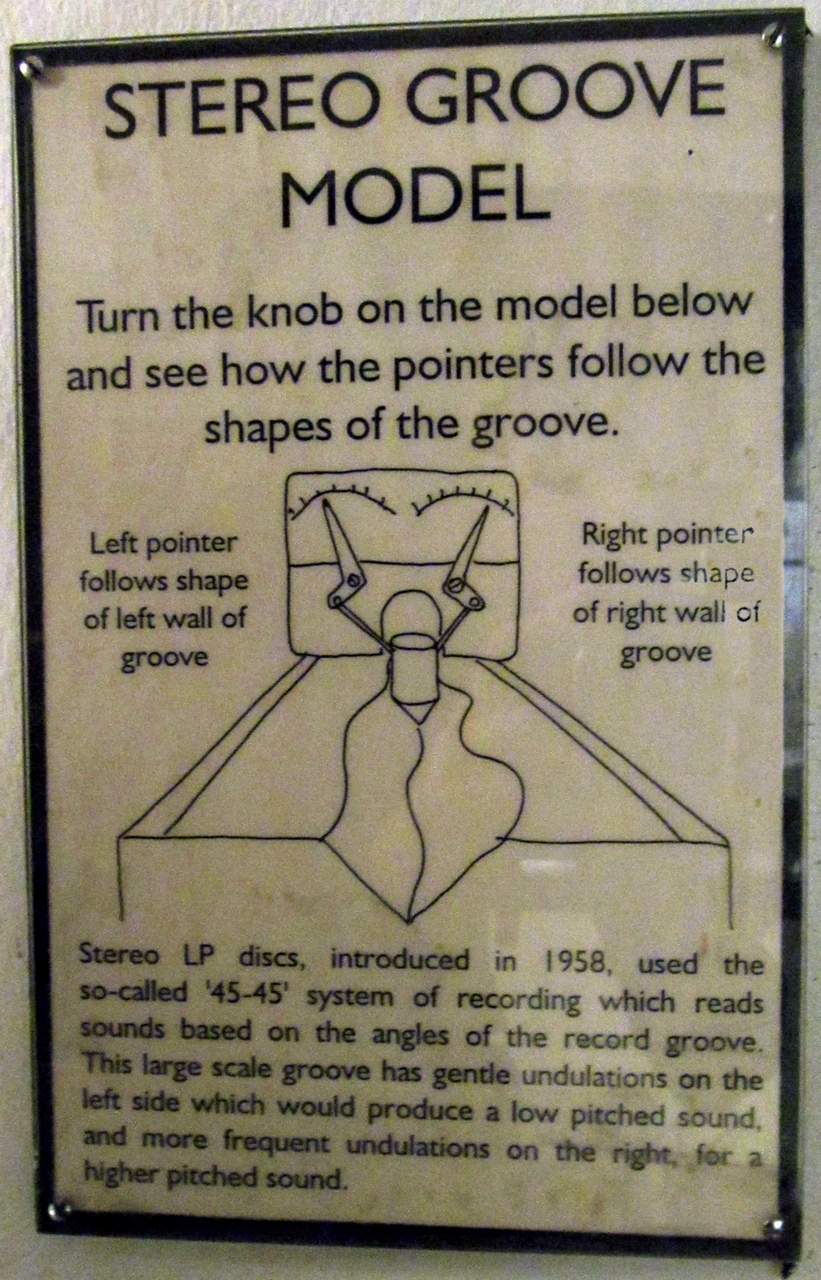
Recorded music available for the wide public has only been around us for less than 150 years. Starting with the invention of the Edison Phonograph, and the beginning of its commercialization in 1888[[1]](#footnote-2), the way we’ve been listening to music went under lots of changes. Nowadays, more than 95% of recorded music revenues come from digital formats, such as CDs, digital downloads, paid subscriptions or ad-supported on-demand streaming (1), but this has not always been the case.

For almost 100 years, the most popular and accessible audio format was the mechanical analog format, and this was used with a wide variety of physical media formats. The first commercially available format was the Edison cylinder, which was developed in the late 19th century. At the same time, using a similar principle but a different mechanical approach, the gramophone disc was developed[[2]](#footnote-3). This disc, also called “shellac” record because of the material it was made of, later evolved into the modern vinyl disc, which came up to be the standard audio format until the introduction of Compact Cassettes and CDs in the mid 1980’s.

In mechanical analog formats, sound is stored as a modulated groove, which is later picked up and converted to either sound or electrical signals by a stylus, as it can be seen in Figure 1.1. The groove and the reproduction mechanism have greatly evolved over time. The early age of discs consisted of low fidelity acoustical recordings, where the groove was cut directly by air sound pressure, which was collected and focused by a horn piped to a diaphragm, which vibrated a cutting stylus. The reproducers were also acoustical, and were typically made out of a play stylus, attached to a soundbox (see fig. 1.2), with the latter amplifying and transmitting the mechanical oscillations picked up by the stylus, and feeding them to a horn, which further amplified the sound. The pressure the stylus put on the record’s groove was immense: depending on the soundbox and tonarm, the steel needle could push against the groove walls with more than 100 grams of force, distributed over an area of a few hundred square microns. The immense pressure will cause irreversible heavy wear, but this problem was accounted for by using soft steel needles and hard and abrasive disc materials, which means that the stylus will take most of the wear damage instead of the record. Because of this, needles have to be replaced after each play[[3]](#footnote-4), but this only limited the groove wear: a sufficient number of plays will damage the groove and cause significant noise and a decrease in the sound quality.

Even though electrically recorded discs became available as soon as the mid 1920’s, it wasn’t until the late 40’s when electrical transducers, called cartridges, were introduced to the home-use market. Instead of amplifying sound, their role was to convert the mechanical oscillations of the stylus into electrical signals, which were later electrically amplified and fed to loudspeakers. This conversion was made by either using piezoelectric crystals, such as Rochelle salts, or using induction coils and permanents magnets[[4]](#footnote-5). These cartridges allowed for using way smaller tracking forces of 4-12 grams, with modern moving-magnet (Fig. 1.3) or moving-coil cartridges usually requiring tracking forces in the 1.5-3 grams range. The electrical reproducers were introduced at roughly the same period as the vinyl records – an improved version of the shellac record. These were made of PVC, which had more fine particles and reduced the surface noise, and this allowed using smaller grooves (called microgrooves), slower speeds (33-45 RPM instead of the 78 RPM standard of shellacs), better frequency response and dynamic range, and longer play times (up to 20-30 minutes per side, instead of the maximum 5 minutes per side allowed by 12-inch shellacs). Because PVC is softer than the coarse and abrasive shellac, their playing styli had sapphire or diamond tips – much harder and smaller than the steel previously used4.

Fig. 1.1: A record groove model depicting stereo modulation. London Museum of Science





VM540ML, TBD

Fig. 1.3: A modern diamond-tipped moving magnet cartridge, tracking at less than 2g

Fig. 1.2: An HMV no. 5A soundbox,

dating from 1931-1932, tracking at 130g

The surface of the discs is prone to scratches and accumulation of dust particles, mainly caused by mishandling or improper storage. But even at these lower tracking forces, groove damage was still present. Although very durable, diamond and sapphire wear out after a few tens or hundreds hours of playing. With a standard 25μm spherical diamond tip, the limit load for plastic deformation of a vinyl groove is 0.64 grams[[5]](#footnote-6). As tracking forces must be over that value to provide an accurate tracking of the groove, even a perfectly shaped diamond tip will cause groove damage proportional with its tracking force. At a vertical tracking force of four grams, the stylus presses on its two resting points with a pressure of nearly 28kgf/mm2 4. Using a worn out stylus will quickly lead to severe groove wear, which results in perceptible and annoying noise when playing the record, and even rendering the record unlistenable.

[ Pictures of grooves and groove damage ]

1.2 Purpose of this work

There are lots of people which collect records, even in today’s age. New records are still produced, with LP sales steadily growing in the last decade (1), but records from their golden age are still around. Some of them are very valuable to their owners because the music on them cannot be found on other media, or just because they come from the era they were originally released in. Here is where groove wear is a real nuisance. Your valuable recording is full of clicks, crackles and noise, but it is something really rare or expensive and you cannot just go and buy another one. A quite costly trick that might work is using a new modern stylus and cartridge. There are some advanced stylus profiles, collectively called “micro-line”, which offer less distortion, less wear, better sound quality and a longer life. These shapes are different from the standard spherical or elliptical tips, and are closer to the shape of a cutting stylus. The vertical contact area is significantly longer than the lateral contact area, and this enables them to track portions of the groove other styli cannot or could not reach. Because of this, if the groove wear is not very deep, an advanced stylus profile may track portions the groove that yield a much better sound quality.

However, most of the time, not even a new stylus will produce a better sound. In this case, the only option left is to process the audio signal outputted by the cartridge, and remove the noise using signal processing. Nowadays, it is very easy to process audio files using software installed on personal computers. One of our previous works already tackled the issue of groove damage in vinyl record (2). It also provides more insight into the evolution of the mechanical analog format, how sound is recorded and reproduced, how different types of transducers work, and how different types of distortion occurs.

The purpose of this work is to increase the time-efficiency and accuracy of the distortion detection and attenuation algorithm described and implemented in (2). The described project accomplishes the detection and attenuation of sound artifacts by using signal filtering, neural networks (for detecting noise and distortion in an audio signal) and linear prediction (for reconstructing the distorted signal sequences). The signal filtering and the Artificial Intelligence components have room for improvement in the efficiency and accuracy domain, but the application as a whole can also be made more user friendly and usable, by adding a few new features.

Chapter 2

Previous work

2.1. A Java application for basic signal processing

In one of our previous works (2), we have described and implemented an algorithm that is capable of detecting regions of damaged audio signal, and then reconstructs those regions using linear prediction to extrapolate from the adjacent signal. But, in order to implement these high-level functionalities, the application needed to also support basic signal processing effects, loading and saving audio files, a basic user interface and various data structures needed by some of the functionalities.

Because the project also had a self-didactic side along the detection and repair research, the whole project was implemented from scratch, using no libraries other than the JavaFX GUI library.

The implementation consists of six modules:

* AudioDataSource: this offers an interface for retrieving and storing audio data, either in memory or in files. The module contains some implementations of this interface that offers support for reading and writing audio data to and from .WAV and .AU uncompressed file formats. One of the in-memory Audio Data Sources is a caching mechanism, consisting of several fixed-size cache pages, that is designed to reduce the number of read/writes to an underlying File AudioDataSource. There’s also an in-memory ADS that is stand-alone; it only uses a chunk of contiguous memory to store audio data, and does not have an underlying permanent storage to read/write from. It is only used as an intermediary ADS, when it is not desired that its contents are to be written to a permanent storage. Perhaps the most complex implementation in this module is that of the Versioned AudioDataSource. It offers an optimized sequential undo-redo mechanism: when creating a new entry in the sequence, i.e. to apply a signal processing effect, it stores in new temporary files only the modified data; data that was not changed from the previous version is not unnecessarily copied. To be able to retrieve the correct data, each ADS version stores a project to file mapping to know where each audio sequence should be retrieved from.
* SignalProcessing: this offers low-level implementations for various signal processing techniques. These implementations are completely uncoupled from the project-specific data structures that store audio signal data; they only use primitive types as inputs or outputs, so that they could be reused in any other project as they are. The implemented effects are: casual and non-casual FIR filters, IIR filters, Fast Fourier Transforms, a linear function interpolator, linear prediction and an implementation of the Burg method of computing the linear prediction coefficients, and some standard windowing functions.
* Effects: this is a package that offers audio signal effects coupled to the project-specific data structures responsible with fetching and saving audio data. These are mostly higher-level implementations that use the functionalities of the SignalProcessing package. Since it may be unreasonable to load all the sound file into memory, and then directly used the signal processing functions, the functions in this package also deal with processing the data in blocks of some reasonable size, which can be easily stored in memory. Besides wrappers over the SignalProcessing functionalities, there are two important algorithms in this package: the one which reconstructs signal using bilateral linear prediction, the one which detects signal regions that contain distortion, and the one which repairs all the damaged audio regions using frequency bands separation and signal reconstruction.
* GUI: its role is to display the audio waveform, menus and other widgets to the user, and to handle the user actions such as requesting to apply a certain processing effect on a specific signal region.
* ProjectManager: this acts like a Controller layer of the typical MVC architecture. It is used to connect actions triggered by the GUI to their implementation and to store project-level variables and constants, such as the current AudioDataSource and default cache sizes.
* Utils: a package that holds together various data types and structures, such as an OrderedNonOverlappingIntervalSet for efficiently storing the distorted intervals, custom exceptions and others.

The process of detecting and reducing distortion consists of two steps:

* An intelligent agent is used to classify each sample into damaged or undamaged. It has been implemented in python using a densely-connected neural network with 129 inputs and two hidden layers. The inputs are used to classify the center sample (the 65th sample). Using a moving-window approach, each sample of the signal is classified using this neural network. The results are then stored as a list of intervals marking the damaged signal portions, called markings.
* Each marking is reconstructed using linear prediction and, in most case, frequency band splitting. Just using linear prediction for reconstruction caused even worse distortion on the majority of markings. This has been greatly improved by splitting the signal into a few frequency bands, and using linear prediction on each of the resulted signals, and then summing them together to generate the final result. This approach proved to be more effective in reducing noise artifacts, but not in all situations. Signal distortions which presented large amplitude spikes were not properly repaired because the spike caused ripples after the frequency filtering step. Those ripples negatively affected the reconstruction result because extrapolating the ripples introduces a new spike in the resulting signal. This case has been handled by computing whether the marking contains a spike, and reconstructing it without frequency band splitting, which has the desired effect of removing the amplitude burst.

2.2. Implementation details and proposed improvements

Signal filtering, also known as signal equalization, is achieved through FIR (Finite Impulse Response) filters. Although easy to implement, they have the disadvantage of requiring significant computation power, because applying a filter has a complexity of θ( N \* M), where N is the number of filtered samples, and M is the number of filter coefficients. A more speed-efficient method, equivalent to the FIR filtering, is desired. In this paper, we present a method that achieves the same results as the FIR filter, but with a better time complexity, and we compare its results with those of the current implementation.

The other component of the project that can be improved is the audio sample classifier. The main application is implemented in Java, but the AI component was implemented in Python due to its better and wider support of Machine Learning frameworks. The communication between the main Java program and the Python script is done through OS pipes stdin and stdout using some rudimentary form of message transmission protocol, where the Java application acts as a client, and the Python script acts as a server. This protocol is poorly designed and very hard to expand and debug. It only supports sending audio data to Python and reading the output of the classifier. Error reporting and debugging are almost inexistent and are handled exclusively by stopping the Python script. This whole AI component can be improved by completely rewriting the Python script and the Java module that communicates with the script. We propose a new communication protocol, using the same stdin/stdout pipes, but with more robustness and expandability in mind. It must be noted that the current implementation takes an unreasonable amount of time to classify signals. Classifying one hour of stereo audio at a 96kHz sampling rate can take even more than 6 hours!

The AI classifier also has room for improvement. The current Neural Network configuration has 129 inputs and one output, so that it is ran once for each sample being classified. This may turn out to be inefficient, as configurations with a larger number of outputs could achieve a greater number of predictions/second.

Another missing feature, with a big negative impact on the application’s usability, is that the AI output (a “damaged” probability for each signal sample) is immediately converted to markings using a threshold. This means that, if the user wants to generate the markings again, for the same signal, but with a different threshold, they will have to repeat the whole classification process (which takes a lot of time) on the same input signal as previous. The proposed improvement is to divide the damage detection process into two steps: generating the probabilities using AI and storing them in the main app, and then generating the markings based on the stored probabilities and a given threshold.

Chapter 3

A faster equivalent to the FIR filtering: Overlap-Add Short-Time Fourier Transform

3.1. FIR filters

A filter is generally considered anything that can modify the sound in some way. The mouth cavity, for example, can be considered a filter, as changing its shape modifies the acoustic characteristics of the vocal tract, resulting in a different sound. Electronic circuitry such as those found as “tone control” in audio amplifiers, loudspeaker cabinets, and even the room you are listening music in, they are all filters, even though some of them are unwanted (3). Filters’ applications include amplification, attenuation or rejection of certain frequency ranges of the input signal. They can be used in communication systems to reduce noise, isolate certain frequency bands (in radio transmissions, for example), and for signal modulation and demodulation[[6]](#footnote-7).

A digital filter is just a filter that operates on digital signals. It does so by performing mathematical operations on an input signal, which is represented as a sequence of numbers, and outputting a new sequence of numbers – the output signal. Altering an audio signal’s frequency response can be achieved by using a casual, linear, time-invariant digital filter. Linearity means that the output due to a sum of input signals equals the sum of outputs due to each signal alone, and time-invariance means that the filter does not change over time (3). The general equation of such a filter is:

*Eq. 3.1* (12)

where:

• is the input signal,

• is the output signal,

• are the filter orders and the numbers of taps for the feedforward and feedback components of the filter. A “tap” is simply a coefficient-delay pair (4); an th order filter needs N previous input samples and has terms on the right-hand side,

• is the set of the feedforward (previous input samples) coefficients; is the weight associated to the th input sample.

• is the set of the feedback (previous output samples) coefficients; is the weight associated to the th output sample.

This type of filter is called Infinite Impulse Response (IIR) filter, with the “infinite” meaning that, due to its feedback, it can continue to output non-zero values even if the input samples are 0. By removing the feedback component, the filter equation becomes:

Eq. 3.2 (12)

This is a Finite Impulse Response (FIR) filter. FIR filters are inherently finite (the output will eventually become 0 after the inputs become 0) because they lack feedback (5). In a FIR filter, its coefficients are also its impulse response, i.e. the filter’s output when presented to an input signal consisting of a single 1 value followed by zeros. Applying a filter to a signal will alter each frequency’s magnitude and phase according to the filter’s impulse response. The problem of calculating the filter coefficients to achieve a desired frequency response is called “filter design”, and it is a well documented subject in the literature (4) (6) (7).

Although it is easy to design a filter knowing the desired response, it is simple to implement and has desirable numerical properties, the main disadvantage of FIR filters is that they require a big computational effort. A filter of length M, applied to a signal of length N, requires N\*M multiply-additions, so the **time cost** of the FIR filter is **θ(N\*M).**

Figure 2.1 gives an example as how an FIR filter works along the input data (x is the array of input samples, y is the output array):

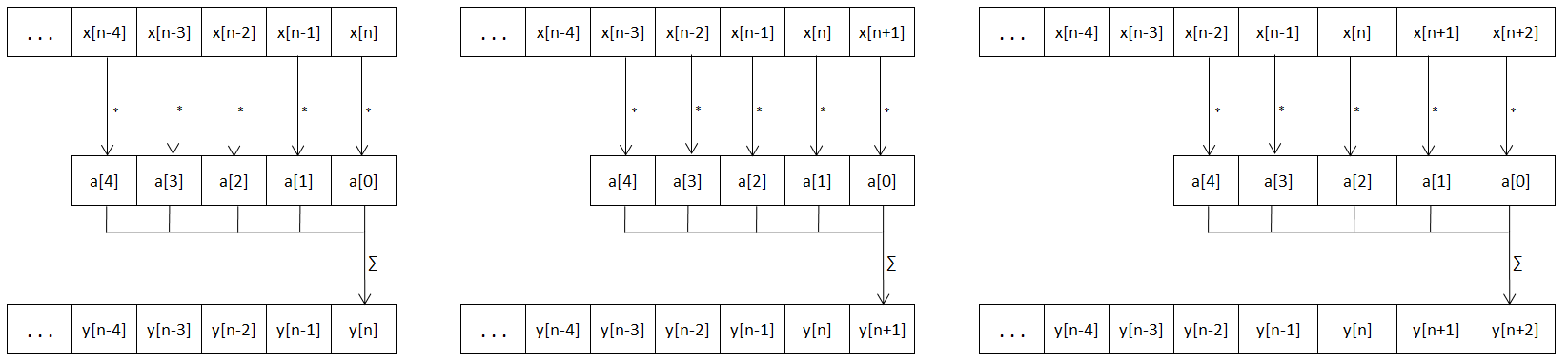


Fig. 3.1: Applying a digital FIR filter

3.2. The Discrete and Fast Fourier Transforms

TBD

An important notion in the FFT is the “twiddle factor” W, given by formula (2.8). It is also important to know that FFT only works for input sizes that are in the form of **N = 2k**, and that the IFFT can be expressed as a FFT if some **θ( N )** pre- and post-processing is done.

The recursive definition of the FFT algorithm separates the even and odd terms and, because of that, the iterative version of it requires to reorder the inputs in reverse-binary. As an example, for N = 8, the inputs need to be rearranged from [ 0, 1, 2, 3, 4, 5, 6, 7 ] (which in binary is [ 000, 001, 010, 011, 100, 101, 110, 111 ]) to [ 0, 4, 2, 6, 1, 5, 3, 7 ] (which in binary is [ 000, 100, 010, 110, 001, 101, 011, 111 ]). Note how the binary representation of each index was reversed.

3.3. Linear filtering in the frequency domain: OLA STFT

Because FIR filtering is computationally intensive, and significant filter lengths are needed to achieve the desired frequency response, a more speed-efficient method, equivalent to the FIR filtering, is desired. Based on the Convolution Theorem (3) (7), which demonstrates that the convolution involved in applying a FIR filter is equivalent to linear filtering in the frequency domain, the method presented in the following sections makes use of the Fourier Transforms to achieve the same results as the FIR, but in a significantly shorter time. This method is also known as **Overlap-Add (OLA) STFT (Short Time Fourier Transform) Processing.** For large enough filters (filter length >~213), the OLA STFT method proves to be hundreds of times faster than its convolution-based counterpart.

This method is already known in the literature. In one of his books, Julius Orion Smith III dedicates a whole chapter to discuss the theory, the importance and the results of the OLA STFT algorithm (8). Julius O. Smith III, a Professor of Music and Electrical Engineering at Stanford University, is a specialist in the *Applications of Fourier Analysis in Digital Audio Signal Processing* and has numerous publications in the field. Many of his results are related to modeling instruments such as violins and woodwind, simulation of real-world acoustics and sound synthesis.

The Convolution Theorem proves that *convolution in the time domain is multiplication in the frequency domain* (3). In other words, the filtering done by a FIR filter can be replaced with applying the DFT, changing the magnitude of each frequency component according to the filter’s frequency response, and then applying the IDFT to go back to time-domain. For this to work, the signal’s length N must be equal to the filter’s length M, which gives a time cost of **θ(N\*M) = θ(N2).**

At first glance, this does not improve the efficiency of the process at all; in fact, it is much slower in practice. But there are some things that can be done to get a significantly lower complexity.

A very efficient method of calculating the DFT is the **Fast Fourier Transform**. The most common FFT algorithm is the **radix-2 Cooley–Tukey algorithm**, previously discussed. What is so important about it is that it provides a means to reduce the complexity of the DFT to **θ(Nlog2N)** by reducing the number of redundant/repeated operations that occur in the normal DFT algorithm.

Using the FFT algorithm, we have reduced the filtering complexity down to **θ( Nlog2N).** Two problems still remain: the signal length N must be a power of 2, and the number of magnitudes in the desired frequency-response must also be N.

Most of the time, the signal will be considerably longer than the length of the frequency-response, so to deal with that we need to split the signal into frames. Doing this directly will not produce the expected results, because DFT is for periodic functions, and most likely a random portion from an audio signal will not be periodic.

Our proposed solution uses the theory of the Overlap-Add STFT processing, documented in (8). On short, OLA STFT allows splitting the input signal into frames and also dealing with the aperiodicity problem. How the OLA STFT works is that, given a frequency response of length N = 2k, the input signal is split into overlapping frames of length N. The hop-size R (the distance in samples between two consecutive frames) is N/2. Then, a window function with the constant overlap-add property at hop-size R (property that is described in (8)) is applied to each of these frames.

Because of the Constant Overlap-Add (COLA) property of the window function, if all the frames were to be added together, the result will be the original signal (Fig. 3.1). The window function also helps reducing the artifacts FFT convolution produces as a result of the frame not being periodic.

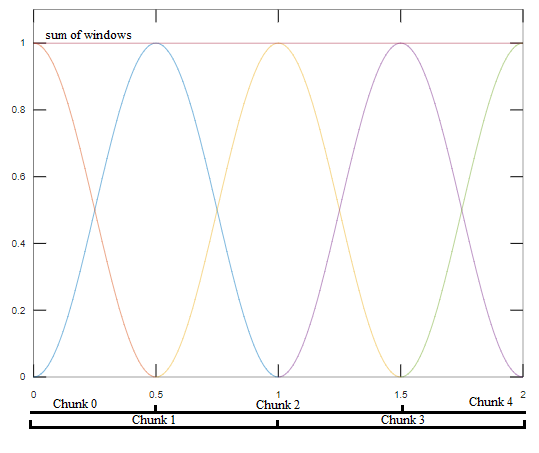
We have now got a new way of filtering signals with an arbitrary length M: split it into chunks of length N and at a hop-size R, apply a windowing function that has the COLA property at hop-size R (such as the Hann window) on each chunk, do FFT convolution, and then add all the chunks together. While knowing that the FFT convolution takes ( N log N ) operations, and there are M/(N/2) frames, this algorithm comes at a time cost of **θ( N log2N \* M / N ) = θ(Mlog2N).**

Fig. 3.2: The Overlap-Add decomposition

Chapter 4

An expandable Java-Python IPC protocol over stdin/stdout

4.1. The DLX STX ETX encoding

As previously stated in chapter 2.2, the application from (2) was implemented using two different programming languages. The main program, which handles everything related to audio signal processing such as loading, exporting, filtering and reconstructing audio data, has been implemented in Java. Because of its wide variety of data structuring and processing libraries, and especially for the Machine Learning libraries, Python was chosen for implementing the Intelligent Agent which has the job of identifying the signal portions containing audio distortions. Having two different processes instead of just one means that we have to somehow make them communicate with each other; the Java application needs to transmit audio data for classification, and the Python script needs to receive it and return the classification results.

Our previous work (2) implemented this using a rudimentary message-passing protocol over OS pipes stdin and stdou. The Java application acted as a client, sending sample classification requests, and the Python script acts as a server, by processing each request and responding with the result. This protocol was poorly designed and very hard to expand and debug. It only supported two messages:

* Sending audio data to Python - the message consists of a 4 byte integer, representing the number N of transmitted signal samples, followed by N float values;
* Reading the output of the classifier – similar to the previous message, but the 4 byte integer N represented the number of classification probabilities (i.e. the probability that each sample is damaged), followed by N float values.

Proper error reporting is almost inexistent, as robustness checking is missing, and anything that goes wrong is handled exclusively by stopping the Python script and redirecting stderr output to a file. This whole AI component can be improved by completely rewriting the Python script and the Java module that communicates with the script. We propose a new communication protocol, using the same stdin/stdout pipes, but with more robustness and expandability in mind.

The improvement of the Java-Python communication will be made using a new implementation and communication protocol, which will allow for better maintainability and expandability. The protocol will use DLE STX ETX framing to send clearly delimited messages, each having an ID, subID, length and effective data. The ID and subID will be used as message identifiers, so that each message is processed accordingly on the receiver side. Using these messages, the communication can be expanded from the previous two messages to properly report any errors, request information about the underlying AI classifier, and to send various commands to the other program. This will allow for changes in the structure of the underlying Neural Network without the need of updating the Java code.

A framing decoder extracts packets from a stream of bytes. Because the received bytes do not necessarily arrive in chunks of the same size they have been sent, we need some sort of mechanism that is able to split a stream of bytes into a sequence of packets (or frames). A very robust, simple and self-resynchronizing framing is byte stuffing, also referred to as character stuffing (9). The DLE STX ETX framing encoder/decoder relies on using the special ASCII characters DLE (0x10 – Data Link Escape), STX (0x02 – Start of Transmission), and ETX (0x03 – End of Transmission) as framing delimiters. Each frame starts with the DLE STX byte pair, followed by the data bytes (or payload), and ends with the DLE ETX pair (9). Upon encountering a decoding error, self-resynchronisation is achieved by discarding every byte until reading the next frame start delimiter. For the DLE character now acts as an escape character in the raw byte sequence, problems may occur if this character also appears in the payload. The solution is that the sender shall “stuff” an extra DLE into the data stream before each occurrence of a DLE in the payload, hence the “byte-stuffing” name. The receiving end is responsible with replacing each doubled DLE byte with a single DLE when decoding the message (9).

The encoding algorithm is pretty simple, as it can be seen in Fig. 4.1: double each DLE byte in the payload, append DLE STX to its head, and DLE ETX to its tail and the frame is ready to be sent. However, the decoding algorithm has to parse the received bytes while keeping in memory the decoding state. This can be modeled using a Finite State Machine, and an example of such a system is given in Fig. 4.2.

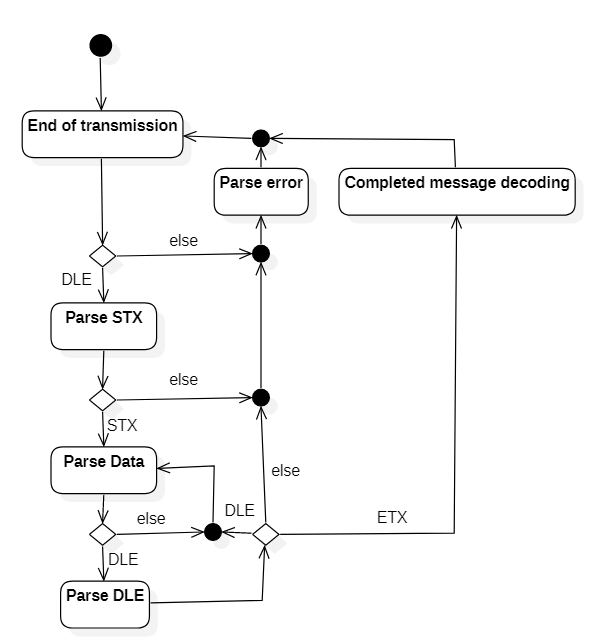
4.2. Message format

To have an extensible messaging protocol, we must define the structure of our IPC messages in such a way that it’s easy to add support for a new message without being concerned of it interfering with the already existing ones. To do that, we propose that each frame’s payload consists of a header containing a two-byte identifier, a two-byte size, and then the effective data. A visual representation of such a message can be seen in Fig. 4.3, together with its associated C-type structure, in Algorithm 4.1.

[Fig 4.3 si Alg 4.1]

This structure allows for up to 65536 unique identifiers, and for messages with a maximum length of 65536 bytes. The ID – subID pairing, noted with (ID, subID), is intended to allow messages to be grouped by functionality, i.e. messages that are used to achieve some kind of functionality can be grouped into having a common ID, and yet, they can be differentiated by their subID. Imagine the following system: component A needs to retrieve some data from component B, and initiates this communication by sending a request message (X, 1). Component B responds with multiple (X, 2) messages, representing chunks of the requested data, and ends up the communication with an (X, 3) message, indicating the transfer is complete. Because all these messages are functionally related, it makes sense to group them under the same ID, which can then be associated with the said functionality.

Fig. 4.1: DLE stuffing and framing



typedef struct

{

unsigned char id;

unsigned char sub\_id;

unsigned short size;

} msg\_hdr\_type;

typedef struct

{

msg\_hdr\_type hdr;

unsigned char \* data;

} msg\_type;

Alg. 4.1. C-type definitions of message header and structure



Fig. 4.2: DLE STX ETX decoding state chart

Fig. 4.3: Message structure

Further robustness of this message format can be added. The integrity of the message could be verified by appending a checksum footer, but because these messages are not transmitted by wire and are not subject to interference, noise or distortion, this can be omitted in our implementation, as it would only add a checksum verification overhead that brings no additional value.

4.3. List of supported messages

Using the aforementioned message structure, packed in frames delimited by DLE STX ETX encoding, the Java-Python Inter-Process Communication module supports a series of request and response messages. The following table describes each of these messages’ identifiers, direction, structure, triggering event (on the sender side) and expected action/response (from the receiving side):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Message Identifier (ID, subID) | Direction | Data structure | Description | Trigger | Action of receiver |
| (1, 1) | Java->Python | { signed long int sample\_count;  signed long int sample\_rate; } | Sent to notify the beginning of transmitting the audio data of a new sample classification request. The receiving part should expect sample\_count samples of audio data, at the indicated sample\_rate. | Occurs as part of the process of detecting damaged signal regions, initiated by the application's user. | Wait for subsequent (1,2) messages, or respond with message (3,2) if the script is not ready to start receiving audio data. |
| (1, 2) | Java->Python | { signed long int offset;  signed short int sample\_count; signed short int   samples[sample\_count]; } | A chunk of audio data, having a sample offset relative to the beginning of the current audio data transmission, and consisting of sample\_count samples. | Occurs as part of the process of detecting damaged signal regions, initiated by the application's user. | If the script is not ready to receive audio data, or if the offset or length is invalid, respond with a (3,2) message. Otherwise, store the audio data. If the current message has completed the transmission of all the samples indicated by the initiating (1,1) message, start classifying the samples. Otherwise, wait for subsequent (1,2) messages. |
| (2,1) | Python->Java | { signed long int count;  } | Sent to notify the beginning of transmitting the results of a classification request. The receiving part should expect count number of probabilities. | End of the classification process for the previously received audio samples. | Wait for subsequent (2,2) messages, or respond with message (3,1) if the sample\_count does not match the expected value. |
| (2,2) | Python->Java | { signed long int offset;  signed short int count; unsigned char  probabilities[count]; } | A chunk of classification probabilities. | End of the classification process for the previously received audio samples. | If the app is not expecting to receive classification results, or if the offset or length is invalid, respond with a (3,1) message. Otherwise, convert the probabilities into markings. If the current message has not completed the transmission of all the probabilities indicated by the initiating (2,1) message, wait for subsequent (2,2) messages. |
| (3,1) | Java->Python | { char message[message\_length]; } | Request to cancel the current operation. | Any exception, error or unexpected behavior occurred while communicating with the other process. | Reset internal state and abort current operation. |
| (3,2) | Python->Java | { char message[message\_length]; } | Request to abort the current operation. | Any exception, error or unexpected behavior occurred while processing a request. | Completely abort the sample classification process. |
| (7,1) | Java->Python | { signed long int sample\_rate;  } | Request to load classifier model trained for the requested sample rate. | Occurs as part of the process of detecting damaged signal regions, initiated by the application's user. | Attempt to load the model into memory. Reply with a (7,2) message with zero status if successful, and non-zero if unsuccessful. |
| (7,2) | Python->Java | { signed long int status;  signed long int sample\_rate;  signed short int input\_cnt;  signed short int output\_cnt;  signed short int offset;  } | Reply of a (7,1) message, with status and information of the currently loaded ML model. | Receiving a (7,1) message. | Store the classifier information if the status indicates success, otherwise abort the classification process. |
| (8,255) | Java->Python | None | Request to terminate the Python script. | A fatal communication error has occurred (such as a timeout), or the application is closing and needs to terminate its child processes. | Peacefully notify the operating system that my life is over. |

Table 4.1: List of supported messages of our Java-Python IPC

Chapter 5

Application updates. Experimental tests and results

5.1. FFT filtering using OLA STFT

This subchapter goes through the process of implementing the Overlap-Add Short-Time Fourier Transform algorithm, improving the FFT implementation, overcoming issues caused by this algorithm as it is, and finally, comparing the performance results with equivalent Finite Impulse Response filtering.

The OLA STFT implementation was done as per the method described in chapter 3.3. The only mentionable aspect of it is that the frequency domain equalization coefficients are built based on the desired impulse response. Using this, constructing OLA STFT filters equivalent to a given FIR filter is a straight-forward task. The impulse response is zero-padded to match the FFT length, and then circularly shifted so that the centermost sample of the impulse response becomes the first sample. The zero padded impulse response is the transformed from time-domain to frequency domain using FFT, and the result is to be used for FFT convolutions.

5.1.1. Improvements to the existing FFT implementation

In practice, it is not that simple to obtain the theoretical results of the Fast Fourier Transform. The implementations for convolution, element-by-element array multiplication, windowing and others are trivial and close to optimal performance can be achieved without too much effort. But implementing the FFT without thinking of memory access, data structures, call overhead and redundant operations can result in having the FFT convolution perform worse than the FIR filter.

Our previous work already contains an implementation of the FFT, but it was made without any performance considerations. Next, we will try to minimize the running time of the algorithm using various computing and memory access optimizations. Remember that, for a filter of length M and N input samples, the FIR filter does N\*M multiply-additions, or in **θ(N\*M)**. One FFT convolution has a complexity of **θ(M\*log2M),** where M is the window size and must be a power of 2. As the OLA STFT method splits the signal into pieces of length M, and because we are using a windowing function with hop-size R = M/2, the overall complexity of the OLA STFT is **θ(N/(M/2)\*M\*log2M)**, or **θ(N\*log2M),** where N is the full signal length.

All diagrams presented contain the execution time of filtering one minute of audio data sampled at 88200Hz, for a total of 5,292,000 samples, and with filter lengths in the range [22, 214]. The presented execution times will each be the average of 10 runs. The system used to run the code and measure its performance is based on an Intel i3-7100 CPU, 3M Cache, @3.90 GHz, with DDR4 RAM memory clocked at 2400 MHz. The program is implemented in Java 8, without using any external library.

The OLA STFT algorithm involves more computations than a FIR filter with the same length. This happens because, while FIR only does a series of multiply-addition operations, FFT uses a larger number of complex operations. Because of this, OLA STFT is faster than its FIR counterpart only if the filter is long enough. Some sources[[7]](#footnote-8) indicate that the minimum filter length is around 26, while others[[8]](#footnote-9) push down this limit to 20-50 (10).

The original FFT implementation, which we will note as **FFT\_0**, is using a class for complex numbers, and complex number operations are done by calling the corresponding method of the class. Each time the FFT function is called, the input signal is copied into a newly allocated array, the bit reversal table is generated, and the Butterfly (described in Chapter 3) is done as another function call for each value pair. The sine/cosine values for the Twiddle factor are computed each time the Butterfly function is called. The results of this implementation (see Fig. 5.1), however, do not come even close enough to the expected performance benefits, as the algorithm performs better only on filters longer than 212.

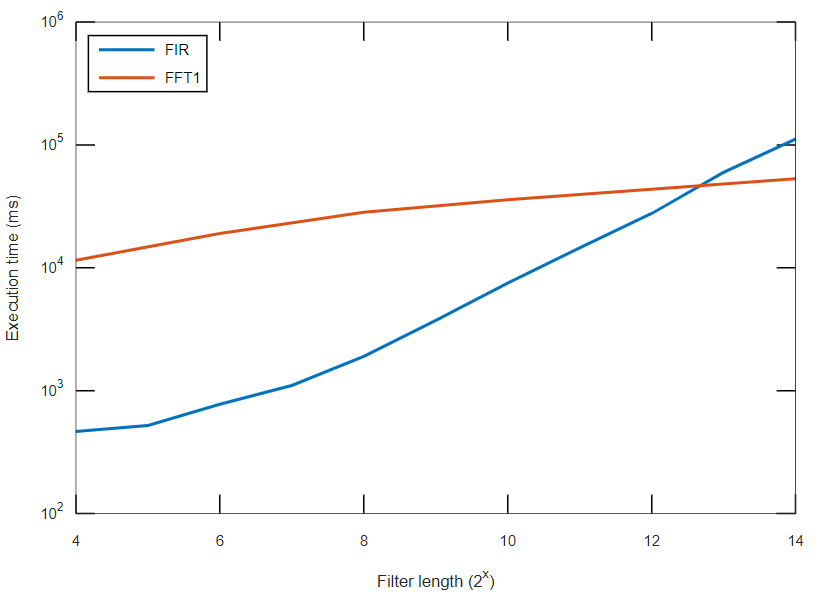


Fig. 5.1: FIR filtering vs original FFT implementation

Some of the issues of FFT\_0 are easy to notice: an unnecessary memory allocation overhead, and large function call overhead. The next implementation attempt, noted as **FFT\_1**, tries to solve these problems by making the FFT perform in-place (on the input array), and by replacing the function calls associated with complex operations with standard floating point operations. This alone reduced the execution time six-fold, as it can be seen in Fig. 5.2.

Let’s think about how the FFT method is called hundreds of times when processing that minute of audio data, all calls being made for the same FFT length. That means computation of bit-reversals, sin and cos are done hundreds of times for the exact same input arguments. We can trade some memory for a better execution time by creating bit-reversal and sin/cos tables. In the **FFT\_2** implementation**,** those tables will be computed once and then used in every FFT call. The execution-time saved by this algorithm can be seen in Fig. 5.2, and we can observe that the execution time of FFT\_2 is ~3 times better than FFT\_1.

Finally, the last optimization done is using two separate arrays for the real and imaginary parts of the complex numbers instead of an array of Complex objects. As we are working in Java, the array of Complex objects would be in fact an array of references, with the independently allocated objects possibly scattered through the memory. Using arrays of primitive types (floats in this case) ensured that the entire working data is contiguous and thus, cacheable. Results of this **FFT\_3** implementation are shown in Fig. 5.2, together with all the algorithms presented here.

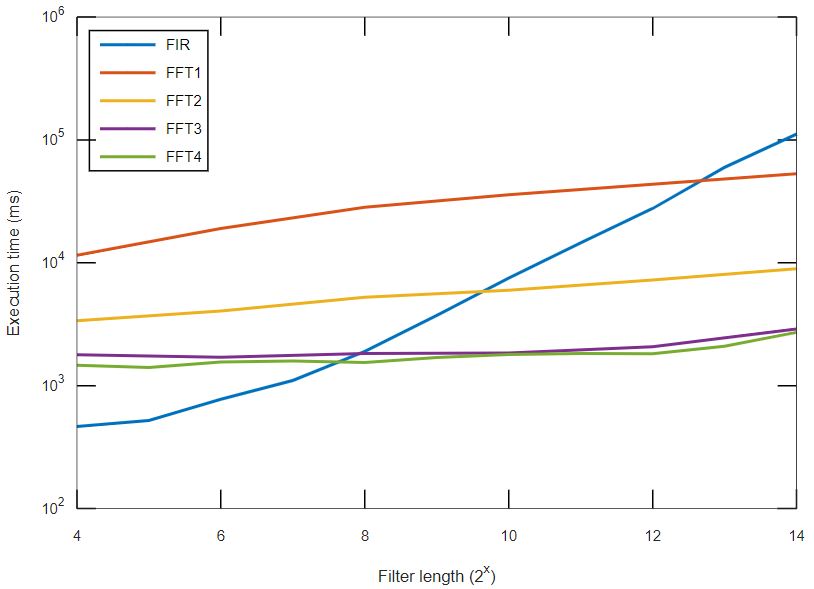
[ big pic ]

Fig. 5.2: Performance of all the 5 algorithms discussed here

5.1.2. Unexpected discrepancies and the proposed fix

During the testing of the OLA STFT implementation and comparing its results with the expected signal (computed using FIR filtering), we have observed periodic discrepancies between the actual and expected signals (Fig. 5.3). The period of these differences was matching the hop-size of the OLA window, which led us into believing this is a problem caused by the cyclic nature of the Fourier Transform.

TBD

Fig. 5.3: Aliasing effects near the FFT convolution intervals

To solve this issue, we tried using zero-padding to the left and right of the signal, in hope that it will attenuate the effects of signal acyclicity. The quantity of each zero-padding (left and right) was chosen to be the half of the signal length, so that the new length is also a power of 2 and FFT can be used. This solution proved satisfactory and successfully reduced the periodic artifacts near the window margins, with the output closely resembling the results of a standard FIR filtering.

However, it seems that this is an already known behavior of the OLA STFT. Source (10) makes an observation on the cyclic nature of the FFT convolution, and explains its effects in more detail. The reason for the observed discrepancies is that, during the DFT, the tail is added to the signal’s head, and this is not wanted and does not normally happen during normal convolution. This effect is called aliasing, and it can be overcome by appending L – 1 zeros to the input signal, where L is the impulse response length. This has the cost of processing less usable data during one OLA STFT step, but knowing this information helped us with choosing the minimum amount of zero-padding, instead of using the heuristic described in the previous paragraph.

5.1.3. Performance results of OLA STFT filtering vs FIR filtering

After applying the described FFT improvements, implementing the OLA STFT method and making sure we do a noncyclic convolution, we obtained a filtering algorithm equivalent to the FIR filtering, but with a way better time-performance. The results of these two filtering methods, embedded into the main Java application, are shown in Fig. 5.4. Interpretation of the results.

[TDB]

Fig. 5.4: Performance comparison between FIR filtering and OLA STFT filtering

5.2. The new IPC protocol and Python classifier

In this subchapter, we will go through some aspects of the Java-Python Inter-Process Communication module implementation and design, and will discuss the design, training and results of the new intelligent classifier. The aim of these changes is to improve the expandability of the IPC messaging system, to decouple the Java client from the underlying Neural Network model structure, to significantly decrease the running time of the classification process, and to increase the classifier’s accuracy.

5.2.1. Message processing, throughput and robustness

The first two objectives are achieved through the new IPC protocol. Expandability is achieved through each message having its unique identifier, and adding support for a new message will not have any impact on the already existing messages. Previously, the Neural Network’s number of inputs and outputs was also hardcoded in the Java code, because they are needed when sending the signal data for classification. This meant that any change in the NN’s structure also had to be reflected in the Java code. Using the new messaging scheme, the classification process starts by requesting a classifier load (see message (7,1), described in chapter 4.3). The response of this request contains the loaded model’s number of inputs, outputs, and the classification offset (the offset of the first input that has a corresponding classification output, i.e. if the model has 129 inputs and one output, and that output is the class of input #65, the offset is 65), and these values are to be used instead of the old hardcoded ones.

The messages are passed from one process to another using the Python process’s stdin and stdout pipes as a mean of directly sending binary data from one process to another. The Python script is exclusively launched by the main Java application. This makes the classifier a child process of the Java program, thus allowing the latter to access the stdin/stdout pipes of the child process.

Messages are processed in a periodic process loop, called IOP (Input/Output Processing) loop. This means that, at a fixed clock rate, messages are read from the pipe, processed, and then output messages are sent. The same processing scheme, which is summarized in Algorithm 5.1, is used in both the Java and the Python implementations.

One of the biggest problems encountered while testing this message-passing scheme was the blocking of pipe read/write commands. OS pipes, such as stdin, stdout and stderr, have limited capacity, and by default, the read/write operations on them are blocking (11). When attempting to read from an empty pipe, the system call will block until there’s some data available. This is not wanted, as it is absolutely normal and expected that not all of the IOP cycles will receive data. For example, while sending a series of audio signal messages – no input is expected during this process. However, completely bypassing the reads during this process is also not an option: an abort command might arrive, and we would surely want to process that as soon as possible. The issue of blocking read operations was solved by checking whether there is available input data or not before attempting to do a read system call.

Similarly, if a process attempts to write to a full pipe, the write system call blocks until sufficient data has been read from the pipe to allow completing the write operation (11). Unlike reading a message, writing one cannot be postponed; we must keep this operation blocked until the receiving process reads sufficient data from the pipe, and the write can complete. However, this may result in a deadlock condition: both processes A and B are blocked on a write and thus cannot read input data from the other process to unblock it. In our application, this is avoided by means of two mechanisms. First, the messaging sequence will never be in a state where both processes send large amounts of data at the same time; Python will not send anything while Java sends signal data, and Java will not send anything while Python replies with classification data. The second mechanism is limiting the maximum number of bytes written in an IOP cycle to the pipe’s capacity. Because the other process always reads all the available data from the current process’s output pipe, a write operation will never attempt to write to a full buffer if the number of bytes written is limited, so the IOP loop will never block on a write.

// The running condition is different in Java and Python.

// Java runs the loop until a timeout is exceeded, an

// exception occurs, or until the current classification round completes.

// Python runs the loop continuously until a fatal exception is encountered or

// a (8,255) message (TERMINATE command) is received.

while( run\_condition is true )

{

// Store the current millisecond clock time

crnt\_tick = SYS\_current\_clock\_ms();

// Process all the available input data

while( true )

{

// Read and store bytes from the pipe into an internal buffer.

bytes\_read = IOP\_get\_bytes();

// Parse bytes from the internal buffer into a message.

// Return a decoded message if available, or NULL otherwise

msg = IOP\_parse\_message();

if( msg != NULL )

{

// Process the input message according to its ID and subID

// This also includes sending reply messages, where applicable

process\_message( msg );

}

else

{

// If the pipe is now empty and the buffer contains no more messages,

// exit the input processing loop

if( bytes\_read == 0 )

break;

}

}

// Send scheduled output messages, i.e. those that are not immediate replies

send\_scheduled\_messages();

}

Alg. 5.1: I/O processing loop

In theory, it seems like the processing loop should work fine and without blocking. In reality, we had a pretty major issue with the Python script blocking on a read/write operation. After secular battles of trying to debug a program that gets stuck after a seemingly random number of successfully processed messages, we found that the problem was that our debug prints, which were written to stderr, were piling into the pipe until it got full and blocked, as the receiving end (Java) was not reading anything from that pipe (as designed).

At a processing rate of maximum 1 millisecond between consecutive IOP cycles, with a maximum baud rate of 16384 bytes per cycle (a presumably safe value, considering the pipe capacities described in (11)), the new IPC protocol allows for very good transmission speed. Of course, we still need to consider the overhead implied by encoding and decoding the raw bytes. In terms of pure throughput, the new communication protocol is clearly slower. The initial implementation had no encoding/decoding phase, and data was sent one value (4 bytes) at a time. Reads were blocking, but the simplicity of the communication did not allow the pipes to get filled up, as every byte written was immediately read.

5.2.2. Generating the training data

To have training data for the neural network, we need some digital recordings of vinyl records, and we need to have each sample labeled as being part of a distortion or not. We named the intervals of consecutive distorted samples as “markings”. Because we want our dataset to contain lots of input data variations, we need lots of labeled audio data – on the order of millions of training samples. It’d impractical to do this manually. Instead, we used the same technique as in our previous work (2): mono records. Stereo grooves and cartridges were designed to be backward compatible with monaural records containing a laterally modulated groove. Because of this, when played through a stereo cartridge, a mono recording will generate the same signal in both left and right channels of the cartridge. In practice, there are signal differences between the channels, mostly caused by imperfections of the cartridge’s position, channel imbalance, tonearm resonance, inherent surface noise, and groove damage. Balance the channels manually, filter the signal using an inverse RIIA filter to accentuate high frequencies and a high-pass to remove low frequencies which are not usually affected by groove damage, and we’re left with a signal that represents the difference between the two grooves. As this difference is supposed to only contain surface noise, we can use this to generate our markings by simply marking each region where the difference signal is greater than a certain threshold. The generated markings were tied to both channels of the original signal.

For the training data, we used some of the sources presented in our previous work, but also quite a few new recordings. All the recordings used, together with the specifics of each one, are listed in Table 5.1. One can notice that some of them contain few to no distortion. This is intended, and was done to prevent lots of false positives in heavily modulated passages. For example, because of the Shostakovich and Dvořák recordings contained in the dataset, all passages containing lots of high frequency components, such as cymbals, “s” sounds and trumpets were marked as being damaged, even if they actually had no damage whatsoever. These mint-condition recordings contain signals that resemble heavily damaged signals, but are in fact in perfect condition: bagpipes, heavy metal music, vocals, brass instruments (trumpets and horns), and loud cymbal crashes.

The markings are, however, just some intervals, and are meaningless without their associated signal. As the training data needs to be structured into inputs and outputs, we need a means of exporting audio files and markings into usable training data files. We implemented the training data generator within the Java application. As the number of labeled data is very large, much larger than what would be practical to use as a training set, we only export some of it. The two classes are very imbalanced – there are way more undamaged samples than damaged ones, and the learning algorithm is sensible to class balancing. In an effort to reduce class imbalance, we used a probability of exporting a marking, and a different one, smaller, for exporting unmarked samples. Table 5.1 also presents the number of training samples generated from each of the marked recordings, and the class balance ratio.

The results achieved in (2) showed that preprocessing the signal by applying the inverse RIIA curve significantly increased the classifier’s accuracy, so we used the same preprocessing for our training datasets.

Table 5.1: Recordings used in generating the training datasets

Because the training set is large enough to not fit in the memory of an average computer, we created aData Generator, which can be used to load the training and testing data from files and return it in batches rather than as a whole. This way, we need far less memory, and the training can run even on average computers. This is in contrast with the previous model implementation, which loaded all data in memory and had to be trained on a supercluster node with more than 64GB of RAM.

5.2.3. Training the classifier. Accuracy and performance

Achieving a better accuracy proved to not be easy at all. We decided on using a convolutional neural network, implemented using the Keras framework, instead of the sklearn Multi-Layered Perceptron used in (2). By using a convolutional network, we can classify more samples at the same time, thus increasing the time efficiency of the classifier and thereby decreasing the time required to generate markings for a given input signal.

The metric we hoped to improve was the overall F1 score[[9]](#footnote-10), while using a binary-crossentropy loss function and the default Adam optimizer. To mitigate the impact of class imbalance, we used different weights for each of the two classes while computing the loss. The first attempts proved to be a failure. F1 score quickly stopped increasing and did not go over 50%. The results of these attempts can be seen in columns 1-2 of Table 5.2. The notation of the Structure should be interpreted as follows: a Dense( X ) is a densely connected layer with X outputs, and a Conv( K, F, S ) is a convolutional layer with a K-length kernel, S strides (which is optional and defaults to 1) and F output features.

After these attempts, we refined the dataset by removing all the markings where the signal reconstruction algorithm did not significantly change the original signal. The logic behind this is that the markings were duplicated on both channels of the input signal, even though the damage may have been present in only of the channels.

With the refined dataset, we tried once again to train some model structures. Because each output represents the same thing, i.e. a single sample’s class, we used convolution kernels that were similar to the model used by our previous work. In that, the model was a densely connected network with 129 inputs, two hidden layers of 64 and 32 nodes, and one output with the offset of 65 (the class of the 65th input sample). In convolutional network terms, that was equivalent with the first layer having a kernel of length 129 and 64 features, the second layer – a 1-length kernel with 32 features, and finally, a 1-length kernel with 1 feature (the output). The results of the tested multi-output convolutional network models can be seen in columns 3-12 of Table 5.2.

Because the F1 score failed to improve after reaching ~70%, we suspected there might be a bug in our code that’s preventing the correct training of the model. After comparing a Keras model with the sklearn model that we used in (2), we added a StandardScaler[[10]](#footnote-11) for data preprocessing the input of our model, just the same as our previous work did. After this change, the results of training the two models on the same dataset became similar enough to prove the equivalence of the two models.

Knowing that the Keras model works as good as the previous one, we moved our attention to the training process variables: optimizer parameters and dataset. To visualize each model’s performance, we also improved the evaluation algorithm to output a confusion matrix instead of combined metrics. Also, we reverted to using an one-output model, which can be later expanded to a convolutional multi-output model by simply copying the weights: a densely connected model with structure “Input( X ), Dense( D1 ), Dense( D2 ), Dense( 1 )” is equivalent to a convolutional model with structure “Input( X, 1 ), Conv( X, D1 ), Conv( 1, D2 ), Conv( 1, 1 )”, with the later being equivalent with one convolution step of a “Input( X + Y - 1, 1 ), Conv( X, D1 ), Conv( 1, D2 ), Conv( 1, 1 ), ( => Output( Y, 1 ) )” model, where Y is the number of outputs, and X is the number of inputs which do not have a corresponding output label.

We trained the model from column 1 of Table 5.3 for over 200 epochs, and its evolution in time can be seen in Fig 5.5. Although it went over the 70% threshold, the positive class F1 score did not come near the 76% achieved by (2).

TBD

EOP->

Table 5.2: Results of different Convolutional Neural Network models and datasets



Table 5.3: Results of same Convolutional Neural Network structure with parameter variations (TBD)

# Bibliografie

1. **The Recording Industry Association of America.** U.S. Recorded Music Revenues by Format. [Online] [Cited: 06 03, 2020.] https://www.riaa.com/u-s-sales-database/.

2. *Distortion detection and attenuation on recordings from mechanical analog audio formats.* **Alexandru, Drimba.** s.l. : Babeş-Balyai University Cluj-Napoca, 2018.

3. **Julius Orion Smith III.** *Introduction to Digital Filters with Audio Applications.* https://www.dsprelated.com/freebooks/filters/ : Online Book, 2007.

4. **M., Kathirvelu.** Chapter 5: FIR Filter Design. *Certain Investigation On Optimized Area And Power Delay Product In Digital Circuit Applications.* http://shodhganga.inflibnet.ac.in/bitstream/10603/24055/10/10\_chapter%205.pdf.

5. **Iowegian International.** FIR Filter Basics. [Interactiv] [Citat: 05 06 2020.] http://dspguru.com/dsp/faqs/fir/basics/.

6. **Jose Maria Giron-Sierra.** *Digital Signal Processing with Matlab Examples.* s.l. : Springer. ISBN 978-981-10-2534-1.

7. **Zölzer, Udo.** Chapter 5: Equalizers. *Digital Audio Signal Processing.*

8. **Julius Orion Smith III.** Spectral Audio Signal Processing. [autorul cărții] Overlap-Add (OLA) STFT Processing. s.l. : Online Book, 2011. https://www.dsprelated.com/freebooks/sasp/Overlap\_Add\_OLA\_STFT\_Processing.html.

9. **Worcester Polytechnic Institute.** Networks: Bit and Byte Stuffing. [Interactiv] [Citat: 05 06 2020.] https://web.cs.wpi.edu/~rek/Undergrad\_Nets/C04/BitByteStuffing.pdf.

10. **Selesnick, I.W. & Burrus, C.S.** Chapter 8. Fast Convolution and Filtering. *Digital Signal Processing Handbook.* http://dsp-book.narod.ru/DSPMW/08.PDF.

11. Linux Programmer's Manual. *Overview of pipes and FIFOs.* [Interactiv] [Citat: 08 06 2020.] https://www.man7.org/linux/man-pages/man7/pipe.7.html.

12. **Siemens.** Introduction to Filters: FIR versus IIR. [Interactiv] [Citat: 05 06 2020.] https://community.sw.siemens.com/s/article/introduction-to-filters-fir-versus-iir.

1. \*\*\*, History of the Cylinder Phonograph, <https://www.loc.gov/collections/edison-company-motion-pictures-and-sound-recordings/articles-and-essays/history-of-edison-sound-recordings/history-of-the-cylinder-phonograph/>, retrieved 04.06.2020. [↑](#footnote-ref-2)
2. \*\*\*, The History of Audio Recording, <https://lossenderosstudio.com/article.php?subject=10>, retrieved 03.06.2020 [↑](#footnote-ref-3)
3. Graham Barber, Gramophone Needles, <http://www.graham-ophones.co.uk/needles/4591285693>, retrieved 03.06.2020 [↑](#footnote-ref-4)
4. \*\*\*, Pickup Cartridge History, <http://www.enjoythemusic.com/cartridgehistory.htm>, retrieved 03.06.2020 [↑](#footnote-ref-5)
5. Barlow, *Groove deformation in gramophone records*, 1958 [↑](#footnote-ref-6)
6. Filter Design, <http://www.circuitsage.com/filter.html>, retrieved 03.06.2020 [↑](#footnote-ref-7)
7. FFT convolution speed improvements, <http://www.dspguide.com/ch18/3.htm>, retrieved 07.06.2020 [↑](#footnote-ref-8)
8. [↑](#footnote-ref-9)
9. Jason Brownlee, **How to Calculate Precision, Recall, F1, and More for Deep Learning Models,** <https://machinelearningmastery.com/how-to-calculate-precision-recall-f1-and-more-for-deep-learning-models/>, retrieved 08.06.2020 [↑](#footnote-ref-10)
10. sklearn StandardScaler documentation, <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html>, retrieved 08.06.2020 [↑](#footnote-ref-11)