Master's Thesis : Sound synthesizer programming using deep learning

Auteur : Vecoven, Frederic
Promoteur(s) : Louppe, Gilles
Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil en science des données, à finalité spécialisée
Année académique : 2019-2020
URI/URL : http://hdl.handle.net/2268.2/9013

University of Liege

Faculty of Applied Sciences

Sound synthesizer programming using deep learning

Author:
Frederic Vecoven

Supervisor:
Prof. Gilles Louppe

Master thesis submitted for the degree of
MSc in Data Sciences Engineering

Academic year 2019-2020

Acknowledgments

First I would like to thank Prof. Gilles Louppe for the time he spent working with me as well as his availability. It makes no doubt that the results of this thesis would not have been the same without his many advises. I am also grateful to him for allowing me to work on such an interesting topic.

I would then like to thank my son Nicolas for his availability, his expertise in machine learning and for having borrowed some of his code.

Finally I would like to address my gratitude to my wife for her support.

Liège, May 2020

Contents

Chapter 1

Introduction

Music is a form of art, which derives from the Greek word meaning "art of the Muses". Music has been performed since the dawn of human time with instruments and through vocal song. While it is not certain how and when the first musical instrument was invented, historians point to early flutes made from animal bones that are at least 37000 years old. The oldest known written song dates back 4000 years and was written in ancient cuneiform. Instruments were created to make musical sounds. Any object that produces sound can be considered a musical instrument and most particularly if it was designed for that purpose. Amongst these instruments are synthesizers which use some method of sound synthesis.

Sound synthesis can be defined as the production of sound that originates electronically from scratch by either analog or digital circuitry, as opposed to sound whose origins derive from recorded or sampled real-world sonic events.

Programming a synthesizer can be an art by itself. Considering the number of parameters supported by most synthesizers, finding a set of parameters to generate a wanted sound is a difficult task. Humans typically start from existing sets and modify parameters one by one until obtaining what they are looking for, using a trial-and-error approach.

The goal of this project is twofold: reverse engineer a synthesizer in order to build a simulator and use deep learning techniques to solve the problem of finding a set of parameters (or distributions of parameters) given a sound.

In this document, we will describe the complete approach taken to solve this problem.

- In chapter 2, we will do a quick review of sound synthesis history, where it stands today and why we are interested in analog synthesizers.
- In chapter 3, we will do a deep-dive in the Roland JX synthesizer: its hardware architecture as well as the software running on its microcontrollers.
- In chapter 4, we will build a simulator based on the knowledge learnt from the reverse engineering done previously. This will allow us to create datasets to be processed by our neural networks.

- In chapter 5, we will review and use deep learning techniques to solve the problem of finding synthesis parameters given a sound.
- Chapter 6 describes the results that we have obtained using the different algorithms from chapter 5.
- Finally, we will conclude and talk about potential enhancements to your solution.

Chapter 2

Sound synthesis

Sound synthesis can be defined as the production of sound that originates electronically from scratch by either analog or digital circuitry, as opposed to sound whose origins derive from recorded or sampled real world sonic events.

Sound synthesis is actually pretty old. Patented in 1895, the Telharmonium can be considered the first significant electronic musical instrument and was a method of electro-magnetically synthesising and distributing music over the new telephone networks of victorian America[1]. In the mid-1960's, Robert Moog released a series of self-contained modular instruments that made voltage-controlled synthesis affordable to many studios. These modular synthesizers were made of building blocks connected together using "patch cords". In 1971, the Minimoog was released. It was designed to be affordable, portable and was the first synthesizer sold in retail stores.

Figure 2.1: The Minimoog

Synthesizers generate audio through different methods including subtractive synthesis, additive synthesis and frequency modulation synthesis. The sounds are shaped and modulated by other components such as filters, envelopes and low-frequency oscillators.

[1] `http://120years.net/the-telharmonium-thaddeus-cahill-usa-1897/`

2.1 Subtractive synthesis

Creasey [5] describes subtractive synthesis as an approach in which *filtering is the principal tonal modification technique*. The sources to be filtered are generally simple waveforms (triangle, rectangle, sawtooth) or noise generators.

The most basic example of subtractive synthesis is shown at Figure 2.2.

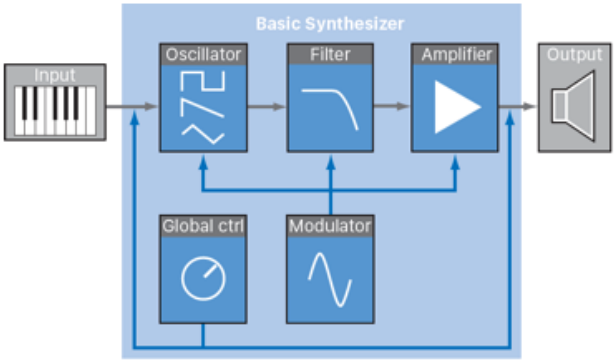


Figure 2.2: Basic subtractive synthesis

2.1.1 Oscillator

The oscillator generates a repeating signal in a particular waveform containing all of its dynamic range and harmonic frequencies. A **sine** waveform will produces a smooth and mellow sound and is often used to create pipe sounds. The **sawtooth** waveform creates a strong "buzz" sound. It contains harmonic frequencies which sound rich and full and great for powerful synth bass and lead sounds. The **square** waves have a sound that is rich in harmonics, not as "buzzy" as the sawtooth but not as smooth as the sine. They have half as many harmonic frequencies as the sawtooth and are use for nasal sounds, such as those created by wind instruments such as a clarinet.

2.1.2 Filter

The filter (often called VCF, Voltage Controlled Filter) is used to filter out unwanted frequencies from the raw oscillator sound. Most synthesizers rely on two kind of filters: a lowpass filter (LPF) and highpass filter (HPF). The lowpass filter allows the low frequencies to pass through cutting off the high frequencies. A highpass filter does the opposite. The frequency at which the sound starts to be affected is called the cutoff frequency. Another important parameter of the filter is the resonance. Most filters have a resonance (often called Q control). Resonance occurs when the sound in the same range as the cutoff frequency is routed back to the filter, creating feedback. VCF can be affected by an envelope (which will be described with the amplifier).

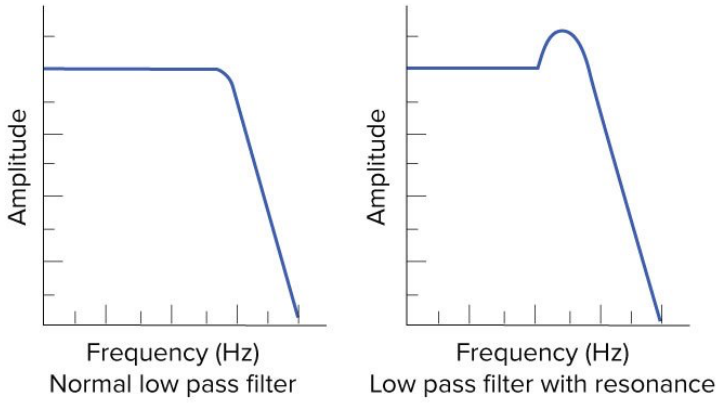


Figure 2.3: Lowpass filter with and without resonance

2.1.3 Amplifier

The next step in the chain is the amplifier, also called VCA (Voltage Controlled Amplifier), which controls the volume of the sound. VCA can be affected by a volume envelope. The envelope allows to sculpt the tone, shaping the waveform to create the sound we are looking for. Envelopes (Figure 2.4) can be broken in four parts: attack, decay, sustain and release.

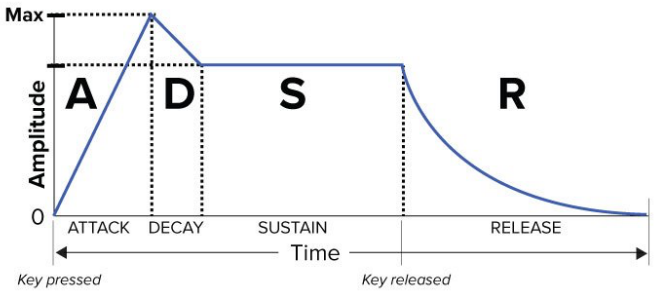


Figure 2.4: ADSR envelope

The envelope is a key in shaping the sound we want to create. Let's compare a drum and a violin : the drum has a sharp, sudden increase in volume (attack) with almost no sustain and a short release time. The violin has a longer and slower attack, building to maximum volume, a long sustain and the sound tails off slowly.

Envelopes don't just apply to the amplifier. Many synthesizers have an envelope on the filter and/or the pitch (oscillator frequency).

2.1.4 LFO

The LFO (Low Frequency Oscillator) is another oscillator using similar waveforms as the main oscillator but its frequency is so low that it is inaudible. Rather than being used to create a tone, it is use to manipulate other parameters. Typical LFO parameters are the rate (its frequency) and depth (amount of LFO applied to a parameter). Many synthesizers include controls to set the LFO to affect pitch

(which creates a vibrato effect), volume (creates a tremolo) or filter cutoff. Having the LFO control the filter creates sounds with a wah-wah effect or wobble basslines.

2.2 Additive synthesis

Additive synthesis is the oldest form of sound synthesis dating back to its application in pipe organs. It is based on techniques based on summation of elementary waveforms to create a more complex waveform. Typically sine waves are used because they don't contain harmonics. We can for example create a square wave by adding a series of sine waves, one sine being the fundamental frequency and the others being the odd harmonics (added in decreasing amplitude).

There is no need to add more details since the principle is trivial. The main drawback is the need to many many oscillators to produce interesting sounds, which makes the technique impractical for real synthesizers.

2.3 Frequency modulation synthesis

FM sound synthesis was first described in a paper [3] from Stanford. Yamaha bought the patent and produced synthesizers based on FM, making the famous DX7 released in the early eighties.

FM (Frequency Modulation) synthesis is based on the use of two oscillators: a carrier and a modulator. The carrier is used as the fundamental frequency of the sound (and is controlled by the keyboard) while the modulating frequency is used to alter the carrier at a specific modulation rate and modulation intensity. Mathematically this is based on the concept of sideband frequencies. When a signal is frequency modulated, sum and difference tones appear.

FM signal can be expressed with the equation

$$Y = A_c \sin(2\pi f_c t - \frac{\Delta f}{f_m} \cos(2\pi f_m t)) \tag{2.1}$$

FM generates frequency components in the output waveform that are not necessarilly harmonically related to the carrier or modulator frequency which in turn affects the sound produced. FM produces an infinite number of side bands (brighter sounds) at frequencies $f_{sb} = f_c \pm n f_m$.

Yamaha uses 6 oscillators in the DX7. They modulate each other in various combinations as shown as Figure 2.5. Each oscillator has its own envelope. This complex synthesis generated a new variety of sounds (revolutionary at the time) which contributed to the huge success of FM synthesis. Programming these synthesizers was very difficult due to the non-intuitive nature of FM.

2.4 Modern sound synthesis

Today, processing audio even at a high sample rate, has become a trivial task for a CPU or GPU. In the beginning era of sound synthesis, this was not the case. There

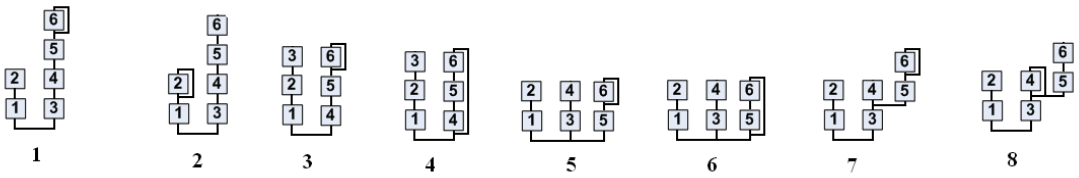


Figure 2.5: DX7 oscillators configurations

was no GPU at that time and the CPU were very limited: they ran way too slow for generating or processing audio in real time.

Modern sound synthesis can pretty much generate anything, the number of oscillators is virtually unlimited, envelopes are multi-points, filters can have any shape, etc... Many techniques are used to try to reproduce the sound of the early synthesizers but it appears to be difficult to reproduce the sounds produced by analog synthesizers which, by definition, don't produce discrete sounds. These synthesizers became "classic" and highly sought machines with crazy prices for some models.

We happen to own a Roland Super JX synthesizer, which was the last analog synthesizer ever produced by Roland. It has 12 voices, each having 2 oscillators, analog filters (resonant lowpass and highpass), analog amplifier and chorus. When trying to replicate a sound on this synthesizer, the manual method is often trial-and-error. We start with a sound which sounds like the sound we want, then modify parameters one by one until reaching the desired output. Our motivation for this project is to fully automate this process.

Chapter 3

The Roland Super-JX

In order to write an accurate simulator of the Roland JX, we have to fully understand how it works internally. The good news is that Roland released schematics of their synthesizers and some high-level overview on how it operates. The bad news is that this is far incomplete and nowhere close to the level of knowledge required to develop a simulator.

3.1 High-level overview

The Roland documentation provides a high-level overview of the synthesizer which we provide at Figure 3.1. We find the following elements:

- an assigner board : this board contains a CPU (Hitachi HD6303), 32KB of ROM, 10KB of RAM, midi interface and a gate-array[1] chip.
- two sound boards : these boards are the sound generators. Each one contains another CPU (Intel 8031) and analog circuitry.
- several other smaller boards : display, buttons, output buffers, ...

We will detail the assigner board and the sound boards in this chapter.
First, let's start with some definitions used with this synthesizer.

- A sound produced by a sound board is referred as a **tone**. A tone is made from a set of parameters and this is what we are interested in for this project.
- A **patch** is made of one or two tones. The Super JX synthesizer can play up to 12 voices in *whole* mode or up to 6 voices in *dual* mode. The patch is the structure which encompasses up to two tones and some extra patch specific parameters such as a detune factor between the tones, how notes are assigned, etc...
- The **assigner board** is the hardware dealing with the human interface (front panel, keyboard, midi interface) and responsible to assign notes to the sound board voices.

[1] Early incantation of CPLD/FPGA chips

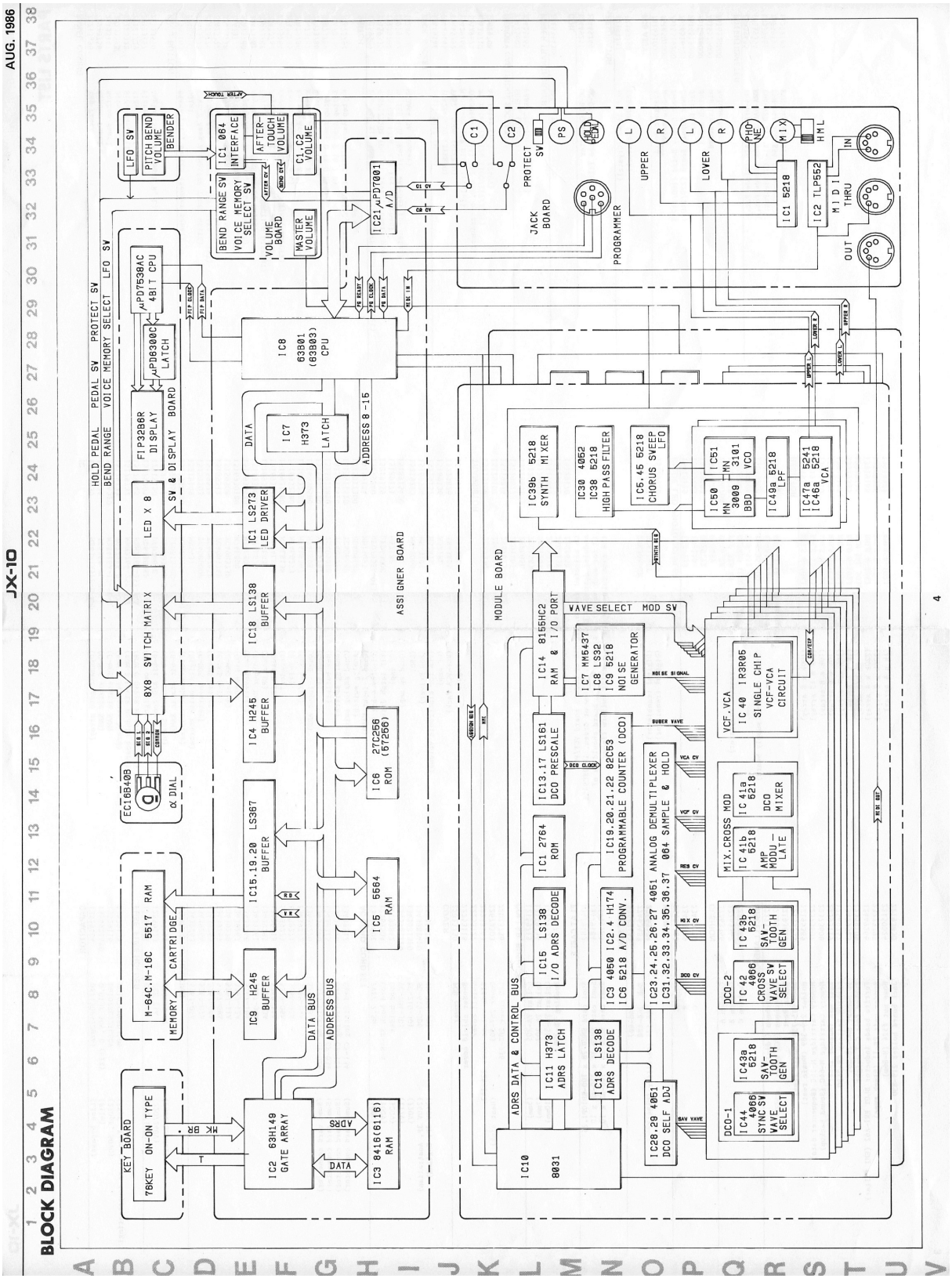


Figure 3.1: Roland JX high-level overview

- The **sound board** is an digital/analog board capable of generating up to 6 voices.
- **Midi** is a standard interface used to interconnect synthesizers and computers.
- A **DCO** is a Digitally Controlled Oscillator. Early analog synthesizer used **VCO** (Voltage Controlled Oscillator) which suffered from temperature stability. By using a crystal to control the frequency, DCO are stable and don't require adjustment. DCO produces sawtooth, square waves (sometimes PWM, Pulse Width Modulated) and noise signals.
- A **VCF** is a Voltage Controller filter. Typically a lowpass filter with a -12dB/octave or -24dB/octave response.
- A **VCA** is a Voltage Controller Amplifier. The gain of the amplifier is controlled by an input voltage. Typically the input voltage is linear and the output gain is exponential.
- An **envelope** is a time dependent function which can be applied to DCO, VCF, VCA (and other entities) to vary their action across time.
- A **LFO**, Low Frequency Oscillator, is an oscillator in the range of 0.1 Hz to 20 Hz which can be applied on control voltages to modulate their action.
- A **CV**, Control Voltage, is a DC voltage applied to an entity (VCF, VCA,...) to control this entity. It is typically generated from a DAC which is itself controlled by a CPU.

All of these items will be described with more details in the coming sections.

3.1.1 Assigner board

The assigner is the brain of the Roland JX. Its purpose is multiple:

1. Scan the keyboard to detect keys which are played and how fast/hard (velocity)
2. Handle the midi interface.
3. Scan and control the human interface: buttons, LEDs, sliders.
4. Control a 40-characters fluorescent display.

A state diagram of the assigner is shown at Figure 3.2.

A quick note about the midi interface. Midi is the standard interface used by musical instruments to communicate. It is basically a serial interface running at 31250 bauds. The difference with a regular serial port is that midi uses opto-couplers to provide electrical isolation between instruments and therefore avoid ground loops known to create hum. See Figure 3.3.

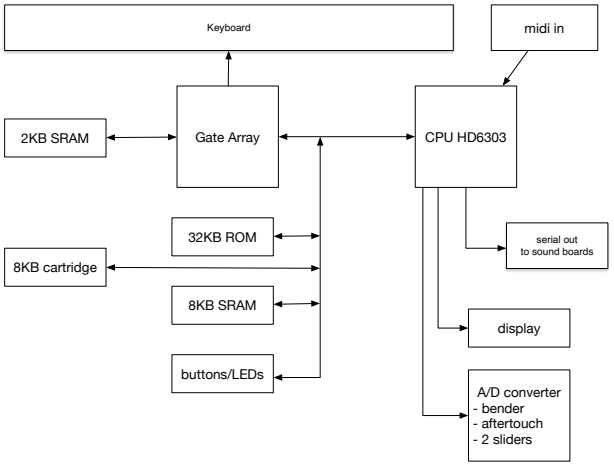


Figure 3.2: Assigner

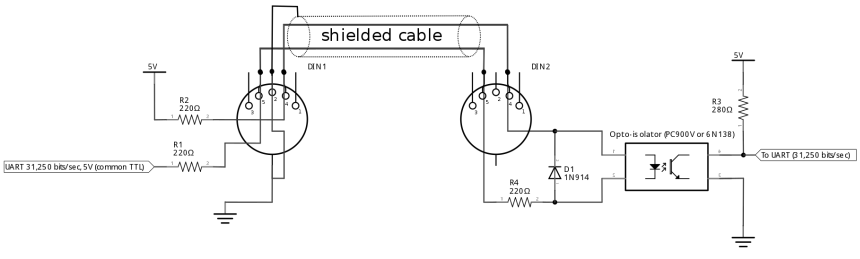


Figure 3.3: Midi hardware interface

3.1.1.1 Hardware

The 8-bits CPU runs at 2 MHz and has an address bus of 16 bits, allowing for a 64KB address space. This address space is divided between RAM, ROM and the gate-array which provides a shared ram between itself and the CPU. This shared ram is used to place information about the keyboard scan. When a key is pressed on the keyboard, the gate-array writes the velocity at a fixed offset and interrupts the CPU which can then retrieve the data by reading the shared ram. Note that this is implemented using a weird scheme: the gate-array controls the clock going into the CPU so it can stale the cpu to prevent it from generating bus traffic to the shared ram. Buttons and LEDs are memory-mapped and directly controlled by the CPU. The code fits in a 32KB eprom but is actually less than 24KB since the eprom contains a bank of factory sounds which takes about 8KB.

The synthesizer has a 40 characters display. This display board has its own controller which receives data from the CPU using a bit-banged spi[2] protocol which only requires 2 signals (clock and data).

Since the synthesizer has a few analog inputs, an A/D converter chip is added connected through another bit-banged spi interface to the CPU. The A/D converter provides 8-bits sampling of 4 analog signals:

- the bender : a pitch wheel to vary the pitch in a continuously variable manner. The first bender with a pitch wheel was the Minimoog in 1971.
- the aftertouch : on electronic synthesizers, pressure sensitivity is called aftertouch. It allows keyboard players to change the tone or sound of a note after it is struck, the way that singers, wind players or bowed instrument players can do.
- two sliders which can modify any parameter in real time. Software simply maps the A/D outputs to defined parameters.

The assigner controls the sound boards. It must also generates midi output since the keyboard can control other synthesizers. It is also useful to have midi in/out to load and save sounds (through midi sysex[3]). Unfortunately the CPU only has one serial interface. Its receive side is used for midi in and Roland engineers have decided to use the transmit side to control the sound boards. The communication is unidirectional. Ultimately, midi out must be provided somehow : without any serial interface left, the solution implemented by Roland is to pass midi out information to a sound board which will then control the midi out signal. The advantage of this approach is money-saving : there is no need for additional hardware. The disadvantage is increased latency when outputting midi and extra burden on the sound board CPU since it gets interrupted to process serial traffic.

3.1.1.2 Software

If we started with a schematics and a high-level overview to understand the hardware, there was no documentation or specification available for the software. For

[2]Serial Peripheral Interface
[3]System Exclusive messages

the software, we started by dumping the eprom in order to disassemble the code. Since our goal is to write a simulator of the sound generators, we don't need to fully understand what the assigner is doing. Instead we will be happy to reverse-engineer how parameters are mapped between the user interface (where most parameters are displayed as 7-bits integers).

The HD6303 is a CPU built by Hitachi but it is based on the Motorola 6800 processor with some extra instructions. It has two 8-bits registers A/B and a 16-bits address register X. From today's standards, it is a very simple CPU. Disassembling the assigner code was straightforward. After configuring the hardware, the CPU enters a main loop :

1. scan the buttons
2. refresh the LEDs
3. update the display
4. read the A/D converter
5. assign notes

Assigning notes is the main job's of the assigner. When a note is played (through the keyboard or incoming midi message), the assigner must decide which sound board will play the note and which of the 6 voices will be used. Various algorithms are used depending on the **assign patch** parameter and the **patch mode** parameter. Indeed the synthesizer can stack 2 tones together in dual mode, play all voices in mono mode, etc... Voices can be assign in FIFO or LIFO order. These details, while interesting, are not relevant for this project.

Amongst the assigner tasks, the one that interests us the most is how tones are configured on the sound boards. We will see in the next section the parameters required for a tone. In the assigner, each tone is stored in memory has a blob/struct of bytes. Each parameter has an associated number used to transmit the parameter to the sound board. Therefore the protocol is very simple : to transmit a parameter, the assigner first sends its number followed by the parameter value. Since each parameter is a number between 0 and 127, parameter numbers are chosen to be bigger than 127 which allows easy synchronization for the sound board. In the event of a corrupted byte during transfer (remember that the communication assigner to sound board is unidirectional), the sound board can detect the condition very easily. The list of the parameter numbers is given in Appendix A.

3.1.2 Sound board

We started again from the schematics, a state diagram of a sound board is shown at Figure 3.4.

3.1.2.1 Hardware

The sound board is a subtle mix of digital and analog electronics. The digital part is based on an Intel 8031 microcontroller, one of the earliest microcontroller

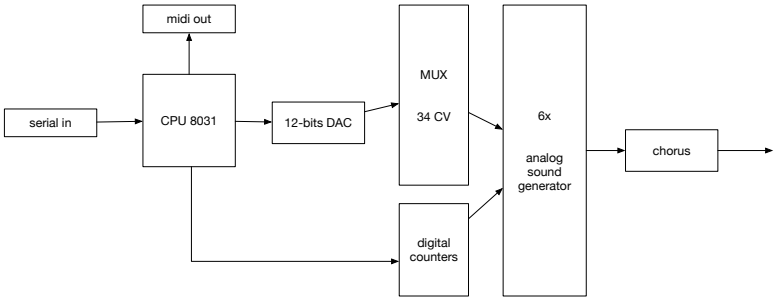


Figure 3.4: Sound board

in the market. It has 128 bytes of internal memory but another Intel chip (a 8155) provides 256 extra memory bytes.

The 8031 microcontroller is a very simple chip which provides:

- a 8051 instruction set
- 2 16-bits timers
- 1 full-duplex serial port
- up to 4 8-bits ports (32 I/O lines)
- 128 bytes of RAM

As stated previously, the serial port is used as the primary communication mechanism. The 8031 receives orders from the the assigner on the RX side. The TX side is used to drive the midi out port.

The sound board hardware can generate up to 6 voices. In the JX terminology, each voice plays a **tone** and is made of a set of parameters. Most parameters are unique to a voice while some parameters of a tone can affect the 6 voices at once. The flow of how a tone is generated is shown at Figure 3.9.

DCO The DCO uses a clever mechanism to generate a sawtooth of a given frequency, starting from a 8 MHz square wave produced by a crystal. Figure 3.5 shows a simplified hardware view. At the input, we have the high-frequency signal. The 8031 CPU computes a divider value which is programmed into the counters[4]. At the output of the counters, we find a square wave of the wanted frequency.

The sawtooth waveform is generated by charging a capacitor linearly in an op-amp integrator configuration. A rising edge of the square waveform discharges the capacitor and resets the sawtooth. While this generates a sawtooth of the perfect frequency, the level of the sawtooth is not constant since the less time the capacitor charges, the less the output amplitude. To circumvent this issue, the input of the integrator is a voltage controlled by the CPU. This voltage is computed based on the wanted frequency and some calibration value computed at boot time[5]. The

[4]The counters are programmable counter chips, 8253.

[5]Capacitors typically have a precision of 10-20% so a calibration is required.

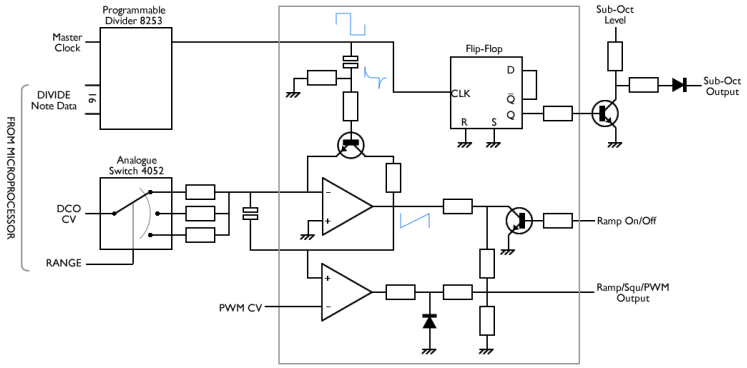


Figure 3.5: DCO waveshaper

result is a sawtooth amplitude nearly constant accross the whole frequency range.

While this method is clever, it comes with a drawback: since the output frequency is based on a digital divider, it cannot produce a smooth variable frequency. Instead it can only produce various discrete frequencies, depending on the value of the divider. When a modulation is applied to the DCO, the frequency actually steps through the various values of the divider under control of the CPU. If these steps are not close enough, the result will be audible. We did some calculation to find how bad this could be. If the master clock is 8 MHz, we must divide by 8000 to produce 1 kHz. Dividing by 4000 will produce 2 kHz which is an octave higher. We see that the higher the frequency, the less steps we have in each octave. While it may look bad, in practice the error is much smaller than 1% and the human ear can only detect errors bigger than 5%. At the bottom of the midi frequency range, another problem happens: a 16-bits divider is no longer enough to produce the wanted low frequencies, therefore Roland engineers added a prescaler to the master clock input of the counters. This prescaler can pass the 8 MHz clock through or divide it by 2, 4 or 8 [6].

There are two DCOs per voice in the JX architecture. Each DCO can generate a sawtooth, a square wave or white noise. The second oscillator can be synchronized to the first one or both oscillators can synchronized each other (refered as *cross-synced*). This is simply done by using the square wave of one oscillator to reset the sawtooth generator of the other oscillator.

Finally, it should be mentioned that there is a detune factor that can be applied to the second DCO. In the 8155 chip, Roland had a 8-bits timer left unused. They use it to slightly change the clock going into the counters of the second oscillator. The result obtained by detuning the second oscillator is a warmer sound containing more harmonics.

The circuit diagram of a single voice generator is shown at Figure 3.6.

VCF and VCA Voltage controlled filters were invented by Roger Moore in the sixties and consisted of transistors ladders as shown in Figure 3.7. It is outside the scope of this thesis to develop this VCF. However, we can note that constructing

[6] This is the `DCO RANGE` tone parameter.

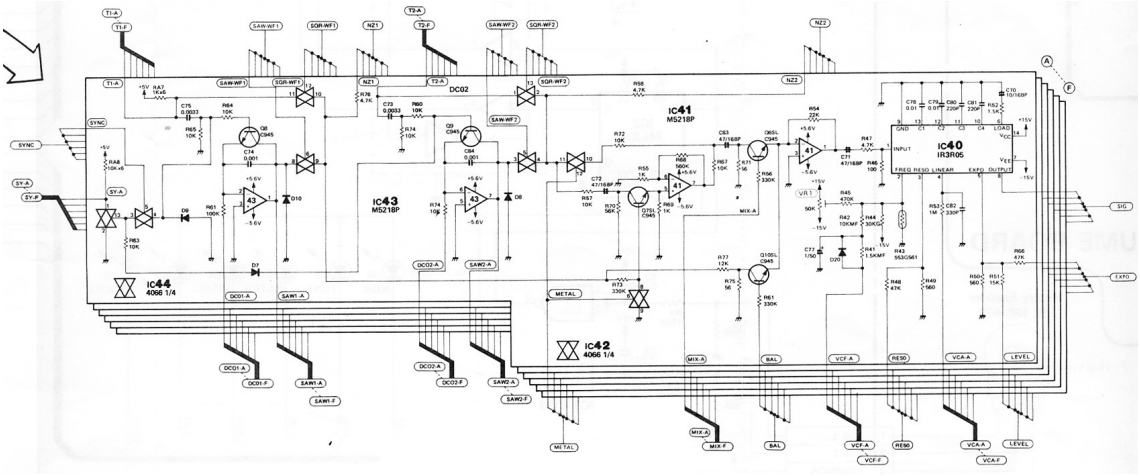


Figure 3.6: Roland JX voice

such filter with discrete transistors is difficult because of different characteristics of transistors (based on the transistor batch, etc...) To avoid this issue, transistors can be manually matched. Another solution is to design a custom chip where internal transistors are matched by design and this is the solution that Roland has taken: the VCF and the VCA are both implemented in a custom chip which just requires a few external components. The 14-pins chip contains two 2-poles state variable filters configured as a 4-poles low pass filter. The chip has external capacitors with two internal VCA: one for voltage controlled resonance and one as the final VCA in the signal chain. What matters for us is the -24dB/octave frequency response of the filter as well as its resonant capability.

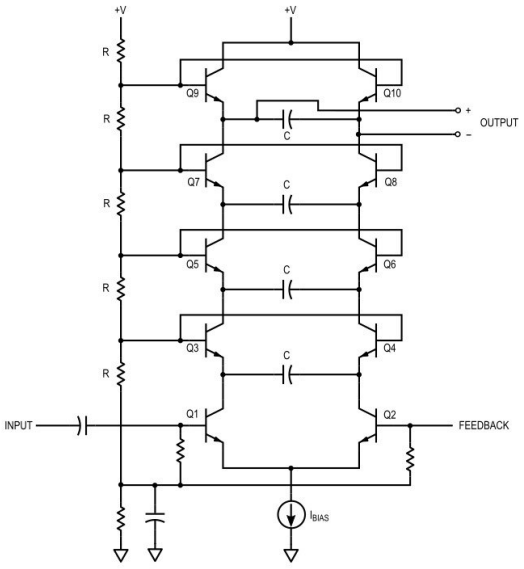


Figure 3.7: Moog VCF

Roland has also added a high-pass filter in the signal chain but it is common to all voices.

Chorus In music, a chorus effect occurs when individual sounds with approximately the same time and similar pitches, converge and are perceived as one. The effect is implemented using a Bucket-bridage chip (BBD). A BBD is a discrete-time analog delay line and consists of a series of capacitance sections C_0 to C_n where n is typically 512, 1024 or 2048. The stored analog signal is stored along the line of capacitors and moves step by step at each clock cycle[7]. The frequency of the clock driving the BBD is itself modulated, producing the desired chorus effect. A circuit diagram of a BBD is shown at Figure 3.8.

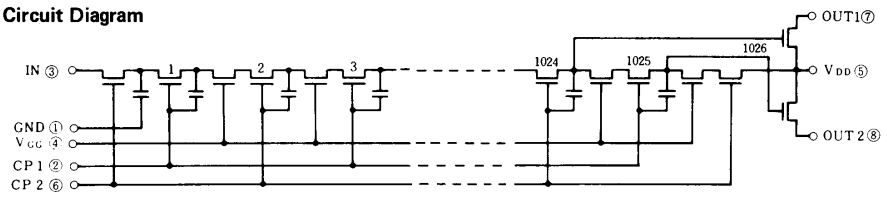


Figure 3.8: Bucket Brigade Device

3.1.2.2 Software

The 8031 microcontroller is responsible for controlling the hardware generating the 6 voices. In order to build an accurate simulator, this code had to be fully understood so we went again through disassembly. Fortunately, the code is small (8KB) and the memory is very limited (384 bytes).

The code can be divided into 2 parts :

1. The main loop which refreshes the CV of the 6 voices, one by one.
2. The interrupt code : triggered when a byte is received or has been transmitted on the serial port. In receive mode, the software is in either one of these two states:
   - waiting for tone parameters or voice control commands
   - forward incoming byte to midi out

Let's describe the simple interrupt code first. In midi forward mode, the 8031 simply takes a byte received from serial and places it in the transmit circular buffer. Once a byte is transmitted, an interrupt is generated and the eventual next byte can be sent. In regular mode, the code receives tone parameters or voice commands. Tone parameters, as explained in Section 3.1.1.2, are made of a byte greater than `0x80` followed by a byte smaller. Voice control commands are also bytes higher than `0x80` making protocol decoding extremely simple. Voice control commands are used to start and stop generating a sound on a given voice, turn the bender effect on/off or select a board. The values are given in Appendix A.

[7]The name comes from analogy with the term bucket brigade used for a line of people passing buckets of water

The main loop (see Algorithm 1) is made of 6 copies of the same code used to control each voice. While a modern software developer would use a subroutine to do this, we have to remember that real time (or close to real time) control of each voice is required. Furthermore, the 8031 CPU doesn't have many registers and it has a single indirect pointer register. Therefore copying the code made a lot of sense. By simulating the code, it can be seen that 100% of the CPU cycles are consumed, resulting in about 10 ms to refresh each voice. To keep the processing time of a voice constant, every conditional instruction is coded such that taking any branch results in the same processing time[8]. We have a hybrid digital/analog implementation: the signal path is fully analog but some of the analog controls come from voltage generated by a DAC and computed by the 8031 CPU. For example, the envelopes are computed, applied (by computation) to the parameters controlling the analog components and finally converted to analog voltage fed to the related chips.

Algorithm 1: 8031 algorithm

```
Initialize hardware
Calibrate DCOs
while true do
    update LFO
    update common parameters
    for i ← 1 to 6 do
        program DCO1_i
        program DCO2_i
        update MIXER_i
        update VCF_i
        update VCA_i
if Interrupt then
    if Byte received then
        if Forward mode then
            add byte to transmit buffer
        else
            decode byte and update corresponding parameter
    else if Transmit buffer not empty then
        send byte to midi out
```

[8]This is actually hard to achieve. The programmer must known the number of cycles taken by each instruction in a branch and eventually add `nop` instructions in the other branch to match the number of cycles in each branch. Obviously when there are nested conditionals, the complexity of this exercise increases.

3.1.2.3 Parameters

Once the code has been disassembled, finding all parameters contributing to a tone (and how they contribute) is relatively easy. Table 3.1 gives a summary of all parameters and what they do. Figure 3.9 provides a block diagram view of a tone generator.

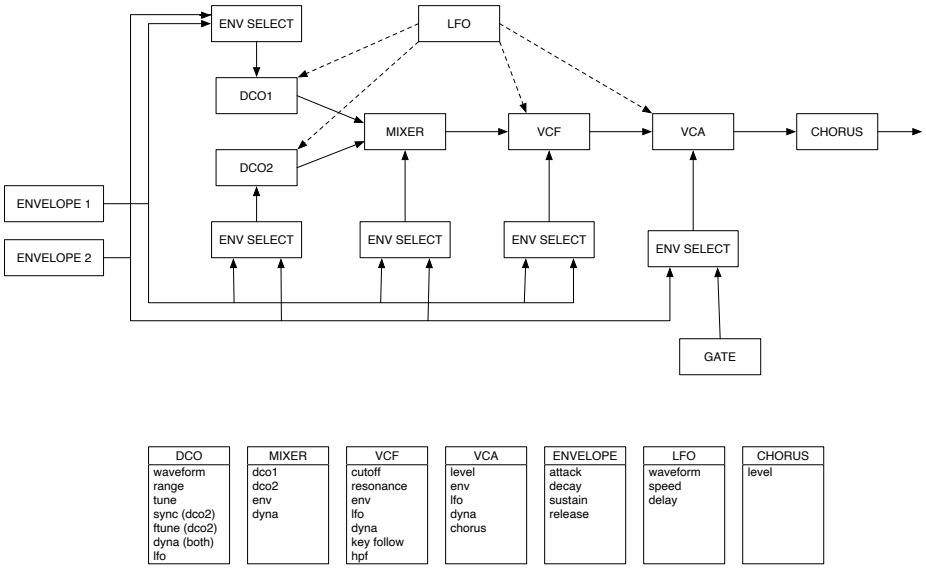


Figure 3.9: JX Tone flow

The signal path is as follow :

1. Two oscillators (DCO1 and DCO2) generate waveforms.
2. A mixer combines the two oscillator outputs into a single signal.
3. A low-pass filter (VCF) removes high frequencies of the signal. Some frequencies near the cutoff can be amplified (resonant filter).
4. A high-pass filter (HPF) eventually removes low frequencies of the signal.
5. An amplifier (VCA) boosts the signal level.
6. Eventually a chorus effect is added.

Most of these building blocks can be affected by one or more of these:

- An envelope which dictates over time the amount of effects applied to the sound currently generated. This effect can be fixed or the envelope can itself be affected by the player's velocity.
- The low frequency oscillator (LFO) can modulate the effect, eventually after a certain delay.

Parameter	**Values**	**Description**
DCO1 WF	0-3	DCO1 waveform : sawtooth, square, pulse or noise
DCO1 RANGE	0-3	DCO1 octave. Shift -2, -1, 0 or 1 octaves
DCO1 TUNE	0-25	DCO1 pitch in step of semi-tones (-12..0..+12)
DCO1 ENV	0-127	DCO1 envelope. Control by how much the envelope controls the pitch
DCO1 LFO	0-127	Amount of LFO to apply to DCO1
DCO2 WF	0-3	DCO2 waveform, same as DCO1
DCO2 RANGE	0-3	DCO2 range (same as DCO1)
DCO2 TUNE	0-25	DCO2 pitch (same as DCO1)
DCO2 ENV	0-127	DCO2 env (same as DCO1)
DCO2 LFO	0-127	DCO2 LFO (same as DCO1)
DCO2 FTUNE	0-127	fine detune of DCO2
DCO XMOD	0-3	DCO synchronization: none, DCO1 $\rightarrow$ DCO2, DCO1 $\leftarrow$ DCO2, DCO1 $\leftrightarrow$ DCO2
DCO DYNA	0-3	Dynamic control of the DCOs : when the DCOs pitch are controlled by the envelope, the amount of envelope is affected by velocity
DCO MODE	0-3	Select which envelope (1 or 2) and envelope polarity is applied to the DCOs
MIX DCO1	0-127	Amount of DCO1 in the final tone
MIX DCO2	0-127	Amount of DCO2 in the final tone
MIX ENV	0-127	Control how much the envelope affects the mixer
MIX DYNA	0-4	Dynamic control of the mixer. Velocity affects the behavior of the mixer
MIX MODE	0-3	Select which envelope and polarity is used for the mixer
VCF FREQ	0-127	Cutoff frequency of the filter
VCF RES	0-127	Amount of resonance of the filter
VCF LFO	0-127	Amount of LFO applied to the filter cutoff
VCF KEY	0-127	Control the shift of the cutoff position.
VCF DYNA	0-3	Control how dynamics (player velocity) affects the filter
HPF FREQ	0-3	Select one of the 4 cutoff frequencies of the high-pass filter (0=no filtering)
VCA LEVEL	0-127	Control the overall volume of the tone

VCA MODE	0-1	Select ENV2 to control the VCA, or simply an on/off envelope form
VCA DYNA	0-3	Control how dynamics affects the VCA
CHORUS	0-3	Control the chorus. Can be off or 2 values
LFO WF	0-2	LFO waveform : sine, square or random
LFO DELAY	0-127	Control when the LFO starts after a key is pressed
LFO RATE	0-127	Control the frequency of the LFO
ENV1 ATT	0-127	Envelope 1 attack time
ENV1 DECY	0-127	Envelope 1 decay time
ENV1 SUS	0-127	Envelope 1 sustain level
ENV1 REL	0-127	Envelope 1 release time
ENV1 KEY	0-3	Envelope key follow. Changes the time required for an envelope curve to complete. The higher the value, the shorter time higher keys will be.
ENV2 ATT	0-127	Envelope 2 attack time
ENV2 DECY	0-127	Envelope 2 decay time
ENV2 SUS	0-127	Envelope 2 sustain level
ENV2 REL	0-127	Envelope 2 release time
ENV2 KEY	0-3	Envelope key follow (same as ENV1)

Table 3.1: All tone parameters

Note that internally each parameter is coded on 7-bits. If the parameter can only take a few values, only the most significant bits are used[9].

[9]This allows some code simplification

Chapter 4

Synthesizer simulator

Armed with a deep knowledge of the hardware of the Super JX, we are now able to develop a simulator capable of generating sounds similar to the analog synthesizer. By *similar* we don't mean exact nor even close to the real sounds. Indeed, simulating analog synthesizers is extremely complex [18]. Even today, synthesizer manufacturers try to approximate the sounds generated by their old devices but fall pretty short. The warmth and complex sounds generated by these devices remain unmatched. However, for our project, an approximation of the waveforms generated will be plenty.

4.1 Simplification

As we have seen in Table 3.1, a tone is made of 42 parameters. Several of these parameters don't apply in a static configuration. If we define a static configuration as a sound which is generated at a given frequency, velocity and duration, then some parameters while affecting the tone, won't be visible to a human or neural network. These parameters are:

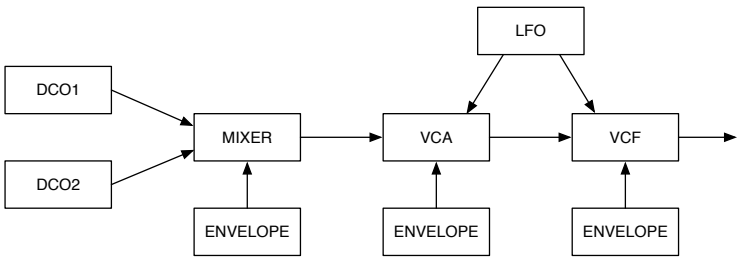
- All *key follow* parameters. These parameters control how the position of the key played matters. An effect will affect a high frequency sound deeper than a low frequency sound. Without listening to several sounds of different frequencies, these parameters are impossible to guess.
- All *dynamic* parameters. Here, the velocity of played keys affects the effect applied. In order to find out the value of such parameters, the same note should be played at different velocities.

Since we limit our research of static sound recognition, we will ignore these parameters.

Without losing any generality, we can also get rid of the envelope select parameters. Indeed, the components affected by envelope have a parameter to select which envelope should be applied. Typically in most sounds, one envelope is used for the filter and the other envelope is used for the amplifier and/or the mixer. In our simulator, we have decided to have dedicated envelope per component (mixer, filter, amplifier).

To keep the number of parameters down, we have further simplified the simulator by removing parameters which are rarely used in common sounds. For example, applying an envelope to the oscillators is uncommon since this produces a sound with varying pitch[1].

We end up with the diagram shown at Figure 4.1. We are down to 33 parameters.



DCO	MIXER	VCF	VCA	ENVELOPE	LFO
waveform	dco1	cutoff	level	attack	waveform
range	dco2	resonance	env	decay	speed
tune	env	env	lfo	sustain	delay
sync (dco2)		lfo		release	
ftune (dco2)					

Figure 4.1: Simulator tone flow

We will now describe the building blocks of the simulator.

4.2 Oscillator

Our oscillator must produce 3 waveforms : sawtooth, square and noise.

The sawtooth is produced using Equation 4.1 where A is the amplitude, T the period of the wave, ϕ its phase and frac(x) the fractional part. The left figure in Table 4.1 shows the sawtooth generated by the fractional method and the Fourier series with 4 and 10 terms.

$$S(x) = A \text{ frac}(\frac{x}{T} + \phi) \tag{4.1}$$

$$\text{frac}(x) = x - \lfloor x \rfloor \tag{4.2}$$

If $\phi = 0$, $A = 1$ and $T = 2L$, the Fourier series is given by

$$f(x) = \frac{1}{2} - \frac{1}{\pi} \sum_{n=1}^{\infty} \frac{1}{n} \sin\left(\frac{n\pi x}{L}\right) \tag{4.3}$$

[1]This is great to generate sound effects!

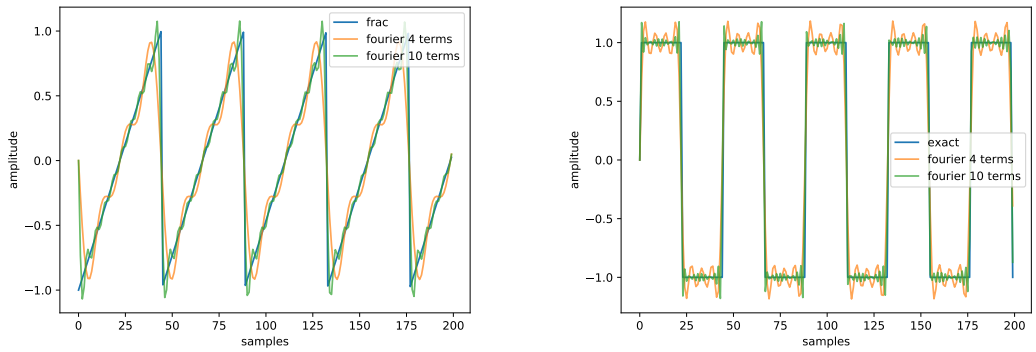


Table 4.1: Oscillator waveforms

The square wave is produced using Equation 4.4.

$$S(x) = A(-1)^{2x/T} = A \text{ sgn}\left[\sin\left(\frac{2\pi x}{T}\right)\right] \tag{4.4}$$

Its Fourier transform is given by

$$f(x) = \frac{4}{\pi} \sum_{n=1,3,5,\ldots}^{\infty} \frac{1}{n} \sin\left(\frac{n\pi x}{L}\right) \tag{4.5}$$

Finally we have the white noise where the Fourier transform is flat.

4.3 Mixer

The mixer takes the two signals coming from the oscillators and mix them together. The amount of one signal vs the other is controlled by the mixer level (one per oscillator) as well as the envelope amount applied. The computation is done using the following equation:

$$S(x) = (1 - env_{level})(\hat{y}_1 + \hat{y}_2) + env_{level}(env(x) * \hat{y}_1 + (1 - env)\hat{y}_2) \tag{4.6}$$

where $\hat{y}_1 = mix_1 y_1$, $\hat{y}_2 = mix_2 y_2$, env_{level} is the amount of envelope applied and $env(x)$ is the actual envelope value computed at time x.

A simple example, without envelope, is shown at Figure 4.2.

4.4 Amplifier

The amplifier is trivial to implement in the digital domain: we just multiply the samples value by a factor. Of course, like most parameters in a synth, this factor is not constant across the sound duration but depends on an envelope effect.

On real hardware, the filter appears in the signal chain before the amplifier. The reason is that the input level of the filter chip is in the range of 20 mV-30 mV. This small signal is then amplified to a few volts.

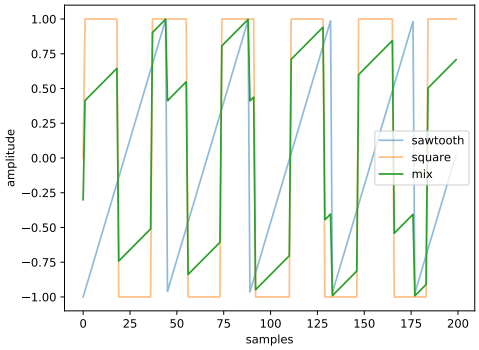


Figure 4.2: Mixer

In the simulator, we don't care about this detail so we can first amplify the signal (in the time domain), then move to the frequency domain and do the filtering there. We don't even need to go back to the time domain because the inputs of our neural networks are spectrograms. We coded the conversion back to the time domain so a human can listen to the generated tones.

4.5 Filter

As mentioned in the JX Section 3.1.2.1, the JX uses a custom chip as the VCF. To reproduce this filter, we will be using second-order lowpass filters [20, 8]. These filters have a transfer function

$$H(s) = \frac{\Omega_n^2}{s^2 + 2\zeta\Omega_n s + \Omega_n^2} \tag{4.7}$$

where Ω_n is the undamped natural frequency and ζ is the damping ratio. Using quadratic formula, we can solve this equation and find the two poles of $H(s)$:

$$s_{1,2} = -\zeta\Omega_n \pm \Omega_n\sqrt{\zeta^2 - 1} \tag{4.8}$$

There are 3 sets of poles that can be characterized as follows :

- **Overdamped**. The poles are real and distinct. This occurs if $\zeta > 1$.
- **Critically damped**. This corresponds to $\zeta = 1$. The two poles are repeated with $s_1 = s_2 = -\zeta\Omega_n = -\Omega_n$
- **Underdamped** when $0 < \zeta < 1$, giving a pair of complex conjugate poles. (If $\zeta = 0$ the system is unstable.)

The frequency response of the 2nd order lowpass filter can be found using the substitution $H(j\Omega) = H(s)|_{s=j\Omega}$ giving

$$H(j\Omega) = \frac{\Omega_n^2}{\Omega_n^2 - \Omega^2 + j2\zeta\Omega_n\Omega} \tag{4.9}$$

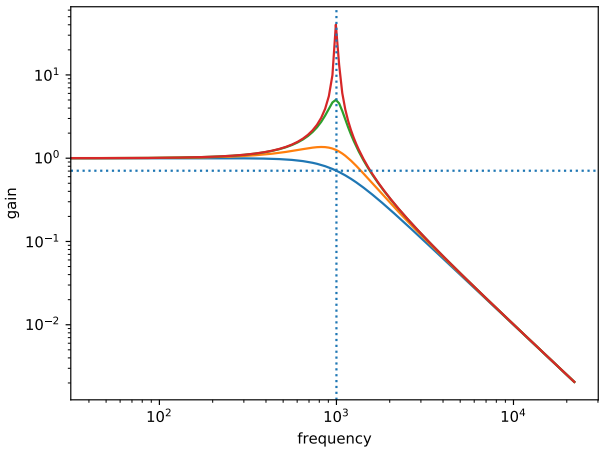


Figure 4.3: Lowpass filter

The critically damped filter represents a filter where no resonance is applied. Applying resonance to the filter is done by underdamping it, resulting in the curves shown at Figure 4.3 where 4 filters are shown with a cutoff frequency of 1000 Hz. The blue curve is critically damped while the orange, green and red are underdamped by different factors.

4.5.1 Short-time Fourier transform

The implementation of this filter is done in the frequency domain where filtering is simply done by multiplication. The conversion from time domain to frequency domain is done using short-time Fourier transform [1].

The short-time Fourier transform (STFT) is a Fourier-related transform used to determine the sinusoidal frequency and phase content of local sections of a signal as it changes over time. To obtain this, the signal is divided into short segments of equal length. The Fourier transform is computed separately on each segment. The concatenation of these Fourier transforms shows the changing spectra as a function of time and is known as a *spectrogram* plot.

Mathematically, we can define the STFT as

$$\mathrm{STFT}\{x(t)\}(\tau,\omega) \equiv X(\tau,\omega) = \int_{-\infty}^{\infty} x(t)w(t-\tau)e^{-j\omega t}dt \tag{4.10}$$

where $w(\tau)$ is a window function (commonly a Hann or Gaussian window) centered around zero, and $x(t)$ is the signal to be transformed. $X(\tau,\omega)$ is the Fourier transform of $x(t)w(t-\tau)$, a complex function representing the phase and magnitude of the signal over time and frequency. Phase unwrapping is employed along either or both time axis τ and/or the frequency axis ω to suppress any jump discountinuity of the phase result of the STFT.

In our simulator, we use tensorflow's stft implementation `tf.signal.stft` with a Hann window `tf.signal.hann_window`.

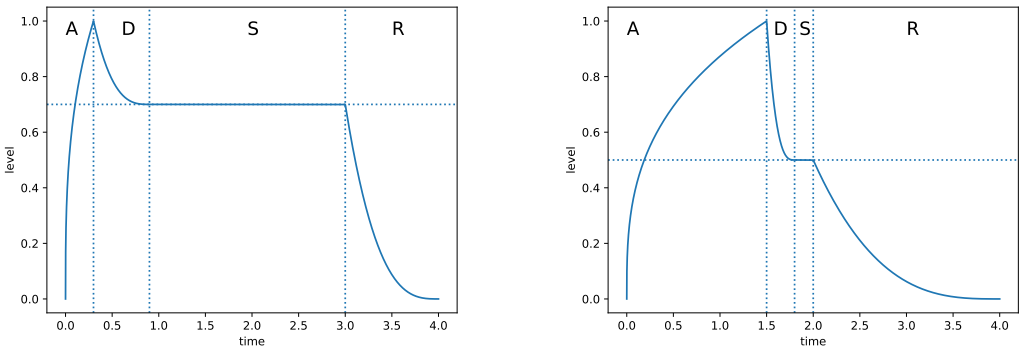


Table 4.2: ADSR envelopes

4.6 Envelope generator

Several envelope formats have appeared in the synthesizer industry [13]. Today, synthesizers allow to draw any envelope shape but at the time where analog synthesizers ruled, generating an arbitrary signal shape was difficult to do (and expensive). Therefore the most common envelope format was the ADSR envelope. ADSR stands for Attack, Decay, Sustain and Release. Attack, decay and release are expressed in time unit, while sustain is a level. The **attack** is defined as the time it takes for the sound to reach its maximum amplitude. **Decay** is the time it takes for the sound amplitude to decrease to the **sustain** level. When the key is released, **release** is the time it takes for the sound amplitude to vanish.

If we denote x_a the attack time in seconds, x_d the decay time, y_s the sustain level and x_r the release time, we can model an ADSR envelope using Equation 4.11.

$$\begin{aligned} 0 < x \leq x_a &: f\left(\frac{1}{x_a}x\right) \\ x_a < x \leq x_a + x_d &: g\left(\frac{(y_s - 1)}{x_d}(x - x_a) + 1, y_s, 1\right) \\ x_a + x_d < x \leq x_a + x_d + x_s + x_r &: g\left(-\frac{y_s}{x_r}(x - (x_a + x_d + x_s)) + y_s, 0, y_s\right) \\ x > x_a + x_d + x_s + x_r &: 0 \end{aligned} \tag{4.11}$$

where f and g are defined by

$$\begin{aligned} f(x) &= \sqrt[3]{x} \\ g(x, y, z) &= \left(\frac{x - y}{z - y}\right)^3 (z - y) + y \end{aligned} \tag{4.12}$$

Two examples of envelopes are given at Table 4.2.

4.7 Dataset

Initially, we intended to use a sound board from a Roland JX synthesizer. Indeed, during the reverse engineering process, we had a setup with a standalone sound board and a microcontroller to control it. This is depicted at Figure 4.4. While it looks very attractive (it requires simply one USB connection where parameters are fed and samples are collected), this setup has one major drawback: it only works in real time. The algorithms that we are trying to use either require a large dataset or generation of samples along the learning process. A quick computation shows that a dataset of 100.000 samples of 4 seconds will require about 5 days to be generated. This is still practical since the dataset can be generated once for all, but if generating samples must occur during learning, it simply renders the generator impractical.

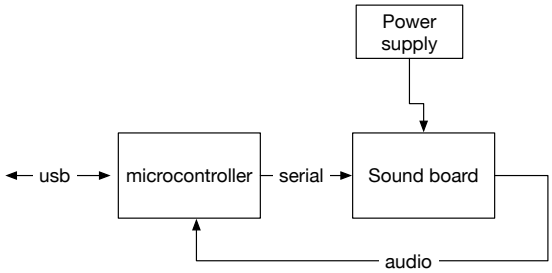


Figure 4.4: Standalone setup

That's why we have developed the simulator. It is written in python using `tensorflow` and `numpy`[2].

Our simulator (see figure 4.1) takes 33 input parameters. At first, we generated sets of parameters randomly using uniform distributions for each parameter. This resulted in a dataset of sounds where almost none of them could be usable to make music. Indeed, from the large input space (about 128^{33}) only a very small subset will produce usable sounds. In order to generate a dataset with useful samples, we use the following approach :

1. Extract tone parameters from banks of sounds of our Roland JX
2. Add code to convert Roland JX parameters to our simulator (and vice-versa)
3. Analyze the distribution of the parameters
4. Generate samples from this distribution

[2]Mixing them proved to be a mistake and had a significant impact on performance

We found about 500 sound parameters on the internet and we compiled them in a small dataset. As we can see in Table 4.3, the parameters are not evenly distributed. Furthermore, we know that the parameters are correlated. This correlation is shown at Figure 4.5 by a covariance matrix. We made a few extra simplifications:

- Remove the envelope of the mixer. Almost no sound made use of it.
- For the same reason, remove the LFO parameter from the mixer and the amplifier.
- Always apply the envelope to the amplifier.

The idea behind these simplifications is to lower the number of parameters and we ended up with 26 parameters.

Equipped with the means and covariance matrix of real sound parameters, we can sample new sets of parameters from the multivariate normal distribution. Parameters falling outside their allowed range are clipped. We generated 3 datasets: a small (1000 samples), medium (100.000) samples and large (1.000.000 samples).

4.7.1 Dataset hyper-parameters

- Sample frequency is $fs = 44.1\,\text{kHz}$
- Each sample is 4 seconds long
- The Fourier transform windows are 512 samples long and overlap half of the previous window.
- The tones are generated at 440 Hz
- Spectrograms are clipped at $fs/16$ without loss of information. This of course would have to be revisited if the generated tones have a higher frequency than 440 Hz. Spectrograms have therefore a $(690, 16)$ size[3][4].
- Each dataset consists of $(\boldsymbol{p}, \boldsymbol{s})$ pairs where $\boldsymbol{p} \in [0,1]^{26}$ are the simulator parameters and $\boldsymbol{s} \in \mathbb{R}^{690}\text{x}\mathbb{R}^{16}$ is the spectrogram.

4.7.2 Sample outputs

Table 4.4 shows a few spectrograms generated by the simulator. Some parameters can be seen easily: the top-left spectrogram, for example, has its amplifier with an envelope with no sustain (the sound fades before the end of the 4 seconds) and it stops brutally, so the release parameter of the amplifier envelope is small. The second spectrogram has not a lot of frequencies so the cutoff is particularly high. On the opposite, the last spectrogram (bottom right) has varying peaks in frequencies, so the cutoff parameter is most likely affected by the LFO or the envelope.

[3] $44100 * 4/256 \approx 690$

[4] the FFT produces $512/2 = 256$ bins but since we clip at $fs/16$, we get 16 bins

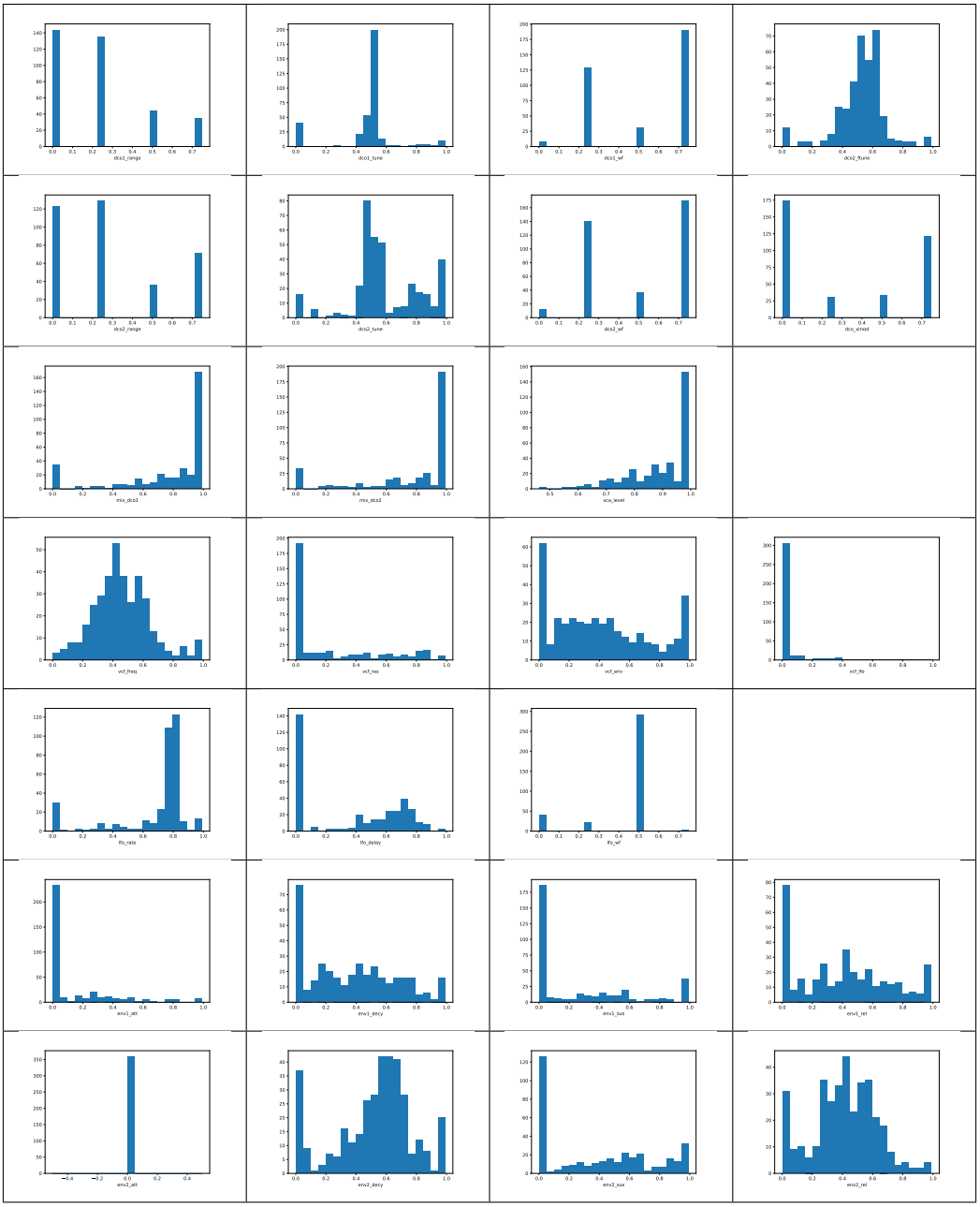


Table 4.3: Roland JX parameters distributions

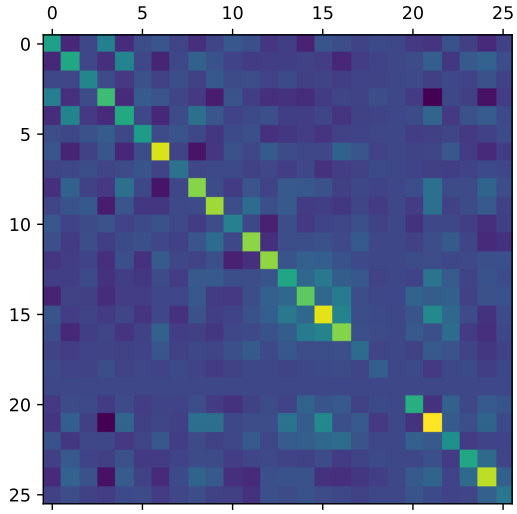


Figure 4.5: Parameters covariance matrix

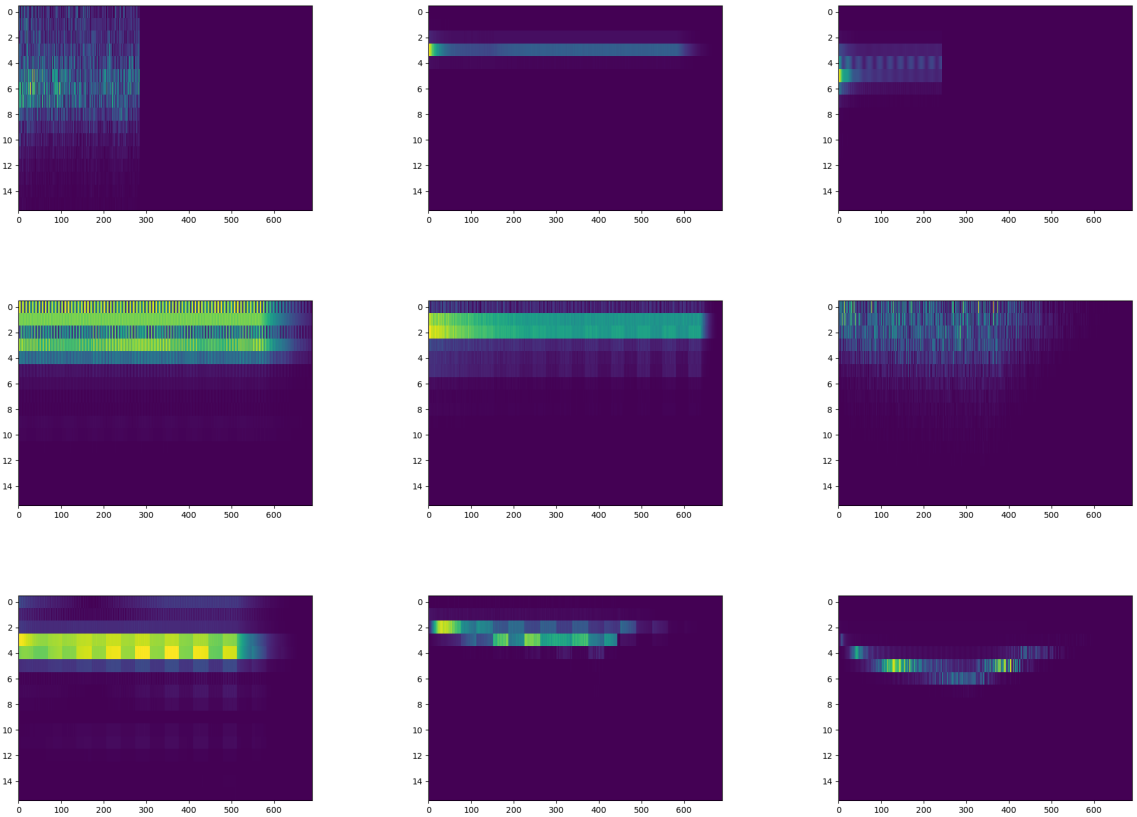


Table 4.4: A few spectrograms

Chapter 5

Deep learning approach

5.1 Introduction

In the last decade, advances in deep learning have raised the state of the art results in various fields, such as computer vision [14], speech recognition [10], natural language understanding [6] and many other fields. In these papers, convolutional neural networks (CNN) and recurrent neural networks (RNN) have shown great performance.

Our problem is to estimate synthesizer parameters to reconstruct a source audio signal. We assume that the source audio is generated by the *same* synthesizer with hidden parameters. Learning is supposed to find out these hidden parameters that were used to generate the source signal. Practically, this would be useful for musicians who want to reproduce sounds generated by other artists. By extension, it could also provide a mechanism to generate sound X from synthesizer Y on another synthesizer Z. For now, we will focus on intra-domain where we try to recover hidden parameters from the same synthesizer.

5.2 Algorithms

This section will describe the various methods that we have applied to solve our problem. We started with classical deep learning methods, using standard regression. In this setting, multiple neural architectures have been tested and will be described:

- Convolutional neural networks
- Recurrent neural networks
- Convolutional neural networks with locally connected layers.

As will be detailed in Chapter 6, the parameters obtained with such methods were good. Nevertheless, we also tried some adversarial methods (this time, only based on recurrent neural networks since they provided the best performances in a standard regression setting, as will also be shown in Chapter 6). The adversarial methods which we used and will now describe are the following:

- Adversarial variational optimization (AVO) [16]
- AVO, with a modification inspired by "Synthesizing programs for images using reinforced adversarial learning (SPIRAL)" [9]

The last method we tried was a mix of both classical regression and adversarial methods. Indeed, we use the prediction of a standard regression network to guide the learning of the SPIRAL algorithm.

5.2.1 Classical regression

The goal of regression is to predict real valued output vectors based on real valued input vectors. In our case, predict simulator parameters, based on input spectrograms. Many machine learning algorithms exist for such task, but artificial neural networks have proven to be particularly effective.

Let θ denote the parameters of a neural network, $\mathcal{D}$ be the set of pairs $(\boldsymbol{p}, \boldsymbol{s})$ generated as detailed in Section 4.7 and $f(\cdot;\theta) : \mathcal{R}^{|\boldsymbol{s}|} \rightarrow \mathcal{R}^{|\boldsymbol{p}|}$ be the function represented by the neural network. A neural network is trained with stochastic gradient descent on a loss. In our case, the loss is defined as:

$$\mathcal{L}_\theta^{\mathcal{D}} = \frac{\sum_{(\boldsymbol{p},\boldsymbol{s})\in\mathcal{D}}(f(\boldsymbol{s};\theta) - \boldsymbol{p})^2}{|\mathcal{D}|}$$

The parameters θ are simply updated multiple times on stochastic batches with the following update:

$$\theta \leftarrow \nabla_\theta \mathcal{L}_\theta^{d\sim\mathcal{D}}$$

where $d \sim \mathcal{D}$ denotes a mini-batch sampled in $\mathcal{D}$. Let us now note a potential problem of this loss in our setting. Indeed, as will be discussed further in Chapter 6, all parameters might not have the same effect on the sound produced from a qualitative perspective. That is, a big prediction error for some parameters might not be heard, while a small prediction error for other parameters might change the sound significantly. There is however no way to know beforehand which parameters are more important than others (especially since there is also correlation between them, so depending on the value of one parameter, another might be very important, or not). Note how the loss that we use (standard mean squared error (MSE)) assigns the same weight to each parameters, which might not be optimal from a qualitative sound reproduction perspective.

As is widely known, all architectures of artificial neural networks are not equal. some are better fitted for some problems than others. Let us now detail the three types of architectures that were implemented and tested on our problem.

5.2.1.1 Convolutional neural network

A Convolutional Neural Network (CNN) is a deep learning algorithm which takes an input image (or more generally, a structured input), assign importance (learnable

weights) to various shapes in the image and is able to distinguish one from the other [15].

CNN requires much less pre-processing compared to other methods because they have the ability to learn the characteristics of the image. This comes at the cost of more computation and more parameters in the neural network.

The architecture of a CNN is analogous to the connectivity pattern of neurons in the human brain: individual neurons respond to stimuli only in a restricted region of the visual field (known as the Receptive Field). A collection of such fields overlap to cover the entire visual area.

The convolution layer extracts features from the input image and it preserves the relationship between pixels by learning image features using small squares of input data. The mathematical operation takes two inputs (image matrix) and a filter (called *kernel*):

- Input image of dimension (h, w, d)
- Filter of dimension (f_h, f_w, d)
- Output a matrix of dimension $(h - f_h + 1), (w - f_w + 1), 1$ (with a stride equal to 1)

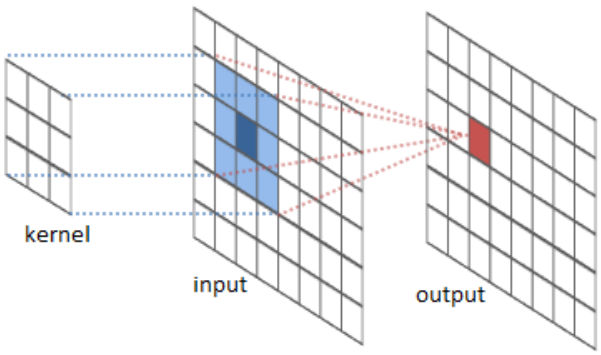


Figure 5.1: Illustration of a kernel

Note that convolving a kernel across an image is an old and well-known technique used in image processing. Various kernels can be used to detect edges, sharpen or blur images, etc.. The beauty of the CNN is that the algorithm will learn the kernels by itself.

A few more quick definitions:

- **Stride** is the number of pixels shifts over the input matrix. When the stride is 1 then we move the filters to 1 pixel at a time. When the stride is 2 then we move the filters to 2 pixels at a time and so on.
- **Padding** is used when the filter doesn't fit the input image (at the borders). The image can be zero-padded or we can drop the part of the image where the filter doesn't fit.

- **Pooling** layers can be seen as a kind of downsampling used to reduce the dimensions of the output while keeping important information. Typical pooling layers can be **max pooling** (keep the maximum element), **average pooling** (compute the average of all elements) and **sum pooling** (compute the sum of all elements).

Deep Convolutional Neural Networks are made of several convolution and pooling layers followed by one or more fully connected layers.

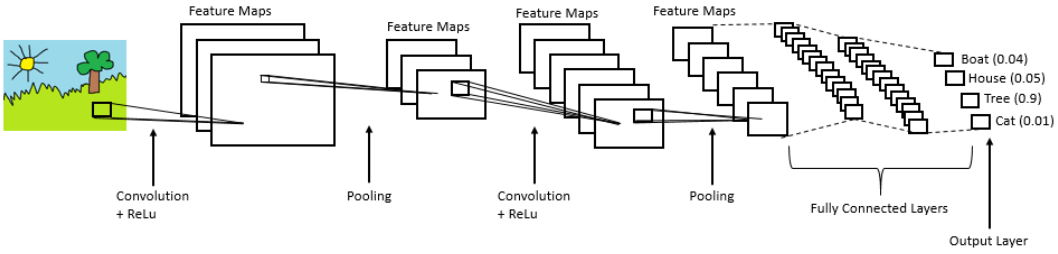


Figure 5.2: Example of CNN

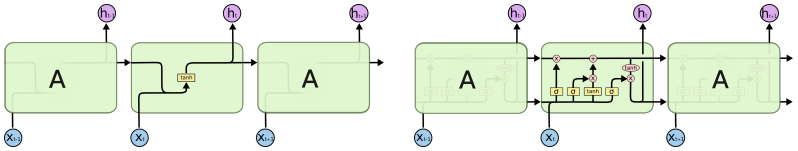
5.2.1.2 Recurrent neural network

Neural networks don't have persistence. Recurrent networks have been widely used in the past years, providing excellent performances on many problems requiring memory such as e.g. sequence to sequence modeling, speech recognition. They look particularly well suited to our problem which is by definition time related. The envelopes, for example, modify the sound shape along the time axis. These achievements often are the result of the development of the long-short term memory (LSTM [11]) and gated recurrent units (GRU [2, 7]) recurrent cells, which allowed RNNs to capture time-dependencies over long horizons [12, 4].

More formally, a time-series can be defined as $\boldsymbol{X} = [\boldsymbol{x}_0, \ldots, \boldsymbol{x}_T]$ with $T \in \mathbb{N}_0$ and $\boldsymbol{x}_i \in \mathbb{R}^n$. In our case, the spectrogram can be cut along the temporal axis (that is, each $\boldsymbol{x}_i$ corresponds to the ith column of the spectrogram $\boldsymbol{s}$). To capture time dependencies, RNNs maintain a recurrent hidden state whose update depends on the previous hidden state and current observation of a time-series, making them dynamical systems and allowing them to handle arbitrarily long sequences of inputs. Mathematically, RNNs maintain a hidden state $\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t; \theta)$, where $\boldsymbol{h}_0$ is a constant. In its most standard form, a recurrent neural network updates its state as follows:

$$\boldsymbol{h}_t = g(U\boldsymbol{x}_t + W\boldsymbol{h}_{t-1}) \tag{5.1}$$

where g is a standard activation function such as a sigmoid or an hyperbolic tangent. However, recurrent neural networks using Equation 5.1 as update rule are known to be difficult to train on long sequences due to vanishing (or, more rarely, exploding) gradient problems. To alleviate this problem, more complex recurrent update rules have been proposed, such as long-short term memory (LSTM [11]) and gated recurrent units (GRU [2]).

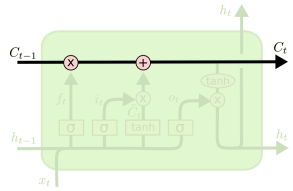


RNN LSTM

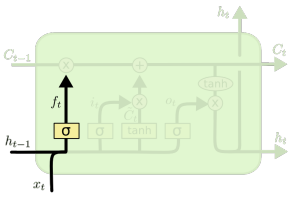
Table 5.1: RNN and LSTM structures

LSTM Network Long Short Term Memory networks are capable of learning long-term dependencies. In a standard RNN, the repeating module has a very simple structure such as a single tanh layer. The LSTM also has a chain structure, but the repeating module has a more complex structure: instead of a single neural network layer, we have four of them. These repeating modules are shown at Table 5.1. Refer to [19] for all details, we will provide a summary below.

Let's walk through the LSTM to understand.

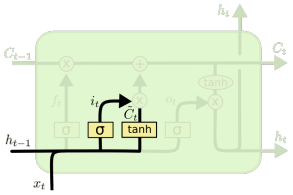


The cell state runs straight down the entire chain with only minor linear interactions. The LSTM has the ability to remove or add information to the cell state. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a point-wise multiplication operation. The sigmoid layer outputs values between 0 and 1, describing how much of each component flows through. A zero value means nothing while one means everything.



The first step is to derive what information will be discarded from the cell state. This decision is made by a sigmoid layer called the *forget gate layer*. It looks at $\boldsymbol{h}_{t-1}$ and $\boldsymbol{x}_t$ and outputs a number between 0 and 1 for each number in the cell state $\boldsymbol{C}_{t-1}$.

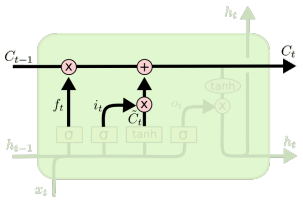
$$\boldsymbol{f}_t = \sigma(U_m \boldsymbol{x}_t + W_f \boldsymbol{h}_{t-1} + \boldsymbol{b}_f) \tag{5.2}$$



The second step is to decide which information the cell will store. This is made of two parts: a sigmoid layer called *input gate layer* decides which value will be updated, and a tanh layer which creates a vector of new candidate values $\tilde{\boldsymbol{C}}_t$ that can be added to the state.

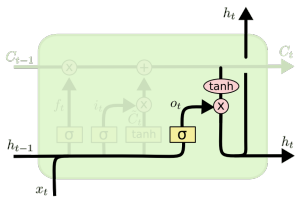
$$\boldsymbol{i}_t = \sigma(U_i \boldsymbol{x}_t + W_i \boldsymbol{h}_{t-1} + \boldsymbol{b}_i) \tag{5.3}$$

$$\tilde{\boldsymbol{C}}_t = \tanh(U_C \boldsymbol{x}_t + W_C \boldsymbol{h}_{t-1} + \boldsymbol{b}_C) \tag{5.4}$$



Steps 1 and 2 decided what to do, so we need to apply their decisions. The old state $\boldsymbol{C}_{t-1}$ is multiplied by f_t (forgetting things). The old candidate $\tilde{\boldsymbol{C}}_{t-1}$ is multiplied by i_t to reflect the magnitude of the update.

$$\boldsymbol{C}_t = f_t \boldsymbol{C}_{t-1} + i_t \tilde{\boldsymbol{C}}_t \tag{5.5}$$



Finally we compute the output. We run the cell state through a sigmoid layer (it decides what parts of the cell state we will output). The values are mapped between $[-1, 1]$ by passing through a tanh.

$$o_t = \sigma(U_o \boldsymbol{x}_t + W_o \boldsymbol{h}_{t-1} + \boldsymbol{b}_o) \tag{5.6}$$

$$\boldsymbol{h}_t = o_t \tanh(\boldsymbol{C}_t) \tag{5.7}$$

5.2.1.3 Locally connected convolutional neural network

We have described convolutional neural network at Section 5.2.1.1. Convolutional layers are technically locally connected layers with shared weights. The same filter is run for all the (x, y) positions in the image. In other words, all the pixel positions share the same filter weights. The neural network tunes the filter weights until convergence. Researchers found out that while this is great for classification, CNN tend to miss subtle nuances of spatial arrangements.

To circumvent this issue, a different approach can be used: we can have a different filter for each (x, y) position, so there is no more a convolution per-se, but rather a dot product at each pixel position. The drawback is an much higher number of parameters since the kernel is no more shared on the whole image but we have many individual kernels.

The locally connected convolution layer is almost identical to a convolutional layer without any sharing of the weights.

5.2.2 Adversarial Variational Optimization

Let us now move towards a completely different learning paradigm than classical regression, Adversarial Variational Optimization (AVO) [16]. In this learning paradigm, there is no need to acquire a train set with $(\boldsymbol{p}, \boldsymbol{s})$ pairs. Rather, the goal of AVO is to find the set of parameters (actually a distribution over such parameters) that fits best a set of samples generated using the simulator. Contrary to classical regression where after training, obtaining a prediction for a set of parameters $\boldsymbol{p}$, corresponding to a spectrogram $\boldsymbol{s}$ is as easy as calling $f(\boldsymbol{s}; \theta)$, the procedure here is more cumbersome and goes as follows. To obtain a distribution over parameters to replicate a sound (or set of sounds) generated using the set of parameters $\boldsymbol{p}$:

- Generate a set of sounds $\mathcal{S}$ using the simulator with the parameters $\boldsymbol{p}$.

- Use AVO to learn a distribution $q(\theta)$ over parameters, which fits best to replicate $\mathcal{S}$.
- Sample $\hat{p}$ from q. Plugging $\hat{p}$ in the simulator should give sounds close to those in $\mathcal{S}$

We note that this procedure must be repeated for each sound (or set of sounds) we want to find the true set of parameters for. This is a big drawback with respect to classical regression where after training, it is very straightforward to obtain new predictions. However, there is also a huge advantage to this technique. With AVO there is no groundtruth on the set of parameters to predict. Rather the algorithm receives a set of sounds $\mathcal{S}$ and is only asked to generate a distribution over parameters, which when sampled, gives parameters that should produce similar sounds than those in $\mathcal{S}$. This alleviates completely the problem of regression that some parameters might be important, others not, and so on, which could not be integrated easily into the regression's loss. Furthermore, it is also possible that two very different sets of parameters would lead to the same sound distribution when plugged in the simulator. This would not be a problem for AVO as it should output a distribution that can converge to either parameters set. With these advantages and drawbacks in mind, let us now proceed to the details of the algorithm. Note that this algorithm is based on the idea behind adversarial optimisation, an algorithm first introduced in generative adversarial networks (GANs [17]).

Generative Adversarial Network GANs are a kind of generative model. They are used to generate samples which resembles an input distribution. For example, GANs can generate images of digits when trained on the MNIST dataset.

We can express the task of learning a generative model as a 2-players zero-sum game between two networks [17]:

- A **generator** $g(\cdot;\theta) : \mathcal{Z} \to \mathcal{X}$ mapping a latent space equipped with a prior distribution $p(\boldsymbol{z})$ to the data space, inducing a distribution

$$\boldsymbol{x} \sim p(\boldsymbol{x};\theta) \Leftrightarrow \boldsymbol{z} \sim p(\boldsymbol{z}), \boldsymbol{x} = g(\boldsymbol{z};\theta) \tag{5.8}$$

- A **classifier** $d(\cdot;\phi) : \mathcal{X} \to [0,1]$ trained to distinguish between true samples $\boldsymbol{x} \sim p_r(\boldsymbol{x})$ and generated samples $\boldsymbol{x} \sim p(\boldsymbol{x};\theta)$

Supervised learning is used to guide the training of generative model.
We want to solve

$$\arg\min_{\theta}\max_{\phi} \mathbb{E}_{\boldsymbol{x}\sim p_r(\boldsymbol{x})}[\log d(\boldsymbol{x};\phi)] + \mathbb{E}_{\boldsymbol{z}\sim p(\boldsymbol{z})}[\log(1 - d(g(\boldsymbol{z};\theta);\phi))] \tag{5.9}$$

In practice, the min-max solution is approximated using alternating stochastic gradient descent on the following losses:

$$\mathcal{L}_d(\phi) = \mathbb{E}_{\boldsymbol{x}\sim p_r(\boldsymbol{x})}[-\log d(\boldsymbol{x};\phi)] + \mathbb{E}_{\boldsymbol{z}\sim p(\boldsymbol{z})}[-\log(1 - d(g(\boldsymbol{z};\theta);\phi))] \tag{5.10}$$

$$\mathcal{L}_g(\theta) = \mathbb{E}_{\boldsymbol{z}\sim p(\boldsymbol{z})}[\log(1 - d(g(\boldsymbol{z};\theta);\phi))] \tag{5.11}$$

for which unbiased gradient estimates can be computed with Monte Carlo integration.

5.2.2.1 AVO

In AVO, generation is replaced with a simulator $g(\boldsymbol{z}, \boldsymbol{p})$, where $\boldsymbol{p}$ is a set of parameters and $\boldsymbol{z}$ is a latent variable used to model the stochasticity of the simulator (which can happen with, for example, noise waveforms). Let $\boldsymbol{p}^*$ denote the true set of parameters which we want to replicate and do not have access to and $p_{\boldsymbol{x}}(\boldsymbol{s})$ denote the distribution over sounds generated with parameters $\boldsymbol{x}$. We have:

$$\boldsymbol{s} \sim p_{\boldsymbol{x}}(\boldsymbol{s}) \Leftrightarrow \boldsymbol{z} \sim p(\boldsymbol{z}|\boldsymbol{x}), \boldsymbol{s} = g(\boldsymbol{z}, \boldsymbol{x}) \quad . \tag{5.12}$$

In AVO a proposal distribution (in our case a multivariate Gaussian) over parameters $q(\boldsymbol{p}; \boldsymbol{\theta})$ is maintained, and the goal to achieve is to have $p_{\hat{\boldsymbol{p}}}(\boldsymbol{s}) \approx p_{\boldsymbol{p}^*}(\boldsymbol{s})$, with $\hat{\boldsymbol{p}} \sim q(\boldsymbol{p}; \boldsymbol{\theta})$.

To this end, a discriminator is trained to guide the training of the generator. Let $d(\boldsymbol{s}; \psi) : \mathcal{R}^{|\boldsymbol{s}|} \rightarrow [0, 1]$ be the discriminator. The discriminator is trained to assign 1 to real examples ($\boldsymbol{s} \sim p_{\boldsymbol{p}^*}(\boldsymbol{s})$)) and 0 to false examples ($\boldsymbol{s} \sim p_{\hat{\boldsymbol{p}}}(\boldsymbol{s})$)). In our case the network used as a discriminator is a LSTM and it is updated thanks to the following loss (which is a simple binary cross-entropy):

$$\mathcal{L}_{\psi} = \sum_{\boldsymbol{s}_m \in \hat{\mathcal{S}}} -y_m \log(d(\boldsymbol{s}_m; \boldsymbol{\psi})) - (1 - y_m) \log(1 - d(\boldsymbol{s}_m; \boldsymbol{\psi})) \tag{5.13}$$

where $y_m = 1$ if $\boldsymbol{s}_m \sim p_{\boldsymbol{p}^*}(\boldsymbol{s})$ and $y_m = 0$ if $\boldsymbol{s}_m \sim p_{\hat{\boldsymbol{p}}}(\boldsymbol{s})$ and where $\hat{\mathcal{S}}$ is a subset composed of half true samples and half false samples (more details in Algorithm 2). The discriminator is used differently than in GANs, as the simulator is not differentiable. The main trick behind AVO is to use the REINFORCE loss, by taking the output of the discriminator as "reward". This allows the distribution to put more density on samples which are classified as true by the discriminator, and less density on samples which are classified as false by the discriminator. This leads to the loss used to train the distribution $q(\boldsymbol{p}; \boldsymbol{\theta})$:

$$\mathcal{L}_{\theta} = \sum_{\boldsymbol{s}_m \in \hat{\mathcal{S}}} \log q(\boldsymbol{s}_m; \boldsymbol{\theta})(\log(1 - d(\boldsymbol{s}_m; \boldsymbol{\psi})) - b) \tag{5.14}$$

where b is a baseline strategy used to reduce variance of REINFORCE estimates (which can be quite high), without adding bias, and is computed as follows:

$$b = \frac{\mathbb{E}\left[(\nabla_{\theta} \log q(\boldsymbol{p}; \boldsymbol{\theta}))^2 (1 - \log(d(\boldsymbol{s}; \boldsymbol{\psi}))^2\right]}{\mathbb{E}\left[(\nabla_{\theta} \log q(\boldsymbol{p}|\boldsymbol{\theta}))^2\right]} \quad . \tag{5.15}$$

We note that some regularization terms are also added to stabilize the learning. These are given more explicitly in Algorithm 2. Finally we note that, as a starting point for $q(\boldsymbol{p}; \boldsymbol{\theta})$, we use a multivariate Gaussian, where each component has a mean of 0.5 and a standard deviation of 0.5.

Algorithm 2: Adversarial variational optimization (AVO)

Inputs : Observed data $\boldsymbol{s}_i \sim p_r(\boldsymbol{s})_{i=1}^N$, simulator g
Outputs : Proposal distribution $q(\boldsymbol{p};\boldsymbol{\theta})$ such that $q(\boldsymbol{s},\boldsymbol{p}) \approx p_{\boldsymbol{p}^*}(\boldsymbol{s})$
Hyper-parameters: k : number of iterations of the discriminator d
M : size of a mini batch
λ : R_1 regularization coefficient
γ : entropy penalty coefficient
b : baseline strategy in REINFORCE estimates

$q(\boldsymbol{p};\boldsymbol{\theta}) \leftarrow$ prior on $\boldsymbol{p}$ (with differentiable and known density, in our case, multivariate Gaussian with $\boldsymbol{\mu} = 0.5$ and $\boldsymbol{\sigma} = 0.5$)
while $\boldsymbol{\theta}$ *has not converged* **do**
  for $i \leftarrow 1$ **to** k **do**
    Sample true data $\{\boldsymbol{s}_m \sim p_{\boldsymbol{p}^*}(\boldsymbol{s}), y_m = 1\}_{m=1}^{M/2}$
    Sample synthetic data $\{\boldsymbol{p}_m \sim q(\boldsymbol{p};\boldsymbol{\theta}), \boldsymbol{z}_m \sim p(\boldsymbol{z}|\boldsymbol{p}_m), \tilde{\boldsymbol{s}}_m = g(\boldsymbol{z}_m;\boldsymbol{p}_m), y_m = 0\}_{m=M/2+1}^{M}$
    $\nabla_\psi U_d \leftarrow \frac{1}{M}\sum_{m=1}^{M} \nabla_\psi \left[-y_m \log(d(\boldsymbol{s}_m;\boldsymbol{\psi})) - (1-y_m)\log(1-d(\boldsymbol{s}_m;\boldsymbol{\psi}))\right]$
    $\nabla_\psi R_1 \leftarrow \frac{1}{M/2}\sum_{m=1}^{M/2} \nabla_\psi \left[||\nabla_\psi d(\boldsymbol{s}_m;\boldsymbol{\psi})||^2\right]$
    $\psi \leftarrow \texttt{RMSPROP}(\nabla_\psi U_d + \lambda \nabla_\psi R_1)$

  Sample synthetic data $\{\boldsymbol{p}_m \sim q(\boldsymbol{p};\boldsymbol{\theta}), \boldsymbol{z}_m \sim p(\boldsymbol{z}|\boldsymbol{p}_m), \tilde{\boldsymbol{s}}_m = g(\boldsymbol{z}_m;\boldsymbol{p}_m)\}_{m=1}^{M}$
  $\nabla_\theta U_g \leftarrow \frac{1}{M}\sum_{m=1}^{M} \left[\nabla_\theta \log q(\boldsymbol{p}_m|\boldsymbol{\theta})(\log(1-d(\tilde{\boldsymbol{s}}_m;\boldsymbol{\psi})) - b)\right]$
  $\nabla_\theta H \leftarrow \frac{1}{M}\sum_{m=1}^{M} \nabla_\theta[-q(\boldsymbol{p}_m|\boldsymbol{\theta})\log q(\boldsymbol{p}_m|\boldsymbol{\theta})]$
  $\theta \leftarrow \texttt{RMSPROP}(\nabla_\theta U_g + \gamma \nabla_\theta H)$

5.2.3 SPIRAL

SPIRAL is a paper [9] which describes a method to synthesize images using an agent. This agent is trained to generate a program which is executed by a graphic engine to obtain an image. The goal of the agent is to fool a discriminator network that distinguishes between real and rendered data, trained with reinforcement learning without supervision.

SPIRAL wants to construct a generative model G capable of sampling from some target data distribution p_d. The simulator $\mathcal{R}$ takes a set of parameters $a = (a_1, a_2, ..., a_N)$ and transforms them into the domain of interest. In other words, they want to recover a distribution p_a such that $p_d \approx \mathcal{R}(p_a)$. p_a is modelled with a recurrent neural network π (called the *policy* network or the *agent*).

In order to optimize π, a GAN framework is used. The generator G goal is to confuse the discriminator D which is trained to distinguish samples drawn from p_d and those generated by the model. This results in the distribution defined by G, p_g to gradually become closer to p_d.

There is a very interesting property in SPIRAL, which distinguishes it from AVO and GANs: the discriminator is not bounded (i.e. $d(\cdot;\phi) : \mathcal{X} \to \mathbb{R}$). Rather, the discriminator is tasked to assign the lowest possible value to false examples and the highest possible value to true examples. As will be seen in Chaper 6, this property can be relevant in our setting. Indeed, one of the caveat of AVO is that if the discriminator learns too fast compared to the "generator", the discriminator will distinguish false examples too easily and assign them a value of 0. Thus, it will be very hard for the "generator" to update its parameters in a direction which improves its performance.

We thus carried some tests, using the discriminator loss of SPIRAL in AVO. That is the loss becomes:

$$\mathcal{L}_\psi = -\mathbb{E}_{\boldsymbol{s}\sim p_{\boldsymbol{p}^*}} d(\boldsymbol{s};\psi) + \mathbb{E}_{\boldsymbol{s}\sim p_{\hat{\boldsymbol{p}}}} d(\boldsymbol{s};\psi) \quad . \tag{5.16}$$

As will be seen, this loss tends to give bigger feedbacks to the generator, resulting in better updates and more stable learning.

5.2.4 Enhanced SPIRAL

Enhanced SPIRAL is a quick simple improvement that we tried lately: instead of starting with a wide distribution $\mathcal{N}(0.5; 0.5)$ of the unknown parameters, we use a network trained using supervised learning (with a dataset of a large numbers of spectrograms, generated using the multivariate normal described in Section 4.7). The idea behind this enhancement is to provide a better initial guess to the REINFORCE algorithm and therefore hoping for a faster convergence. Furthermore, this acts as a mean of regularisation for the discriminator, as it should thus be much more difficult for it to distinguish true and false samples at the beginning of the learning procedure. This should also result in better initial updates and even more stable learning.

Chapter 6

Results

This chapter describes the application of the algorithms explained in the previous chapter to the datasets generated by the simulator and gives a quantitative, as well as qualitative analysis of the results.

6.1 Regression

6.1.1 CNN

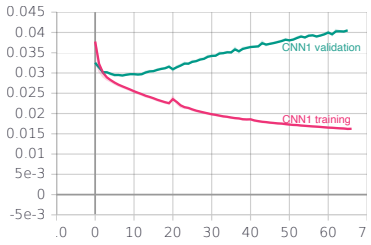
We tested 3 CNN layouts, described at Table 6.1. The dataset is made of 200.000 samples. Each model is more complex than its predecessor. The training and validation graphs are given in Table 6.2.

First, we see that all networks are learning and achieve good results. The mean squared error drops to around 0.03 which is pretty good. Secondly, we can see in the training phase that more complex models are able to overfit easily the datasets. Indeed, the neural networks can fit any data if they have enough parameters. With more parameters, the error on the training set goes lower and lower but it doesn't mean that the accuracy over the validation set will be better. The minimum error in the validation set occurs after a few epochs (remember that these results are coming from a big dataset, so an epoch means a lot of sound samples).

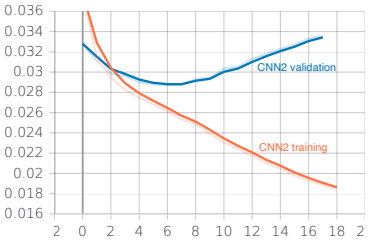
The 4th graph in Table 6.1 shows only the validation curves and the light-blue curve gives the best model. It corresponds to the more complex CNN. We will compare this model with the other networks at Section 6.1.4.

model	1	2	3	4	5	6	7	8
cnn1	conv 3x10x32	maxpool 2x2	conv 2x2x64	maxpool 2x2	fc 128	fc 26	-	-
cnn2	conv 3x10x32	maxpool 2x2	conv 3x10x64	maxpool 2x2	conv 2x2x64	fc 128	fc 26	-
cnn3	conv 3x10x32	maxpool 2x2	conv 3x10x64	maxpool 2x2	conv 2x2x64	fc 512	fc 128	fc 26

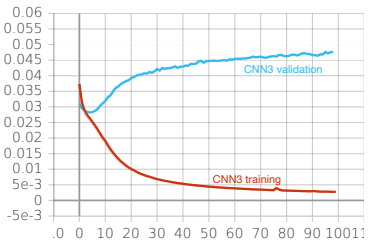
Table 6.1: Convolutional Neural Network layouts



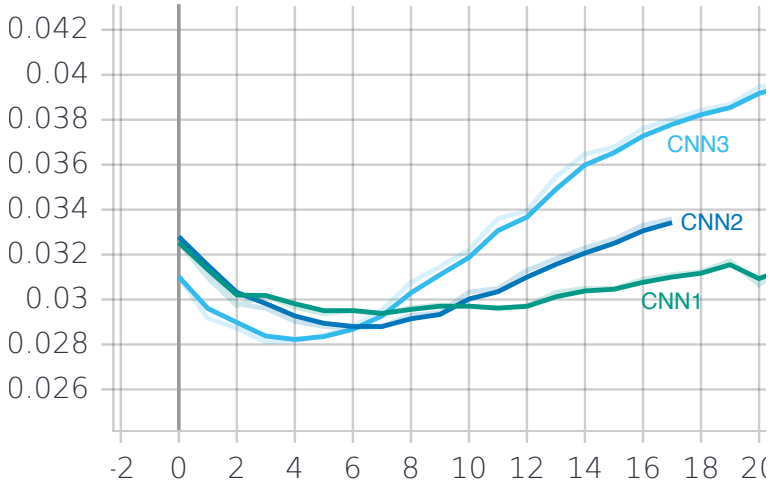
cnn1



cnn2



cnn3



cnn1-2-3 validation

Table 6.2: CNN results

6.1.2 LSTM

We tested 3 LSTM layouts, described at Table 6.3. Our RNNs are unrolled over 690 timesteps (corresponding to the temporal dimension of the spectrogram) and each $\boldsymbol{x}_t$ has dimension 16 corresponding to the number of bins of the STFT. The training and validation graphs are given in Table 6.4. Like the CNNs, we focused on 3 models of increasing complexity. We noticed that all 3 models will eventually overfit but unlike the CNNs, they slowly converge to the same accuracy on the training set. The validation set is showing a minimum, achieved by the three LSTM networks. The dark blue network surprisingly gets the lowest mean squared error despite being the simplest network. We noted that the longest LSTM network got disturbed: after learning, the mean squared error become much higher and it took a large number of epochs for the network to train again and reach a minimum similar to what it had already learned. This might be due to a gradient explosion during learning (which should normally not happen with LSTMs). This behaviour might have been avoided by using learning rate decay or gradient clipping.

model	1	2	3	4
lstm1	lstm 64	fc 64	fc 26	-
lstm2	lstm 128	fc 64	fc 26	-
lstm3	lstm 256	fc 128	fc 64	fc 26

Table 6.3: Long Short Term Neural Network layouts

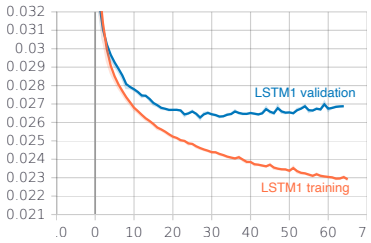
6.1.3 LCNN

We tested 2 locally connected convolutional network layouts, described at Table 6.5. The training and validation graphs are given in Table 6.6. We tested LCNN with 2 models. They both achieved the same accuracy after a few epochs (this number of epochs being similar to their CNN counterparts). Unfortunately the accuracy achieved is lower than the CNN and LSTM, so we didn't spend too much time on these models.

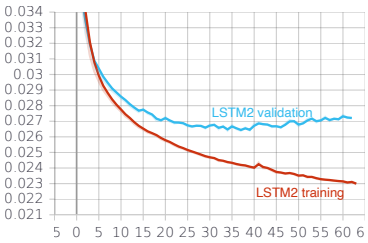
We believe that this lower accuracy is due to the fact that CNN have characteristics that make them invariant to shifts or translations while LCNN don't have these properties. The LCNN will focus and learn on some specific parts of the spectrogram, which could be appealing for various applications but not in our case where the sounds have features which are distributed along the time axis.

6.1.4 Comparison of CNN, LSTM and LCNN

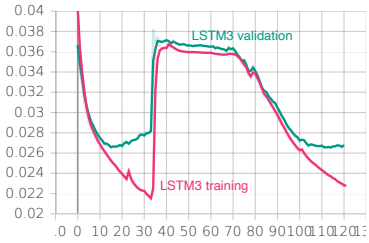
Figure 6.1 is a graph representing all validation curves for the 8 models that we tested (3 CNN, 3 LSTM, 2 LCNN). We are looking for the curve containing the lowest mean squared error. We can directly forget the LCNN which don't match



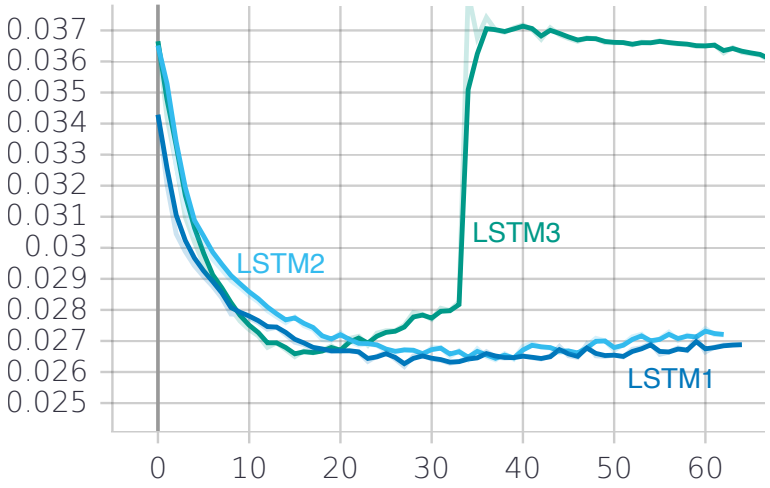
lstm1



lstm2



lstm3

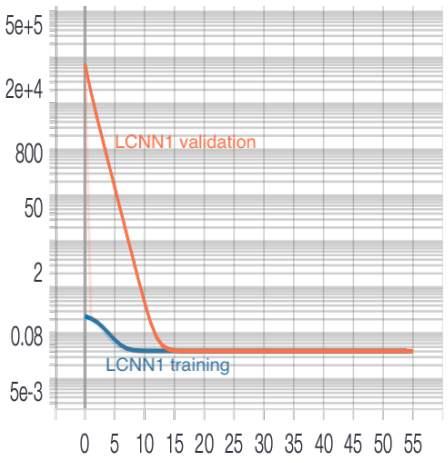


lstm1-2-3 validation

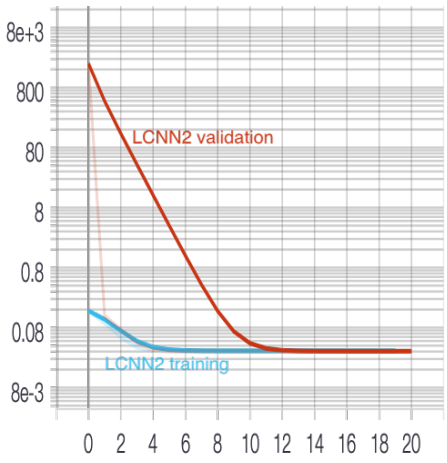
Table 6.4: LSTM results

model	1	2	3	4
lcnn1	lcnn2d 3x10x64	fc 128	fc 26	-
lcnn2	lcnn2d 3x10x64	maxpool 3x3	fc 256	fc 26

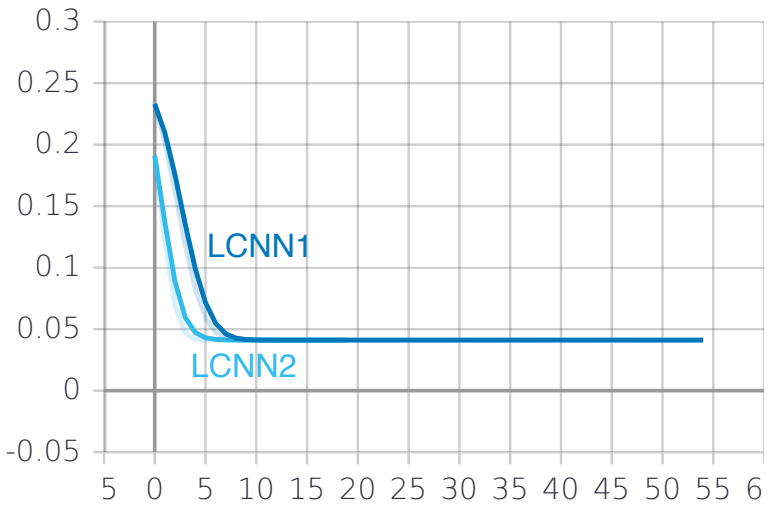
Table 6.5: Locally connected convolutional layouts



lcnn1



lcnn2



lcnn1-2 validation

Table 6.6: LCNN results

the level of accuracy achieved by the CNN and LSTM. We see directly that the winners are the LSTM which have a lower mean squared error than the CNN but the CNN train faster. While it takes less than 5-6 epochs to train the CNN, it takes about 30 epochs to reach the lowest error with the LSTM.

The mean squared errors of all algorithms are given at Table 6.7. We used 3 datasets : small, medium and large, respectively 10^3, 10^5 and 10^6 samples. The small dataset didn't provide good results so we simply dropped it. Results were slightly better by using the bigger dataset. Training MSE is given at epoch 50 to allow for some comparison.

Qualitative analysis. So far, we looked at the mean squared errors to compare the results of the various neural networks. While this measure is easy to compare mathematically, it may not reflect a qualitative comparison of the reconstructed sounds. We explain this by the fact that some parameters have more weight than others:

- Some parameters are discrete, such as the waveforms (sawtooth, square or noise). An error on such parameter may be completely innocuous or may generate a very different sound. So the contribution of such errors is not taken in account properly with a mean squared error.

- Some parameters have strong effects on the final sound while other parameters only produce mild effects. A small variation of a mixer value is almost inaudible while the same variation of a tuning frequency is immediately noticeable!

- Some parameters represent internally exponential values, so the error on these parameters is not linear. A small difference in high values produces an effect much stronger then in low values. This is again something that is not captured in a mean squared error. The resonance of the filter is such a parameter.

We did obviously some listening tests and confirmed our explanations. Some sounds are actually very close to their original while some others are indeed very different. Any error in tuning is immediately detected by a human ear while some other errors are completely indistinguishable. Fortunately, most of the sounds are actually usable and the ones which aren't can often be corrected manually. In most cases, we had good initial sets of parameters that could be used by musicians for further tuning. We note that despite the differences between LSTMs and other architectures being seemingly small in terms of MSE, they can actually lead to some big differences in the sounds produced. Indeed, a mean squared error of 0.0261 (lstm1) leads to an average absolute error of 0.16, whereas a mean square error of 0.04 leads to an average absolute error of 0.2, resulting in an average further difference of 0.04 with respect to the true parameters, in turn leading to highly different sounds.

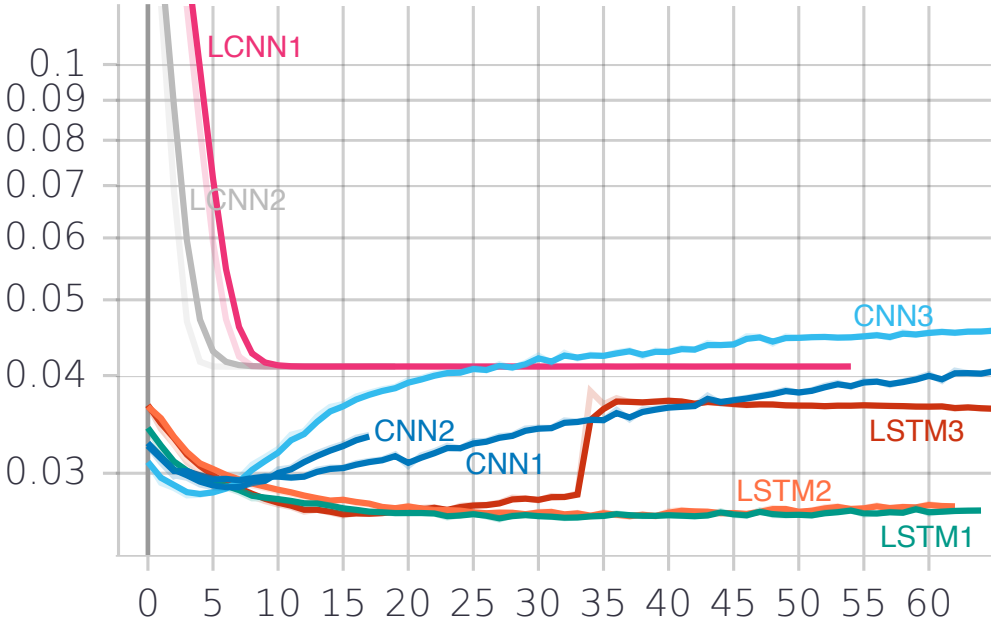


Figure 6.1: Comparison of CNN, LSTM and LCNN

model	medium dataset		large dataset	
	train	validation	train	validation
cnn1	0.01717	0.02856	0.01753	0.02933
cnn2	0.01923	0.03055	0.01845	0.02874
cnn3	0.00429	0.02701	0.00436	0.02811
lstm1	0.02517	0.02721	0.02335	**0.02618**
lstm2	0.02398	**0.02671**	0.02369	0.02643
lstm3	0.02332	0.02866	0.02146	0.02671
lcnn1	0.04435	0.04244	0.04056	0.04108
lcnn2	0.04059	0.04153	0.04058	0.04109

Table 6.7: Mean squared errors comparison

6.2 Adversarial optimisation

6.2.1 AVO

During training, AVO is supposed to move the parameters distributions (each parameter having its own normal distribution) towards the true values used to generate the input spectrograms or to some values which would produce similar sounds.

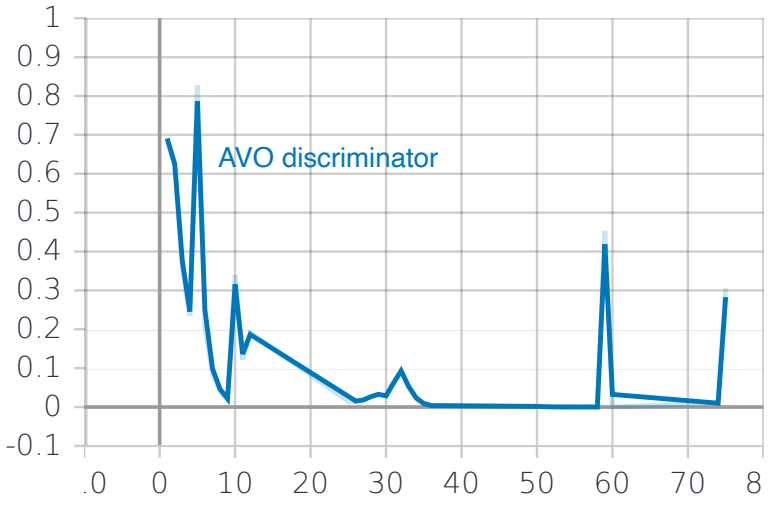
Unlike for standard regression, it is much harder to quantify how well AVO works, as some wildy different set of parameters can lead to similar sounds (in fact, a set of parameters with a greater MSE can lead to sounds that are more similar to the wanted sound than some sets of parameters with lower MSE).

Unfortunately, we were unable to make AVO work well in our setting. We believe that the issue lies in the discriminator whose job is too simple: the generated spectrograms are initially mostly random, so the generator immediately classifies them properly. As can be seen on Table 6.8, the generator's loss is almost always 0, or very close to it, leading to almost no updates or at least no relevant updates. To make the task easier, we tried to predict only 3 or 4 parameters by treating the other 23 or 22 parameters as constant. We finally saw AVO learning something (the unknown distributions slowly converged to the hidden parameters, that is the means slowly moved in the good direction and the standard deviations converging towards low values).

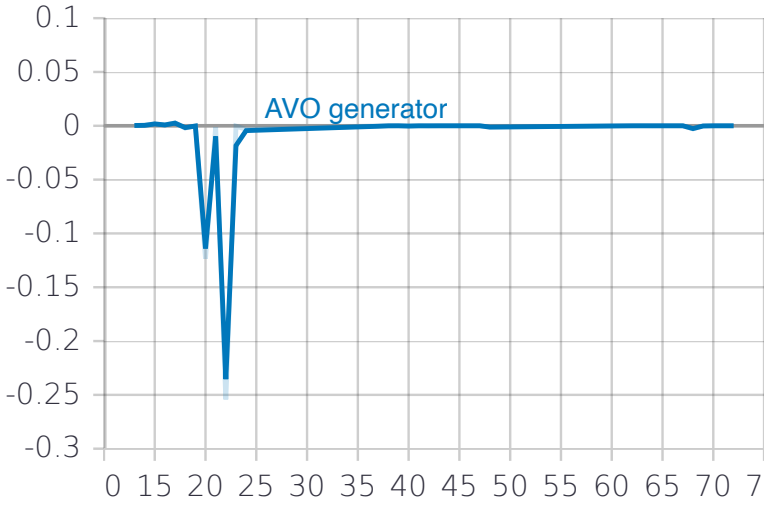
We noticed that this algorithm was highly sensible to hyper-parameter tuning. In fact, building a discriminator with too simple of an architecture led to it not being powerful enough to learn how to distinguish samples properly, while building a too complex discriminator led to it being too powerful and never being fooled. We thus believe that with careful tuning and a better choice of parameters, we could probably make AVO work much better and obtain decent results (even with more than 3 or 4 parameters). However, due to our implementation of the simulator[1], slow computation prevented to make a lot of tests using this method. For example, the first iterations of AVO in Table 6.8 took over 6 hours.

Qualitative analysis Despite being extremely hard to train, and the set of parameters sampled from the proposal distribution not being as close to the true set (from a MSE point of view) than those of standard LSTMs trained with regression, the results are actually quite good. The sounds produced using the parameters sampled from the proposal distribution seem to adhere quite well with those given as input to the algorithm. The caveat being of course that we only learn 3 or 4 parameters. Also note that we chose parameters which should have a high influence on the sound (with basic prior knowledge on the synthesizer), as the analysis would otherwise not make much sense.

[1] As we already mentioned before, the simulator is written using `tensorflow` and `numpy`. While this looks reasonable, it prevents running the whole simulation on the GPU. Furthermore, we wrote the simulator at the beginning of this project without envisaging the need to compute several spectrograms in one pass. These two limitations result in an unfortunate slow computing time in AVO where multiple simulations must be run at each iteration of the algorithm. These simulations being sequentially run on the CPU literally make the computation time impractical.



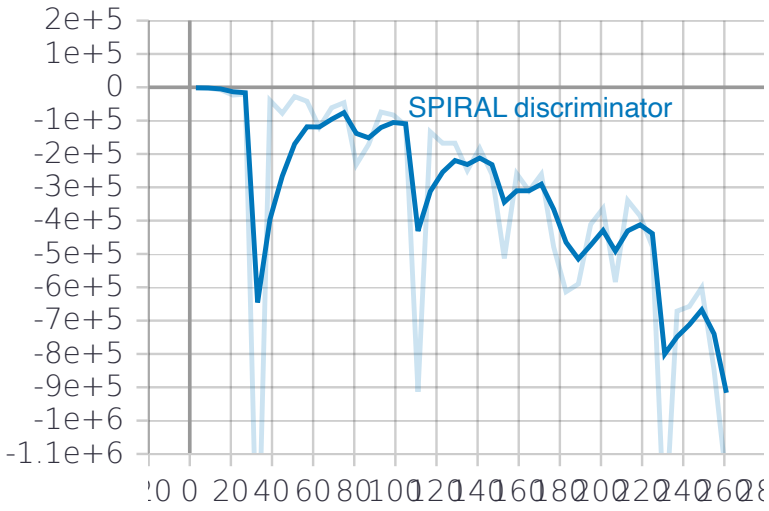
discriminator loss



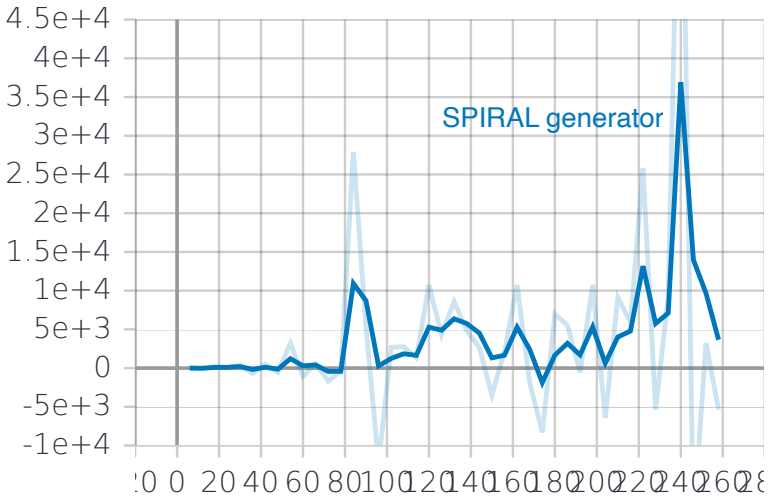
generator loss

Table 6.8: AVO results

6.2.2 SPIRAL



discriminator loss



generator loss

Table 6.9: SPIRAL results

Thanks to the new loss used for the discriminator, as can be seen in Table 6.9, the losses tend to have a much bigger magnitude than in Table 6.8 resulting in more stable learning. Through the sawtooth shape of the generator's loss on Table 6.9, one can see that the generator manages to learn how to fool the discriminator (before this last one enhances and learns how to discriminate even better). However, despite obtaining decent results, one can see that the discriminator still ends up "beating" the generator in the long term. We note that these results were also obtained while limiting the number of varying parameters to 4. It remains however extremely difficult to compare the results quantitatively. From a qualitative viewpoint, the sounds were actually quite similar to those expected. In fact, a noticeable improvement could be heard from the simple AVO algorithm.

6.2.3 Enhanced SPIRAL

The result is given at Figure 6.2 where a single parameter distribution is shown after different numbers of iterations. Initially, the mean is obtained by the "lstm2" network and the standard deviation set to be rather large. We see that the mean slowly moved towards the ground truth (it was already close at the start) and the standard deviation slowly decreased. Using this method, we were able to increase the number of varying parameters to 9, against 4 for the SPIRAL method. We believe that tuning the hyper-parameters better would lead to the possibility of managing even more varying parameters.

Again, it is hard to qualify quantitatively the performance of enhanced SPIRAL. However, the sounds generated were on the same level as SPIRAL, with 5 more varying parameters. This proves that the regularisation added with the "hot-start" was beneficial to the training procedure. In Figure 6.2, we provide a plot showing the evolution of the distribution for one of the 9 varying parameters, over the course of training. As one can see, for this parameters, the mean ends up close to that of the true parameter (note that this could well not be the case even if the sound generated still matched the true sounds distribution).

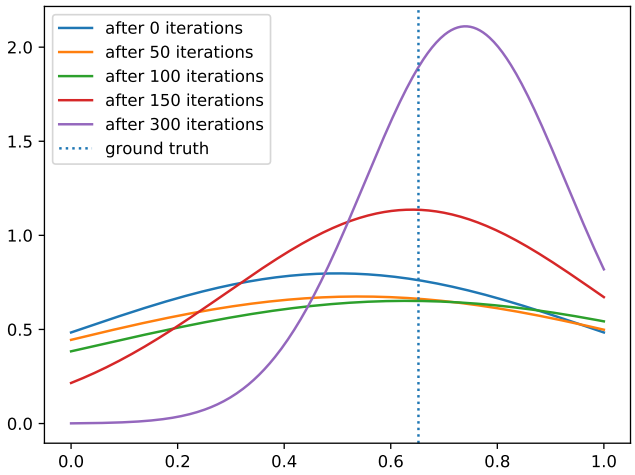


Figure 6.2: Enhanced SPIRAL

6.2.4 Possible improvements

Some more improvements for the adversarial methods could be thought of, but due to time constrains, the implementation and tests of such improvements proved unfeasible. Most of these improvements would lead to the regularisation of the discriminator, as it showed to be the main problem we encountered.

- We could use the trunk of a pretrained neural network, with regression, such as "lstm2" as the trunk of the discriminator and freeze its weights. We would then simply train the head of the discriminator with the AVO procedure.

The trunk of the pretrained network should have learned relevant features for distinguishing sounds. However, these features were not trained to distinguish true and false samples. Thus, using only the head of the network for such purpose would lead to a regularization of the discriminator.

- We could add some noise to the true parameters when sampling the sounds. This would lead to true sounds samples with more variance, making the task for the discriminator harder. With this technique, the discriminator would be less certain of what is a true sound and thus again, be regularized.
- Do a grid-search on hyper-parameters of the algorithm.

Chapter 7

Conclusion

This thesis has been a long journey! We started from a very well defined problem: given a synthesizer and a sound, find how the synthesizer could be programmed to make that sound or a somehow relatively similar sound. This task is difficult for humans and it turned out that it is also difficult for machines.

We selected the latest analog synthesizer made by Roland and reverse engineered it. This step was necessary so that we could build a simulator and hence be able to generate sounds much faster than using the real synthesizer itself. We discovered that several parameters of the synthesizer cannot be guessed from a single sound and so we simplified our problem by removing such parameters.

Armed with the simulator, we generated datasets and processed them with different machine learning algorithms. With supervised learning, good results were obtained using convolutional neural networks and recurrent neural networks. One difficulty was to qualify the guessed parameters. A mean squared error with these algorithms achieved a relative similitude between the true sound and its estimate. However, in certain cases, the estimated sound was very different because some parameters are more important than others even if they contribute for the same amount in a mean squared errors. This leads to the question "can the neural networks learn how to differentiate sounds?". We applied learning algorithms from two papers, AVO and SPIRAL, which are based on adversarial techniques. These techniques obtained good and encouraging results but due to technical limitations of our simulator (which caused extremely long compute times), we couldn't do as many experiments as we wanted. For example, we couldn't sweep the hyper parameters to tune them.

The next step would have been to use the real synthesizer to generate a dataset of real samples. The supervised learning algorithms are well suited for processing such a dataset: even if its generation will take a lot of time (several days), this operation must only be done once. The adversarial algorithms won't be practical in this situation because they use the sound generator during learning.

One area that remains to be explored for this particular problem is finding the parameters which were removed because of their dynamic behavior. Instead of providing a single sound to the machine learning algorithm, we could provide several samples of the same sound. These samples would have to be created at different frequencies and played with different velocities. We would hope that the algorithms would be able to recognize the dynamic characteristics and hence guess the missing parameters.

Appendix A

JX parameters

```
;;; sound parameters
SOUND_PARAM_DCO1_RANGE          equ $80
SOUND_PARAM_DCO1_WF             equ $81
SOUND_PARAM_DCO1_TUNE           equ $82
SOUND_PARAM_DCO1_LFO            equ $83
SOUND_PARAM_DCO1_ENV            equ $84
SOUND_PARAM_DCO2_RANGE          equ $85
SOUND_PARAM_DCO2_WF             equ $86
SOUND_PARAM_DCO_XMOD            equ $87
SOUND_PARAM_DCO2_TUNE           equ $88
SOUND_PARAM_DCO2_FTUNE          equ $89
SOUND_PARAM_DCO2_LFO            equ $8A
SOUND_PARAM_DCO2_ENV            equ $8B
SOUND_PARAM_AFTER_VIB           equ $8C
SOUND_PARAM_AFTER_BRI           equ $8D
SOUND_PARAM_AFTER_VOL           equ $8E
SOUND_PARAM_DCO_DYNA            equ $8F
SOUND_PARAM_DCO_MODE            equ $90
SOUND_PARAM_MIX_DCO1            equ $91
SOUND_PARAM_MIX_DCO2            equ $92
SOUND_PARAM_MIX_ENV             equ $93
SOUND_PARAM_MIX_DYNA            equ $94
SOUND_PARAM_MIX_MODE            equ $95
SOUND_PARAM_HPF_FREQ            equ $96
SOUND_PARAM_VCF_FREQ            equ $97
SOUND_PARAM_VCF_RES             equ $98
SOUND_PARAM_VCF_LFO             equ $99
SOUND_PARAM_VCF_ENV             equ $9A
SOUND_PARAM_VCF_KEY             equ $9B
SOUND_PARAM_VCF_DYNA            equ $9C
SOUND_PARAM_VCF_MODE            equ $9D
SOUND_PARAM_VCA_LEVEL           equ $9E
SOUND_PARAM_VCA_DYNA            equ $9F
SOUND_PARAM_CHORUS              equ $A0
SOUND_PARAM_LFO_WF              equ $A1
SOUND_PARAM_LFO_DELAY           equ $A2
SOUND_PARAM_LFO_RATE            equ $A3
SOUND_PARAM_ENV1_ATT            equ $A4
SOUND_PARAM_ENV1_DECY           equ $A5
SOUND_PARAM_ENV1_SUS            equ $A6
SOUND_PARAM_ENV1_REL            equ $A7
SOUND_PARAM_ENV1_KEY            equ $A8
SOUND_PARAM_ENV2_ATT            equ $A9
SOUND_PARAM_ENV2_DECY           equ $AA
SOUND_PARAM_ENV2_SUS            equ $AB
SOUND_PARAM_ENV2_REL            equ $AC
SOUND_PARAM_ENV2_KEY            equ $AD
SOUND_PARAM_VCA_MODE            equ $AF

SOUND_PARAM_PORTA_TIME          equ $B0
SOUND_PARAM_MIDI_VOL            equ $B1
SOUND_PARAM_BENDER              equ $B2
SOUND_PARAM_AFTERTOUCH          equ $B3
SOUND_PARAM_MASTER_TUNE1        equ $B4
SOUND_PARAM_PORTA_SW            equ $B5
SOUND_PARAM_SILENCE             equ $B6
SOUND_PARAM_BEND_RANGE          equ $B7
SOUND_PARAM_AFTER_VIB_SW        equ $B8
SOUND_PARAM_AFTER_BRI_SW        equ $B9
SOUND_PARAM_AFTER_VOL_SW        equ $BA
SOUND_PARAM_HOLD_SW             equ $BB
SOUND_PARAM_MODULATION          equ $BC
SOUND_PARAM_CHECK_ENV_SW        equ $BD
SOUND_PARAM_MASTER_TUNE2        equ $BE
SOUND_PARAM_BENDER_POLARITY     equ $BF
```

Bibliography

[1] J. Allen. Short term spectral analysis, synthesis, and modification by discrete fourier transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 25(3):235–238, 1977.

[2] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[3] J. M. Chowning. The synthesis of complex audio spectra by means of frequency modulation. *Journal of the audio engineering society*, 21(7):526–534, 1973.

[4] J. Collins, J. Sohl-Dickstein, and D. Sussillo. Capacity and trainability in recurrent neural networks. *arXiv preprint arXiv:1611.09913*, 2016.

[5] D. Creasey. *Audio processes: musical analysis, modification, synthesis, and control*. Taylor & Francis, 2016.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[7] R. Dey and F. M. Salemt. Gate-variants of gated recurrent unit (gru) neural networks. In *2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS)*, pages 1597–1600. IEEE, 2017.

[8] S. Dhabu, A. Ambede, N. Agrawal, K. Smitha, S. Darak, and A. Vinod. Variable cutoff frequency fir filters: a survey. *SN Applied Sciences*, 2(3):343, 2020.

[9] Y. Ganin, T. Kulkarni, I. Babuschkin, S. Eslami, and O. Vinyals. Synthesizing programs for images using reinforced adversarial learning. *arXiv preprint arXiv:1804.01118*, 2018.

[10] A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[11] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[12] R. Jozefowicz, W. Zaremba, and I. Sutskever. An empirical exploration of recurrent network architectures. In *International conference on machine learning*, pages 2342–2350, 2015.

[13] B. Katz and R. A. Katz. *Mastering audio: the art and the science*. Butterworth-Heinemann, 2003.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[15] Y. LeCun, Y. Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[16] G. Louppe, J. Hermans, and K. Cranmer. Adversarial variational optimization of non-differentiable simulators. *arXiv preprint arXiv:1707.07113*, 2017.

[17] A. Marafioti, N. Holighaus, N. Perraudin, and P. Majdak. Adversarial generation of time-frequency features with application in audio synthesis. *arXiv preprint arXiv:1902.04072*, 2019.

[18] B. McFee, C. Raffel, D. Liang, D. P. Ellis, M. McVicar, E. Battenberg, and O. Nieto. librosa: Audio and music signal analysis in python. In *Proceedings of the 14th python in science conference*, volume 8, 2015.

[19] C. Olah. Understanding-lstms. *URL: http://goo. gl/hwG4OC*, 2015.

[20] M. Singh and E. N. K. Garg. Audio noise reduction using butterworth filter. *Int. J. Comput. Org. Trends*, 6(1), 2014.