ATIAM

Internship Report

Learning latent spaces for real-time synthesis of audio waveforms

Author :
Cyran AOUAMEUR

Supervisors :
Phillipe ESLING
Gaëtan HADJERES

March 12th - August 31st 2018

Acknowledgments

I would first like to express my gratitude to my supervisors Philippe and Gaëtan, for their constant availability and help.

I would also like to thank Sony CSL and Ircam teams for having shared their knowledge and their helpful hints all along this internship.

Abstract

Modern synthesizers are getting increasingly powerful and now provide an overwhelming amount of parameters to carve a sound spectrum. This simultaneously increases creative freedom but can also complicate the sound design process. In parallel, recent generative learning models have been developed towards audio synthesis.

Here, we aim at providing an intuitive control over sound synthesis with deep learning models, through *synthesis by learning*. Only a scarce number of approaches have been proposed to deal with this new type of synthesis, which allow to learn a synthesizer directly from audio sample examples. One of the most notable proposal rely on the framework of *variational autoencoders*, which allows to generate sounds from a *parameter latent space*, by simultaneously learning inference and generation networks from existing data. However, handling temporal information with such models is still a central issue that hampers their generalization to complex sounds.

In this work, we develop generative models for audio synthesis that are able to handle complex temporal information, thus, allowing to generate a wide variety of sounds. To evaluate the capacity of such models, we collected and labeled a dataset representing a variety of percussive sounds. Here, we developed three separate models based on combinations of variational autoencoders and convolutional neural networks for audio synthesis.

First, we introduce an architecture using 2-dimensional convolutions that is able to learn and generate time-frequency representations. We show that, while the former approach produces accurate drum distributions and provides intuitive control over the latent space, it is unable to provide a real-time audio synthesis system, due to the need to invert the time-frequency representations. Hence, we propose an inversion model based on recurrent network designed to perform the inversion of time-frequency representations to audio waveform. Although the model is able to train correctly, we show that it fails to generalize on examples without prior information. We analyze the reasons behind this problems and propose directions of future work to enhance the model.

Finally, we expand the original 2-d synthesis model using 1-dimensional convolutions in order to learn and generate audio waveforms directly. This model, trained on slices of audio waveform, yields variable reconstruction quality depending on the nature of the sounds. While performing poorly on noisy sounds, it gives interesting results for kick samples. We provide extensive experimentations and analyses of results.

Contents

1 Introduction

Research in computer music is usually divided between two main purposes, which are either to create tools to better *analyze* existing music or to find new ways of *creating* music. Hence, researchers usually benefit from scientific advances in concomitant fields to consider these problems from a different perspective [1]. However, the recent advent of *deep learning*, which is now commonly used in fields like Natural Language Processing (NLP), is still a long way from being fully adapted to the specificities of musical creation. Indeed, even if music analysis has benefited from machine learning in the framework of Music Information Retrieval (MIR), the use of such techniques in music generation seems to be only sporadic yet. Very recently, some deep learning models were developed, focusing on speech generation [2, 3]. These ***generative models*** perform what we could define as ***synthesis by learning***. In this work, we aim at using these new generative learning approaches in order to create a novel type of audio synthesis technique, based on learning, that would give a wide range of creative expression while being easily controllable. The following short introduction details our motivation.

1.1 Sound synthesis

Sound synthesis has been a field of interest for over a century now, opening interesting avenues for both musicians and scientists alike [4]. We can define *sound synthesis* as the process of generating sound, using electronic hardware or software. Research in this field was mainly motivated by the will to expand the degrees of freedom in sounds and creative expression. Concretely, it allows to generate sounds that are not necessarily feasible with existing acoustic instruments. The tremendous success of synthesizers in the late 60's shaped the sound of new generations so much that the synthesizers are amongst the most widely used instruments in nowadays western music production. This novelty in music was also made possible by yet another type of scientific advances: the continuous technological improvements in terms of computational resources. Moreover, descending costs of computer technology made synthesizers affordable to the general public, which greatly stimulated researches in synthesis techniques.

Since the beginning of the 20th century, researchers have developed numerous techniques based on a wide variety of paradigms. Here, we propose a simplified classification of the most noticeable audio synthesis approaches, relying on the taxonomy proposed by [4]. We refer interested readers to this article for more details.

Spectral model The development of the harmonic analysis by Joseph Fourier in 1807 led to the formalization of a so-called *spectral model* for sound. In this model, a sound pressure wave $y(t)$ is seen and decomposed as a sum a sinusoids

$$y(t) = \sum_{i=1}^{N} A_i(t) sin[\theta_i(t)] \tag{1}$$

where $A_i(t)$ is the amplitude of i-th partial and $\theta_i(t)$ its phase over time t. Based on this formulation, we can generate a wide variety of sounds by adding pure tones (sinusoids), a process called *additive synthesis*. Reciprocally, *subtractive synthesis* starts from a spectrally rich sound (usually a broadband noise) and, then, filters out specific frequencies in order to carve the sound spectrum.

Physical model Researches in instrument making and acoustics also led scientists to model instruments as systems ruled by equations. In *physical modeling synthesis* [5], the sound to be generated is computed with a mathematical model that attempts to replicate the acoustical laws governing the production of sound. For example, one could model how the strings of a guitar are vibrating, and how this movement is coupled with those of the neck and the soundboard. Examples of application of these models are the Karplus-Strong algorithm [6] or the modal methods [7], which aim to describe instruments as a group of oscillators with various resonance frequencies.

Abstract algorithms The terminology of *abstract models* has been proposed by some researchers [4], based on the fact that these methods were not designed as an attempt to replicate an existing physical phenomenon. An important example includes the Frequency Modulation (FM) synthesis method, introduced in 1973 by Chowning [8] which is based on chained oscillators. However, we can consider later experiments such as the work by Risset on brasses as part of the spectral paradigm, since they used the FM principle jointly with spectral observations.

Direct (or waveform-based) synthesis Finally, the class of *waveform-based* methods takes advantage of recorded sounds in order to use them in a synthesis process. The simplest example is *sampling-based* synthesis where pre-recorded sounds are used and processed to play drum sounds on a keyboard for example. *Concatenative* and *granular* synthesis [9, 10] can be seen as extensions to sampling methods. Indeed, these techniques are also based on pre-recorded sounds but the samples are sliced into "grains" that are recombined based on statistical transition models.

As we can see, the existing methods for direct waveform synthesis are solely based on the recombination of existing recorded chunks. This stems from the fact that the raw waveform data (usually recorded at 44100 samples per second) is of very high dimensionality, which renders its direct modeling a daunting task. In this work, we will address this complex category, in order to tackle novel avenues in musical creativity.

1.2 The importance of control

Recently, music production has turned essentially digital, hence, drastically increasing the scope of possibilities in terms of sounds synthesis. Modern synthesizers are getting increasingly powerful thanks to the computational power provided by new generations of

CPUs. This has led to a concomitant increase in the creative freedom provided by sound synthesizer. However, even if it is now theoretically possible to tune a sound to one's will, the sound design process has become increasingly complex given the overwhelming amount of parameters provided by modern synthesizers.

This casts the light on the crucial trade-off between *control richness* and *ease of use*. Therefore, a method allowing an easy and rich fine-tuning of sounds becomes a key requirement in music production, especially for non-expert users. Citing Julius O. Smith (1991) [4], "*the problem* [of musical control] *can be better appreciated by considering that instruments made of metal and wood are played by human hands; therefore, to transfer past excellence in musical performance to new digital instruments requires either providing an interface for a human performer or providing a software control layer that "knows" a given musical context. The latter is an artificial intelligence problem*".

Since this observation, numerous research efforts have been done in the domain of user experience, in order to provide interfaces that enhance the fluidity of human-machine interactions. However, artificial intelligence is not yet used in music generation and control as much as in other fields like Natural Language Processing (NLP) or even Music Information Retrieval (MIR). Only very recently, some machine learning models were developed specifically for the problem of audio generation. These ***generative models*** perform what we could define as ***synthesis by learning***. Amongst machine learning approaches that could suit our purpose, generative modeling allows to perform audio synthesis by learning while tackling the question of intuitive parameter control [11].

1.3 Generative models in audio

Generative models are a flourishing class of machine learning approaches whose purpose is to generate novel data based on the observation of existing examples [12]. This process is usually performed by trying to model the underlying probability distribution of the data. Hence, based on a dataset of audio samples, seen as multi-dimensional vectors (as audio is encoded in 44,1 kHz, one second of audio is a 44100-dimensional vector), a generative model will try to capture the underlying dependencies between different dimensions to understand the distribution of this data [12].

To formalize this problem, we rely on a set of data $\{\mathbf{x}_n\}_{n\in[1,N]}$ defined in a high-dimensional space $\mathbf{x}_i \in \mathbb{R}^{d_x}$. We assume that these examples follow an underlying probability distribution $p(\mathbf{x})$ that is unknown. The goal of generative models will be to approximate this distribution, so that examples that are coherent with the dataset are highly probable, while irrelevant data is less probable. Having access to this distribution, we can later sample from it and, therefore, generate novel data having characteristics similar to that of the dataset. Furthermore, we introduce a set of *latent variables* $\mathbf{z}$ in the model. These latent variables can be understood as a higher-level representation of the data in a simpler space that could have led to generate a given example. Therefore, the set of latent variables (which we call the ***latent code***) is defined in a lower-dimensional space $\mathbf{z} \in \mathbb{R}^{d_z}$ with $d_z \ll d_x$. The relationship between the data $\mathbf{x}$ that we want to model

and the latent variables $\mathbf{z}$ that might generate our data can be expressed through their joint probability distribution

$$p(\mathbf{z}, \mathbf{x}) = p(\mathbf{z})p(\mathbf{x} \mid \mathbf{z})$$

There is an essential advantage in introducing latent variables: they enhance the *expressiveness* of the model. Indeed, these variables can be interpreted as a representation of the generative factors leading to the data. Hence, we make the assumption that different latent codes $\mathbf{z}$ can allow the model to generate various sounds coherent with the examples $\mathbf{x}$ (with similar *properties*), while being more expressive than the original data space. Thus, the latent space can be seen as a continuous synthesis parameters' space that we can sample from in order to generate an infinite variety of sounds (see Fig. 1).

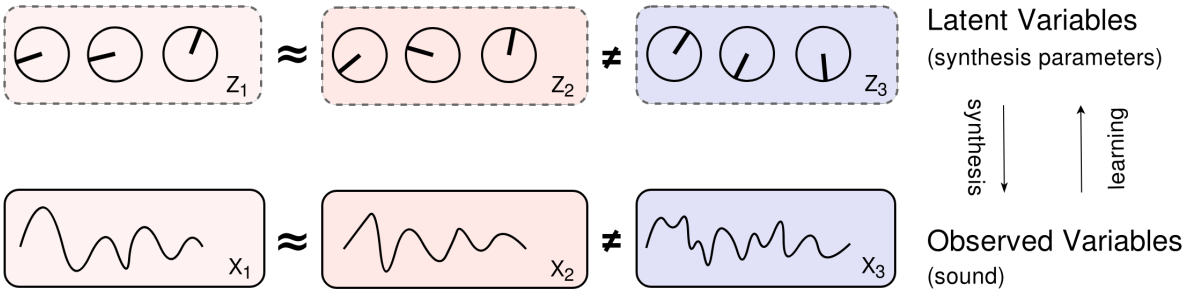


Figure 1: Learning and generation with latent variables. Similar latent codes z_1 and z_2 should generate similar waveforms x_1 and x_2. A very different latent code z_3 should generate a very different waveform x_3

1.4 Learning latent spaces for audio

As we discussed in the previous sections, only few systems have been very recently proposed to address the learning of latent spaces for audio data. Prior systems were specifically designed to model sequences including time-dependent information. Recurrent Neural Networks (RNN) have proven very efficient on processing such sequences [13, 14]. Some variants were designed to model and synthesize waveforms, such as SampleRNN [3] but it does not allow the user to control the synthesis in real-time. Similarly, Google's Wavenet succeeds in modeling waveforms with Convolutional Neural Networks (CNN), thanks to *dilated causal convolutions*. However, these models give little cue and control over the output or the features it results from. Moreover, these models require very large number of parameters, long training times and a large number of examples.

Separately from these approaches, the framework of Variational Auto-Encoders (VAE) was recently developed [15]. VAEs are generative machine learning models based on a Bayesian framework, which allow to model the data distribution even with a small number of examples and also provide an explicit modeling of the latent space. Using VAEs, Esling et al. [11] created a system capable of learning a generative space where instrumental sounds are organized with respect to their timbre. One of the main drawbacks of this system is that it has been trained on single frames of spectrums. Therefore, each

latent code is associated with a single spectrum and the model lacks temporal modeling. Hence, this hampers the capacity of the model to easily allow users to generate evolving structured temporal sequences.

For instruments whose frequency content is highly varying across time, this problem becomes widely problematic (as shown in Fig. 2). This is why, in this work, we will tackle this aspect by trying to extend the VAE to learn temporal dynamics of spectral distributions and model specifically drum sounds. We will first rely on time-frequency representations and try to learn sounds through these representations that intrinsically contain both the temporal and frequency evolution of sounds. Then, we will address modeling sounds directly from the raw waveform data. Thus, we will try to learn from the waveform directly while benefiting from the control provided by VAEs.

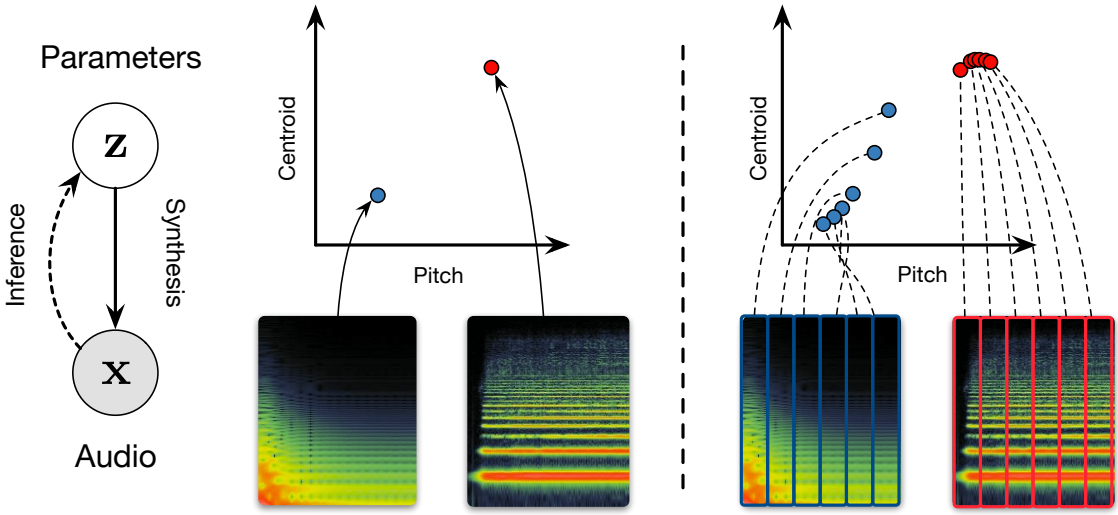


Figure 2: Spectrograms of a kick drum and a flute and their positions in the latent space. On the right, we show a latent space constituted with single frames. For the flute, the frequency content is pretty stable through frames so we could generate a sound from one single point. Nevertheless, is impossible to generate convincing drums without a complicated interpolation strategy. Since we want to generate all types of sounds effortlessly, a potential solution is that the whole transforms are encoded as a single latent position (left situation).

1.5 Our proposal

In this work, we aim to create a controllable audio synthesis space that explicitly contains informations about time so that we can use it to synthesize novel sounds in an intuitive manner. Navigating in this space should allow us to explore the whole diversity of sounds we have trained the model on, and also generalize from the learning process to generate novel sounds. Hence, the latent space should be organized along understandable properties and generation from such a model should be fast enough for the exploration to be smooth (real-time sound rendering), to ease interactivity.

In the remainder of this document, we provide an introduction to deep learning, starting from the basic concepts of neural networks (Sec. 2.1). Then, we introduce the Variational Auto-Encoder (VAE) (Sec. 2.2.3) framework that we will use, based on the concept of Variational Inference (VI). Then, we briefly present recent generative models targeted at learning waveform data and how we can adapt them to fit into the variational approach (Sec. 2.3). After this state-of-the-art, I will present the 3 models that I have developed along this internship and present the experimentations and results that I have obtained based on a drum sound dataset that I collected specifically for this work.

1. First, the convolutional VAE model, which is able to generate full time-frequency spectrums (Sec. 3.2)
2. Second, a RNN-based inversion model that tries to retrieve a sound waveform from its amplitude spectrum (Sec. 3.3)
3. Third, an hybrid temporal model, that takes advantage of dilated convolutions to learn directly on the waveform data (Sec. 3.4)

This will be followed by a discussion and comparisons of the different experimentations, in order to analyze and understand the strength and weaknesses of different approaches. Finally, a brief conclusion will summarize the content of the report, stress my contributions and introduces directions of future work.

2 State of the art

In this work, we will mostly rely on Variational Auto-Encoders (VAEs), which have been recently introduced based on Variational Inference (VI) [15]. Although VAEs are defined as a probabilistic extension to traditional AEs, they still rely on Artificial Neural Networks (ANNs) as one of their core components. Hence, we here only briefly introduce the basic principles of ANNs (2.1) before developing certain architectures, namely CNNs and RNNs that will be used throughout our work (2.1.2). Then, we will stress out the differences between supervised and unsupervised learning, leading to the formulation of VAEs (2.2). Finally, we will detail some of the recent generative models introduced for handling raw waveform data (2.3).

2.1 Artificial Neural Networks

An artificial neural network is a network composed of units called *neurons*. Each of these neurons receive a set of inputs $\mathbf{x}$ and outputs a value $\mathbf{a}$ called *activation* by computing

$$\mathbf{a} = \phi\left(\mathbf{w}.\mathbf{x} + b\right)$$

where $\mathbf{w}$ is a set of parameters called *weights* and ϕ is an *activation* function, which is non-linear. Usually, the functions used for ϕ are the *sigmoid* or $ReLU(\mathbf{x}) = max(0, \mathbf{x})$ activations. Thus, through the linear combination of input values and the application of such a function, each neuron performs a non-linear transformation of its input. The complete network is built by having *layers* of independent neurons that are connected to neurons in the following layer, overall defining a weighted graph and producing an output $\mathbf{y}$ (see fig. 3) from the neurons of its last layer.

If we denote as $\mathbf{\Theta}$ the set of all parameters (weights and biases of all neurons) in the network $\mathbf{\Theta} = \{\mathbf{W}, \mathbf{b}\}$, we can see that another way to see neural networks is that they define a function $g_{\mathbf{\Theta}} : X \to Y$ parametrized by a set of parameters Θ. The goal of neural network is usually that they act as a *function approximator* by learning to produce an expected answer $\mathbf{y}^*$ to each input $\mathbf{x}$

$$\mathbf{y}^* \approx \mathbf{y} = g_{\mathbf{\Theta}}\left(\mathbf{x}\right)$$

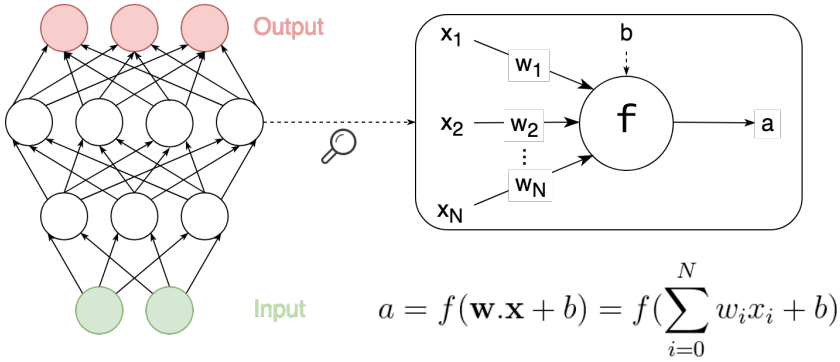


Figure 3: Neural Networks are composed of layers of neurons. These neurons produce an output which is the result of its activation function f applied to a weighted sum of its input x (plus a bias b). The activation of neurons at layer l becomes the input of neurons at layer $l+1$ and so on.

Hence, training such a network consists in optimizing $\mathbf{\Theta}$ to minimize the difference between the output of the network $\mathbf{y}$ and the desired output $\mathbf{y}^*$. To do so, the weights are optimized through gradient descent, by iteratively reducing the approximation error

$$g^*_{\mathbf{\Theta}} = \underset{\mathbf{\Theta}}{\operatorname{argmin}} \mathcal{L}\left(\mathbf{y}^*, \mathbf{y}\right)$$

where $\mathcal{L}$ is a *loss function* allowing to measure our approximation error, such as the Mean Squared Error (MSE) $\mathcal{L}\left(\mathbf{y}^*, \mathbf{y}\right) = \|\mathbf{y}^* - \mathbf{y}\|^2$. For the sake of brevity, we will not go deeper in the specificities of neural networks, but interested readers can find a more-in-depth discussion on neural networks in [16, 17].

2.1.1 Convolutional Neural Networks

Each layer in a Convolutional Neural Network (CNN) consists in a set of N *kernels* (or *filters*) that are convolved across the input. We denote $\{k^l_n\}_{n\in[1;N]}$ this set of kernels for the layer l. For a given layer, these kernels all share a unique size *kernel size*. By convolving each one of its N kernels across a d-dimensional input x, a convolutional layer produces N d-dimensional outputs (one for each kernel) called *feature maps* that we denote $\{a^l_n\}_{n\in[1;N]}$. Hence, the computation of the n-th activation map in layer l for input x is defined as

$$a^l_n = \sum_{m=1}^{M} k^l_n \star x_m + b^l_n \tag{2}$$

Thus, the feature map corresponding to kernel n consists in the sum of the d-dimensional discrete convolutions (denoted by the $\star$ operator) between the kernel n and each one of the d-dimensional data $\{x_m\}_{m\in[1;M]}$, plus an optional bias.

Since the exact expression of the d-dimensional discrete convolution is highly dependent on other hyper-parameters (padding, stride), we will just explain the basic principle of this operation. The 2-dimensional discrete convolution consists in sliding a kernel over

an input. For each position, we compute the dot product between the kernel and the zone of the input located "under" the kernel (see Fig. 4).

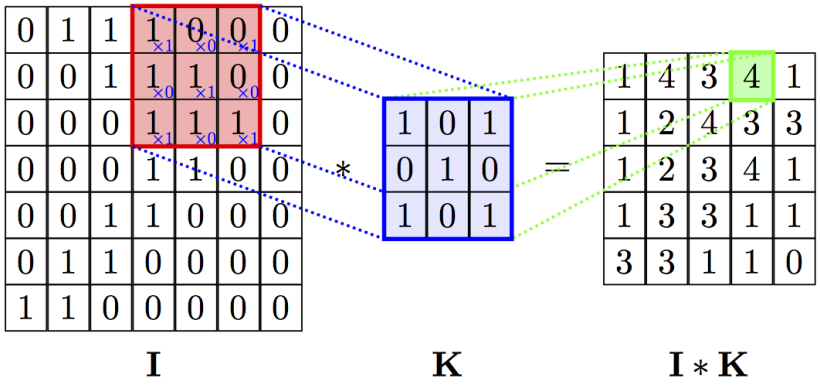


Figure 4: 2D discrete convolutions: the kernel is slid over the input data. For each position, the dot product between the kernel and a particular zone of the input is computed, producing an activation map.

While learning in a convolutional layer consists in finding the optimal kernels, several parameters can be tuned when defining the layer architecture such as the *size of the kernels* (I, J), their *number* n, the *stride* parameter (step size when sliding the kernel over the input) and the *padding* of both the input and the output.

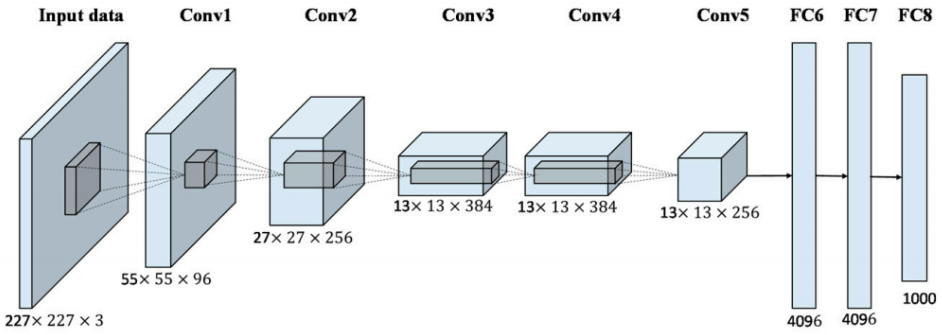


Figure 5: AlexNet architecture: Each convolutional layer is followed by a ReLU. The last linear layer outputs predictions over the 1000 classes available (image from [18])

CNNs are now widely used and several architectures have been proposed, such as AlexNet [19] shown in figure 5. This network built for image classification is constituted with 5 convolutional layers (each one followed by a ReLU activation) and 3 fully-connected linear layers. This illustrates a classic use of a NN after convolutional layers to process the last features maps (after vectorization). We will use this structure when considering our convolutional synthesis model (Sec. 3.2).

2.1.2 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) have the ability to model structured sequential data. To do so, the networks are augmented with recurrent loops, allowing to retain information across time steps. RNNs have proven to be efficient on learning temporal problems, even with application to high-level musical concepts such as chords sequences [20].

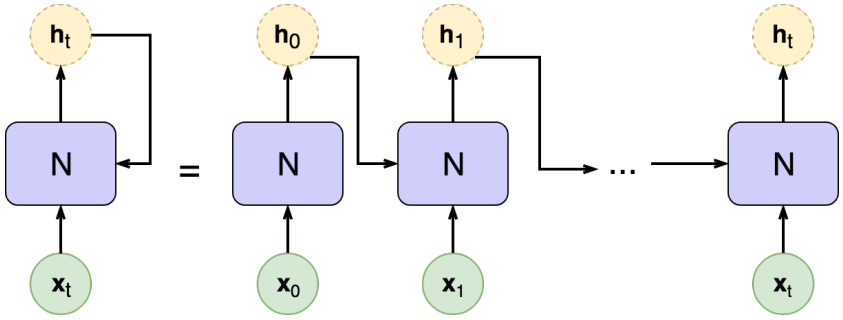


Figure 6: The Vanilla RNN in its folded and unfolded forms. At timestep t, the network is fed with both the input and the hidden state output at time $t-1$

Formally, given a sequence $\mathbf{X} = \{\mathbf{x}_t\}$, RNNs can handle the dependency between elements $\mathbf{x}_t$ by having a recurrent hidden state $\mathbf{h}_t$ whose value at each time depends on both that of the previous time and the input. Hidden states are updated following

$$\mathbf{h}_t = \begin{cases} \phi(\mathbf{x}_0) & \text{if } t = 0 \\ \phi(\mathbf{h}_{t-1}, \mathbf{x}_t), & \text{otherwise} \end{cases} \tag{3}$$

where ϕ is generally a non-linear function. Hence, RNNs are composed with traditional layers as NN that process the input (main blocks in purple in Fig. 6), along with the previous time state $\mathbf{h}_{t-1}$. Therefore, we can see $\mathbf{h}$ as a form of memory since it keeps information from elements at previous time steps and the network use these when processing a new sample.

Thanks to the memory introduced by h, the system has a knowledge of the context and if it has been successfully trained, it should be able to predict the next steps of a given sequence (Fig. 7). Theoretically, RNNs should always be able to perform such tasks, no matter how extended the temporal information is. However, it has been shown that simple RNNs do not scale well to situations that require a longer context because of problems like *vanishing or exploding gradients* [21]. Therefore, simple RNNs are usually bound to perform well only on *short-term* dependencies.

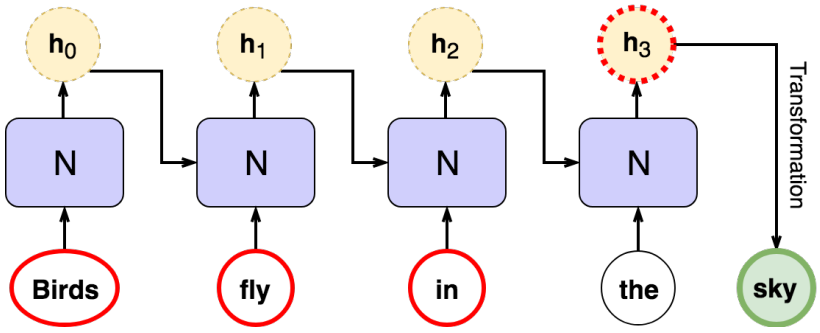


Figure 7: A Vanilla RNN dealing with "short-term" dependencies. The missing final word can be predicted from a quite local context.

To tackle this issue, various specific architectures have been designed, such as the Long Short-Term Memory (LSTM) units or Gated Recurrent units (GRU) [14]. Despite the improvements brought by these architectures, they still fail to model complex temporal tasks where the input has a very high dimensionality. We face this issue when processing audio waveforms because of the high sampling rates. Furthermore, the waveform is made of complex dependencies and at different timescales. Indeed, correlations exist between neighboring values as well as between those that are thousands of samples apart. Thus, some RNN models have been designed specifically to deal with these different timescales (see section 2.3.1).

2.1.3 Supervised or unsupervised

As discussed in previous sections, most machine learning approaches have been designed to solve *supervised* tasks, where we want to approximate known solutions to a problem. Indeed, supervised learning requires the training data to be labeled. For each dataset entry x, a label y^* is associated and this label is precisely what we want to predict with our learning techniques. This is typically applied when one wants to perform *classification* or *regression task*. For example, we can consider the task of instrument classification. If each sound is associated with a discrete label (defining the corresponding instrument), one can perform classification, by learning to recognize the specificities of different sounds. Then, given a new audio sample, we could predict the corresponding instrument.

However, the supervised paradigm requires a large set of labeled examples, which is hard to gather. Furthermore, supervised tasks lead to *discriminative* systems without any generative abilities. On the other hand, *unsupervised* learning is performed when one does not have access to labels (or any type of ground truth). In absence of such labels, we wish to learn the inherent structure of our data. This is typically what we aim at doing for sound synthesis: we want to understand the inner structure of our audio examples in order to be able to create some new ones.

2.2 Unsupervised variational learning

2.2.1 Unsupervised learning and auto-encoders

Auto-encoders are a particular type of networks trained to learn an efficient encoding $\mathbf{z}$ of any unlabeled input data $\mathbf{x}$ [22]. They were originally designed to perform dimensionality reduction on the data. As there is no ground truth for the encoding, auto-encoders are unsupervised models. To perform such a task, auto-encoders are trained to both *encode* the data $\mathbf{x}$ in a code $\mathbf{z}$ and then to *decode* that information to produce an output $\mathbf{x}'$ which is aimed to be as close to $\mathbf{x}$ as possible. Thus, an auto-encoder is always made of two parts: the parametric encoder $\mathcal{E}_\phi$ whose task is to compress the data, and the parametric decoder $\mathcal{D}_\theta$ whose task is to uncompress the code (see Fig. 8).

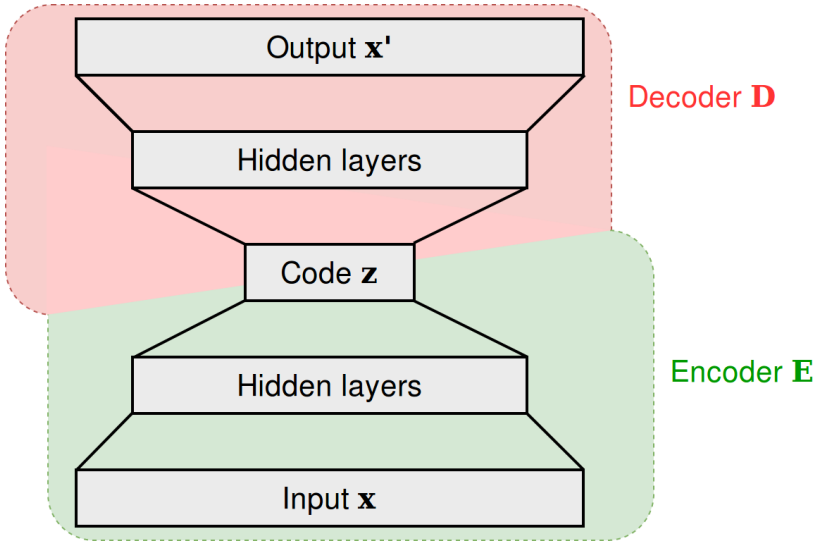


Figure 8: Basic architecture of an auto-encoder. The encoder compresses each input $\mathbf{x}$ to a position $\mathbf{z}$ in the latent space. The decoder then tries to reconstruct the input.

Hence, we can summarize the behavior of auto-encoders as

$$\mathbf{z} = \mathcal{E}_\phi(\mathbf{x})$$
$$\mathbf{x}' = \mathcal{D}_\theta(\mathbf{z}) = \mathcal{D}_\theta(\mathcal{E}_\phi(\mathbf{x}))$$

As we see from the previous definitions, these functions are neural networks parametrized by ϕ and θ (as defined in the previous sections), respectively for the encoder $\mathcal{E}$ and decoder $\mathcal{D}$. Therefore, training an auto-encoder consists in finding the optimal encoding and decoding functions (respectively $\mathcal{E}^*$ and $\mathcal{D}^*$), by performing gradient descent on the difference $\mathcal{L}$ between $\mathbf{x}$ and $\mathbf{x}'$

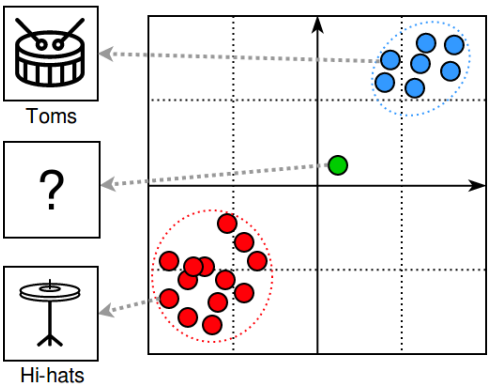
$$\mathcal{E}^*, \mathcal{D}^* = \arg\min_{\phi,\theta} \mathcal{L}(\mathbf{x}, \mathcal{D}_\theta(\mathcal{E}_\phi(\mathbf{x})))$$

where $\mathcal{L}$ can be any reconstruction error function such as the previously defined MSE.

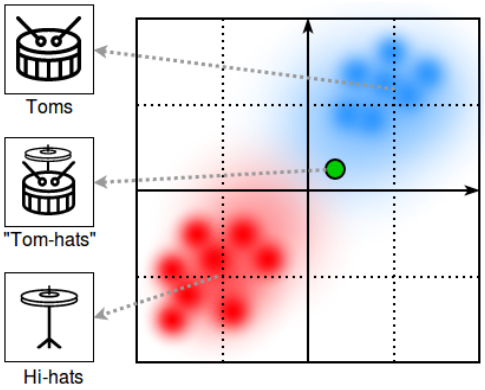
Usually, we want the code $\mathbf{z}$ to be of smaller dimensionality than the input as this acts as an incentive for the network to find the principal factors of variations in the dataset.

This is why we can expect the code to be somewhat expressive about the input data. In the framework of audio synthesis, we could interpret $\mathbf{z}$ as a set of synthesis parameters (or *generative factors*) that have produced this sound. Hence, auto-encoders would seem adapted to our generative endeavor. However, as they are deterministic, there is no guarantee that the resulting code space could allow a robust generalization.

Indeed, since the decoder always relies on $\mathbf{z}$ values that comes from a proper encoding of a given input $\mathbf{x}$, there is no way to ensure that any random $\mathbf{z}$ would generate a meaningful output (see Fig. 9a). We could assume that a latent code close to the ones that generate a hi-hat sound should also generate a sound close to a hi-hat but the deterministic mapping performed by standard auto-encoders is usually not robust to small perturbations [23].



(a) An auto-encoder latent space (2 dimensions). Since the network has never tried to reconstruct a $\mathbf{z}$ that does not directly come from an encoding, it has no generalization property



(b) A potential solution to this problem: each data is encoded as a distribution over $\mathbf{z}$, thus enabling generalization

Figure 9: The problem of generalization in standard auto-encoders

This represents a main issue in our framework as we want to design a generative model. Indeed, our main goal is not to reconstruct existing sounds, but rather to be able to generate novel meaningful data. Therefore, to reach our goal, we would rather like to know which latent codes can generate (decode) a broad variety of a given sound, by mapping each input sound to a distribution $p(\mathbf{z}|\mathbf{x})$ in the latent space rather than a single point. Therefore, we are going to use the recently introduced paradigm of Variational Inference (VI) to tackle this issue.

2.2.2 Variational Inference

Our aim is to design an audio synthesizer based on learning. Here, the type of sounds that we want to generate are represented by dataset entries $\mathbf{x}$ and their corresponding parameters would be the set of latent variables $\mathbf{z}$. Therefore, to learn our parametric synthesizer directly from examples, we have to learn simultaneously an *encoding* distribution $p(\mathbf{z}|\mathbf{x})$ (learning what is the optimal space of parameters) and *decoding* distribution $p(\mathbf{x}|\mathbf{z})$ (generating a sound from a set of parameters). First, we will consider the encoding

distribution $p(\mathbf{z}|\mathbf{x})$, which we can rewrite as

$$p(\mathbf{z}|\mathbf{x}) = \frac{p(\mathbf{x}, \mathbf{z})}{p(\mathbf{x})}$$

The denominator is the marginal probability distribution of the data $p(\mathbf{x})$ (also called *evidence*). Based on the relation between the distribution of latent variables $p(\mathbf{z})$ and the probability of generating the data given the latent variable $p(\mathbf{x} \mid \mathbf{z})$ we could find $p(\mathbf{x})$ by marginalizing $\mathbf{z}$ from the joint probability as follows

$$p(\mathbf{x}) = \int p(\mathbf{x} \mid \mathbf{z}) p(\mathbf{z}) d\mathbf{z}$$

However, this still requires to know how to generate the data given the latent variable $p(\mathbf{x} \mid \mathbf{z})$ and also the distribution of the latent variable $p(\mathbf{z})$ itself (or equivalently the joint distribution $p(\mathbf{x}, \mathbf{z})$). However, for most models, this integral can not be found in closed form or might need a prohibitive time to compute.

For decades, the dominant paradigm for approximating this distribution has been through *sampling* [12], based on a conceptually simple approach. First, we first sample a large number of latent values $\mathbf{z}_i, i \in [1, n]$. Then, we approximate the probability distribution of the data by taking its expectation $p(\mathbf{x}) \approx \frac{1}{n} \sum_i p(\mathbf{x} \mid \mathbf{z}_i)$. However, we can clearly see that the most important issue with sampling approaches is that the quality of the approximation directly depends on the number of sampling operations. But as $\mathbf{x}$ lies in a very high-dimensional space, the number of samples to compute might be extremely large before we have an accurate estimate of $p(\mathbf{x})$. Therefore, the application of sampling methods has been mostly confined to low-dimensional problems.

Another approach to solving this problem has recently been proposed through the framework of ***Variational Inference*** (VI) [24]. The key idea behind VI is to rely on *optimization* rather than sampling. To do so, VI works under the assumption that even though the distribution itself is too complex to find, we could find a simpler approximate distribution that still models the same data, while trying to minimize its difference to the original distribution. Hence, VI approximates a given distribution $p(\mathbf{z}|\mathbf{x})$ by choosing a distribution $q(\mathbf{z}|\mathbf{x})$ from a family $\mathcal{Q}$. Each distribution $q \in \mathcal{Q}$ is an approximation of $p(\mathbf{z}|\mathbf{x})$. In order to find the closest one, we can use a measure of the difference between two distributions, such as the Kullback-Leibler Divergence (KLD) defined by

$$D_{KL}[q(\mathbf{z}|\mathbf{x}) || p(\mathbf{z}|\mathbf{x})] = \mathbb{E}_{\mathbf{z} \sim q}[\log q(\mathbf{z}|\mathbf{x}) - \log p(\mathbf{z}|\mathbf{x})]$$

The best candidate is then found by minimizing the KLD. Thus, the previous integration problem has become an optimization problem formulated as

$$q^*(\mathbf{z}|\mathbf{x}) = \arg\min_{q \in \mathcal{Q}} D_{KL}(q(\mathbf{z}|\mathbf{x}) || p(\mathbf{z}|\mathbf{x}))$$

Once we have found $q^*(\mathbf{z}|\mathbf{x})$, we obtain the best approximation of $p(\mathbf{z}|\mathbf{x})$ inside $\mathcal{Q}$. In order to solve this KLD minimization problem, we can develop this expression [25] by

expanding the conditional probability $p(\mathbf{z}|\mathbf{x})$ and distributing the log, leading to

$$\begin{aligned} D_{KL}[q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}|\mathbf{x})] &= \mathbb{E}_{\mathbf{z}\sim q}\left[\log q(\mathbf{z}|\mathbf{x}) - \log \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{p(\mathbf{x})}\right] \\ &= \mathbb{E}_{\mathbf{z}\sim q}[\log q(\mathbf{z}|\mathbf{x}) - \log p(\mathbf{x}|\mathbf{z}) - \log p(\mathbf{z}) + \log p(\mathbf{x})] \end{aligned}$$

A dependence with $\log p(\mathbf{x})$ now appears. Recall that we do not have access to this quantity and thus it prevents us from computing this objective. However, as $\log p(\mathbf{x})$ does not depend on $\mathbf{z}$, it can be taken out of the expectation and moved to the left member of the equation. Hence, we can obtain the following development

$$\begin{aligned} \log p(\mathbf{x}) - D_{KL}[q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}|\mathbf{x})] &= -\mathbb{E}_{\mathbf{z}\sim q}[\log q(\mathbf{z}|\mathbf{x}) - \log p(\mathbf{x}|\mathbf{z}) - \log p(\mathbf{z})] \\ &= \mathbb{E}_{\mathbf{z}\sim q}[\log p(\mathbf{x}|\mathbf{z})] - D_{KL}[q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z})] \end{aligned}$$

This leads to the formulation of an objective that we can optimize. Indeed, the left side describes the negative divergence that we seek to minimize, plus $\log p(\mathbf{x})$, which is here an added constant as it does not depend on $q(\mathbf{z})$. Hence, we can define

$$\mathcal{L}(q) = \mathbb{E}_{\mathbf{z}\sim q}[\log p(\mathbf{x}|\mathbf{z})] - D_{KL}[q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z})] \tag{4}$$

$\mathcal{L}(q)$ is called *ELBO* (Evidence Lower BOund) because as shown by equation 5, with the KLD always being positive, $\mathcal{L}(q)$ is a lower bound for the evidence $\log p(\mathbf{x})$.

$$\log p(\mathbf{x}) = \mathcal{L}(q) + D_{KL}[q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}|\mathbf{x})] \tag{5}$$

Maximizing this ELBO will solve our original variational problem as we can finally rewrite the optimization problem under the new form

$$\begin{aligned} q^*(\mathbf{z}|\mathbf{x}) &= \arg\min_{q\in\mathcal{Q}} D_{KL}(q(\mathbf{z}|\mathbf{x})\|p(\mathbf{z}|\mathbf{x})) \\ &= \arg\max_{q\in\mathcal{Q}} \mathcal{L}(q) \end{aligned}$$

Finally, optimizing our generative model will amount to optimize the parameters ϕ and θ of these distributions (omitted until here for clarity), leading to the objective

$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}) = \underbrace{\mathbb{E}_{q_\phi(\mathbf{z})}\big[\log p_\theta(\mathbf{x}|\mathbf{z})\big]}_{\text{reconstruction}} - \underbrace{D_{KL}\big[q_\phi(\mathbf{z}|\mathbf{x}) \parallel p_\theta(\mathbf{z})\big]}_{\text{regularization}} \tag{6}$$

Analyzing this expression, we can associate each term with its role :

- The first term $\mathbb{E}_{\mathbf{z}\sim q_\phi}\big[\log p_\theta(\mathbf{x}|\mathbf{z})\big]$ is the likelihood of the data $\mathbf{x}$ generated from the set of latent variable $\mathbf{z}$. Maximizing this is conceptually equivalent to minimizing a *reconstruction error*.
- The second term $D_{KL}\big[q_\phi(\mathbf{z}|\mathbf{x}) \parallel p_\theta(\mathbf{z})\big]$ is the distance between $q_\phi(\mathbf{z}|\mathbf{x})$ and $p_\theta(\mathbf{z})$. It represents the error that we make by using the approximate $q_\phi(\mathbf{z}|\mathbf{x})$ instead of the true latent distribution $p_\theta(\mathbf{z})$. Minimizing this will *regularize* the distribution $q_\phi(\mathbf{z}|\mathbf{x})$ to make it closer to $p_\theta(\mathbf{z})$.

Overall, the model can be optimized by maximizing $\log p_\theta(\mathbf{x} \mid \mathbf{z})$, while regularizing the approximate distribution $q_\phi(\mathbf{z} \mid \mathbf{x})$ to match a prior distribution $p_\theta(\mathbf{z})$. The first term can be obtained through a traditional *maximum likelihood estimation* with any classifier or loss function. However, the second term requires that we define a prior $p(\mathbf{z})$. As we will later want to sample from that distribution to generate some data, the easiest choice would be to choose $p(\mathbf{z}) \sim \mathcal{N}(0, 1)$. Hence, all that remains is to choose the family of variational densities $\mathcal{Q}$ from which to select our approximation.

The mean-field family The complexity of the family $\mathcal{Q}$ determines the complexity of the optimization. One of the most simple and widespread family used in VI is the *mean-field variational family*, where all the latent variables are mutually independent and are each parametrized by a distinct variational parameter

$$q(\mathbf{z}) = \prod_{j=1}^{m} q_j(z_j) \tag{7}$$

Here we can see that each latent z_j follows an *independent* variational factor, defined by $q_j(z_j)$. As the parameters that we aim to model are continuous, we can choose these factors to be Gaussian. This means that each dimension of the latent space will be governed by an independent Gaussian distribution with its own mean and variance. Thus, for each latent dimension we have $q_j(z_j) = \mathcal{N}(\mu_j, \Sigma_j)$. In the VAE, we have more specifically $q_j(z_j) = \mathcal{N}(\mu_j(\mathbf{x}), \Sigma_j(\mathbf{x}))$, where the parameters depend on the input data. Based on this choice, the KL divergence in our objective can be computed as

$$D_{KL}\big[\mathcal{N}(\mu(\mathbf{x}), \Sigma(\mathbf{x})) \parallel \mathcal{N}(0, 1)\big] = \frac{1}{2} \sum_k \big(\exp(\Sigma(x_k)) + \mu^2(x_k) - \Sigma(x_k) - 1 \big) \tag{8}$$

Therefore, we now have a straightforward way of computing the optimization objective. The generic definition of the mean-field family is expressive enough to capture any density of independent latent variables $\mathbf{z}$. However, because of this independence, the mean-field family cannot capture correlations between these parameters [24].

2.2.3 Variational Auto-Encoder

As we discussed in the previous section, we will simultaneously optimize $q_\phi(\mathbf{z} \mid \mathbf{x})$ which *encodes* the data $\mathbf{x}$ into the latent representation $\mathbf{z}$ and a *decoder* $p_\theta(\mathbf{x} \mid \mathbf{z})$, which generates a data $\mathbf{x}$ given a latent configuration $\mathbf{z}$. Hence, this whole structure defines a *Variational Auto-Encoder* (VAE) where both the encoder and decoder are neural networks. On one hand the decoder takes a data vector $\mathbf{x}$ as input but rather than producing a deterministic latent code $\mathbf{z}$, it outputs the pair $\mu(\mathbf{z}|\mathbf{x}), \Sigma(\mathbf{z}|\mathbf{x})$, which are parameters of the multivariate Gaussian probability density $q_\phi(\mathbf{z}|\mathbf{x})$. Then, we can sample from this density to feed the decoder (see Fig. 10).

The decoder is another neural network, whose input is a latent representation $\mathbf{z}$ sampled from the distribution predicted by the encoder. The decoder also outputs the

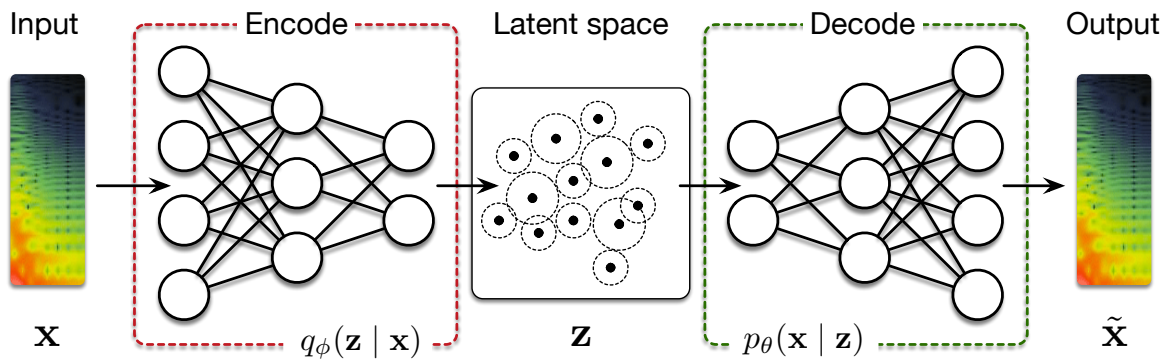


Figure 10: Basic architecture of a VAE. Rather than encoding an input $\mathbf{x}$ as a single position $\mathbf{z}$ in the latent space, the encoder outputs the parameters of the distribution $q_\phi(\mathbf{z}|\mathbf{x})$. The latent space has turned from deterministic to stochastic.

pair $\mu(\mathbf{x}|\mathbf{z}), \Sigma(\mathbf{x}|\mathbf{z})$ which are the parameters of the probability distribution of the data given the latent code. To retrieve the generated data $\mathbf{x}$, it seems logical to sample the mean $\mathbf{x} = \mu(\mathbf{x}|\mathbf{z})$ as it guarantees to generate the sound that is the most likely to correspond to this latent code. Here, we underline the fact that the encoder and decoder can be any neural network model. The resulting whole VAE model will be trained by performing gradient descent with the ELBO (Eq. 6) acting as a loss function.

Re-parametrization trick In order to train the VAE with gradient descent, it should be fully differentiable. However, VAEs features a non-differentiable operation, namely sampling from the distribution $p(\mathbf{z}|\mathbf{x})$. The *re-parametrization trick* [15] was introduced to tackle this issue by observing that we can $\mathbf{z} = \mu(\mathbf{z}|\mathbf{x}) + \Sigma(\mathbf{z}|\mathbf{x}) \times \epsilon$ where $\epsilon \sim \mathcal{N}(0, 1)$. This moves the sampling operation outside of the computational graph, by sampling a set of fixed ϵ, making it fully differentiable (see Fig. 11).

2.3 Generative models for waveform

As discussed in the previous section, the encoder and decoder architectures can be defined as any parametric network. However, the architecture and type of these sub-networks will directly determine the capacity of our system to model different type of data. Hence, as we are interested in modeling audio data, we present some recent models that succeeded in handling waveform data, thus making it adapted to our purpose.

2.3.1 SampleRNN

As explained in Section 2.1.2, the RNNs can explicitly take advantage of the dependences into a sequence. However, they usually fail at modeling long-term dependencies. This problem is particularly crucial when trying to model waveforms. Furthermore, audio

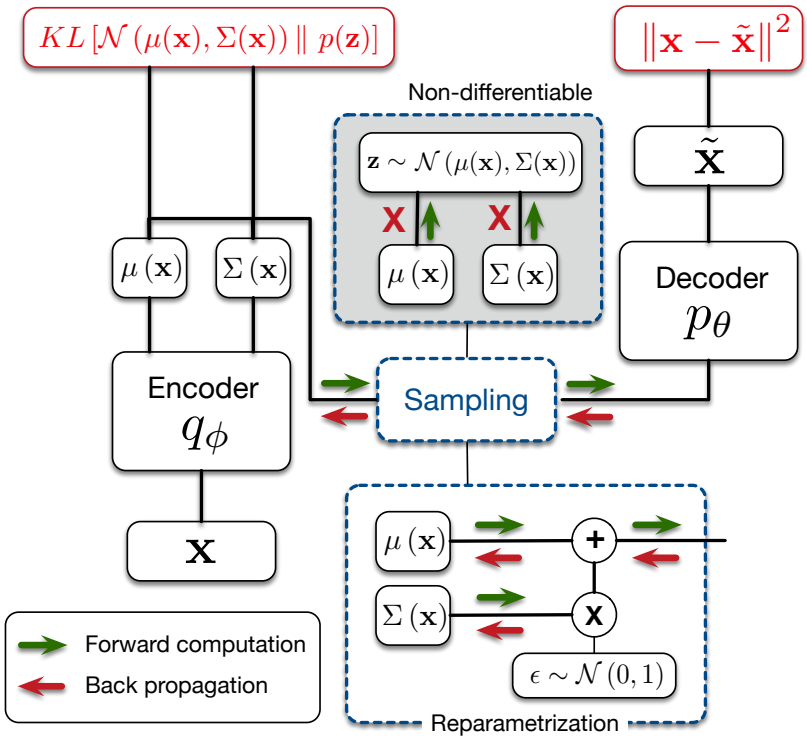


Figure 11: The re-parametrization trick. The error functions which we have to back propagate are shown in red. The upper sampling scheme is not differentiable because the stochastic sampling operation blocks back propagation. Below, the scheme is fully differentiable since the stochastic operation has been moved outside the backpropagation graph

waveforms present an intrinsically *multi-scale* nature. Recently, SampleRNN [3] was proposed to address these issues. This model is defined as a hierarchy of RNNs that operates differently on either an increasingly longer timescale, or a lower temporal resolution, to model longer term dependencies in audio waveforms. To do so, SampleRNN models the probability of a variable-length sequence of samples $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T)$ (which is considered as a random variable over the inputs) as the product of the probability of each sample conditioned on all the previous ones.

$$p(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T) = \prod_{t=1}^{T} p(\mathbf{x}_t | \mathbf{x}_{<t}) \tag{9}$$

Each conditional probability is then modeled by

$$p(\mathbf{x}_t | \mathbf{x}_{<t}) = g(\mathbf{h}_t) \tag{10}$$

We display the overall architecture of SampleRNN in Fig. 12. As we can see, the model introduces three specific mechanisms. First, the lowest modules (in red) are defined as auto-regressive networks that output sample-level predictions. Second, a hierarchy of RNNs are defined with different timescales (in purple), which handles the multi-scale

aspect of the memory. Finally, each module is conditioned by a variable-length portion of the previous sequence (in green).

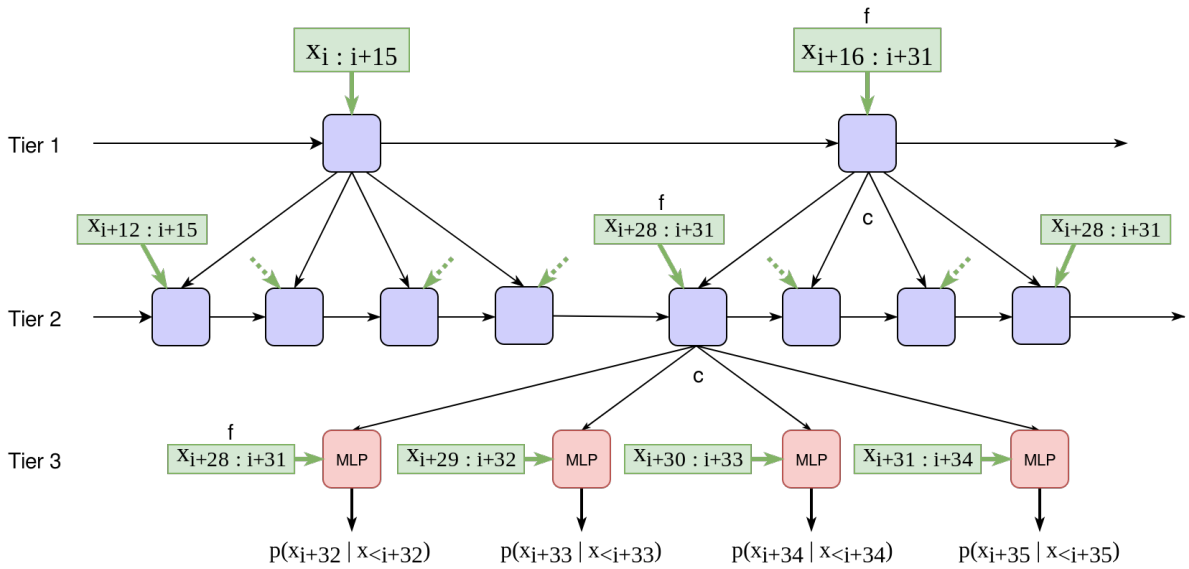


Figure 12: The SampleRNN architecture unrolled at timestep i with $K = 3$ tiers. The model features auto-regressive components (in red) and the higher-level RNNs (in purple). Different scales of waveform conditioning are shown in green.

For the sake of clarity, we are not going to detail the computations of each step, but if the reader is interested in the specificities of the model, we redirect him to the original article [3]. The important aspects to retain from this model are that

- Its hierarchical architecture allows to consider an increasingly large context, while handling the notion of multiple scales of time
- The conditioning mechanism (in green) allows to strengthen the information of the neighboring samples. This represents the fact that even if a larger context is important, the autoregressive behavior is stronger near the sample of interest.

2.3.2 WaveNet autoencoder

Concurrently to SampleRNN, Van den Oord et al. proposed the WaveNet model [2], in order to generate speech waveforms. Interestingly, and unlike SampleRNN, this model does not rely on any recurrent module. However, the model mimics a recurrent by relying on convolutions. Indeed, one of the key concept at the core of the success of WaveNet is the introduction of *dilated* convolutions (see Fig. 13) also called *a-trous* convolutions. The advantage of this structure is two-fold. First, by stacking layers of such convolutions, the model *receptive field* grows exponentially, allowing to model long term dependences that occur in waveforms while having a reasonably efficient structure. Second, this naturally models the multi-scale nature of waveforms and allows to exploit the redundant nature

of harmonic series as a set of convolution kernels. Hence, the autoregressive behavior of this model comes naturally from its convolution operations.

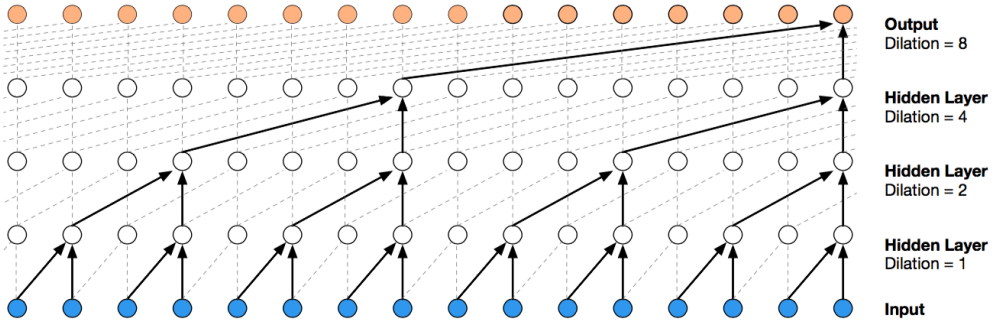


Figure 13: Representation of a stack of *dilated* causal convolutional layers. Here, the dilation factor is multiplied by a factor of 2 at each layer. This allows the receptive field of such stacks to be 2^n, where n is the number of layers (image from [2]).

The WaveNet model was initially developed for speech synthesis. However, Engel et al. later proposed a WaveNet-based autoencoder [26] that extends the purpose of WaveNets to musical sounds generation. This autoencoder is not variational and remains a deterministic mapping. However, because of the humongous amount of data used for training, the latent space is sufficiently dense to generalize for audio synthesis.

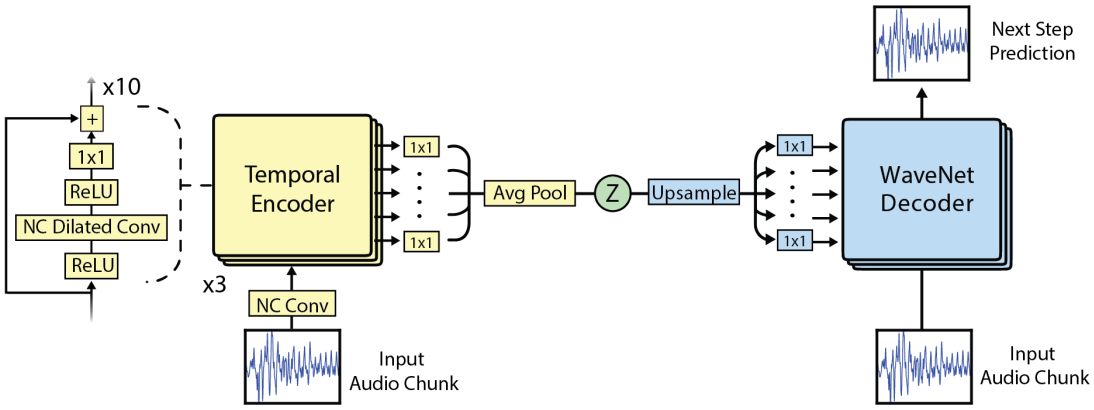


Figure 14: The WaveNet autoencoder (from [26])

As stated in the original article, the authors tried to implement a stochastic latent space through variational learning but the results were not satisfying as the latent code was not used by the model. However, this is a known issue of VAEs [27] that comes from the fact that a decoder with a too large capacity will bypass the latent space. In this work, we will try to implement a VAE based on dilated convolutions, which can be seen as a simplified, variational version of this WaveNet autoencoder.

2.4 Time-frequency representations

Before working with raw waveform data, we will start by expanding existing VAEs approach to generative audio modeling by considering complete time-frequency representations. We will mainly use two representations, namely the Constant-Q Transform (CQT) and the Non-Stationary Gabor Transform (NSGT).

2.4.1 Constant-Q Transform

The CQT [28] is a time-frequency representation based on the Short-Time Fourier Transform (STFT). The STFT for a frame shifted to sample m is defined as

$$X(k,m) = \sum_{n=0}^{N-1} x(n)W(n-m)e^{-j2\pi kn/N} \tag{11}$$

For each bin with center frequency f_k and width δf_k, we can define the quality factor

$$Q = \frac{f_k}{\delta f_k}$$

Hence, we can express the window length for the k-th bin, given a data series sampled at a frequency f_s as

$$N(k) = \frac{f_s}{\delta f_k} = Q\frac{f_s}{f_k} \tag{12}$$

In the computation of the classic STFT, δf_k is constant with k. In the constant-Q transform, as its name states, the quality factor Q remains constant with k. As a side effect, the relative power of each bin will decrease with k as fewer terms will be summed. A normalization by $N(k)$ can compensate for this effect. Finally, equation 12 gives $\frac{f_k}{f_s} = \frac{Q}{N(k)}$ and we are left with

$$X(k) = \sum_{n=0}^{N(k)-1} x(n)W(k,n)e^{\frac{-j2\pi Qn}{N(k)}} \tag{13}$$

This transform is particularly well suited to musical data and convolutional processing. Indeed, the log-frequency scale allows to consider any frequency with the same convolution kernel. Moreover, it is easily computable based on the FFT algorithm. The main drawback of this transform in our context is that it is not invertible. Hence, we are unable to generate an audio signal from this transform without loss. This motivates our use of another invertible, log-frequency transform: the CQ-NSGT. Nevertheless, as the CQT is simpler to compute, we rely on this transform to train our inversion model (Sec. 3.3).

2.4.2 Non-Stationary Gabor Transform

The NSGT can be seen as a generalization of STFT that allows to adapt window size and sampling density in time and frequency [29]. Hence, the CQ-NSGT defines a logarithmic frequency scale with a constant-Q akin to that of the CQT. However, it offers the huge

advantage of being invertible which is essential to our purpose. Hence, we will use this time-frequency representation as the main input to train some of our models, since we will be able to generate data that we can then invert back to waveform. A deeper study of the underlying principles of the NSGT being out of the scope of this report, we refer interested readers to the original article by Holigaus et al. [29].

3 Experiments and results

As discussed earlier, we aim to develop an audio synthesis model for drum sounds based on learning methods. Our main objectives are to obtain a system that can provide a meaningful latent control space, while being able to generate novel audio content in real-time. Hence, in this section, I will first briefly introduce the audio drums dataset that I have created and used throughout this work (Sec. 3.1). Then, I will present the three different models that I have designed (see Sec. 1.5) and the respective results of the different experiments that I have conducted.

3.1 Drums dataset

In order to train learning models, we need a large set of examples of the desired audio distribution. Hence, we first collected a dataset of various drums samples coming from Sony CSL audio database. Overall, we collected more than 8,000 samples across various drum categories. All sounds are WAV audio files PCM-coded in 16 bits and sampled at 44100 Hz. Then, we removed sounds that were longer than 1 second in order to obtain an homogeneous set of audio samples. Indeed, learning models usually require that all dataset entries are of the same dimensionality. Since we want our models to handle temporal information, it is important that the sounds we use are complete and, hence, incorporate the release part.

The final dataset is composed of 6094 samples that we have classified in four classes: *kick drums* (2483 samples), *snares* (1180 samples), *claps* (1110 samples) and *hi-hats* (1321 samples). All sounds in the dataset have a length between 0.1 and 1 seconds (mean of 0.46 second). For all experiments, we have deliberately chosen to train our latent spaces on all classes of sounds so that we can create some 'hybrid' sounds (as depicted in Fig. 9b). In order to validate our models, we performed a class-balanced split between 80% training and 20% validation sets. All results presented are computed on this validation set to ensure generalization.

For each type of model, different preprocessing or transformations to these audio data were performed in order to fit the different requirements of each model. These treatments are detailed in the corresponding sections.

3.2 Convolutional VAE

As discussed in section 1.4, our first objective is to enhance the VAE audio generation model proposed by [11] to incorporate temporal information. In order to avoid the drawbacks of handling time separately, we introduce a convolution-based VAE model (see Fig. 15) able to process an entire time-frequency representation. Hence, this model is a VAE whose encoder and decoder networks are CNNs.

The introduction of convolutions is aimed at providing a uniform treatment of frequency and temporal dependencies. Moreover, convolutions also allow us to work on entire (as opposed to sliced) samples, meaning that we can associate each sound to a single position in the latent space. This approach can provide a simpler user interaction method by making the synthesis process more straightforward.

3.2.1 Data processing

For this model, we will work with time-frequency representations. Therefore, we will rely on the Non-Stationary Gabor Transform (NSGT) computed for each sample in the dataset. First, all sounds waveform data are uniformly zero-padded to be of the same length (1 second). Then we compute the CQ-NSGT of these sounds with a minimum frequency of 30 Hz, a maximum of 11000 Hz and 48 bins per octave. In order to reduce the dimensionality (for memory and computational load optimization), we downsize the resulting transform by a factor of 2 on the time axis. Based on these parameters, we obtain an input dimensionality of $(410, 181)$, 410 being the total number of frequency bins and 181 the number of frames (after downsizing).

Finally, all CQ-NSGTs are rescaled by first taking the logarithm of the amplitude and then rescaling the whole dataset to a zero-mean unit-variance distribution (by subtracting the mean and dividing by the standard deviation). This normalization step has proven to be crucial for learning as other scaling processes have led to less efficient results.

The NSGT being a complex-valued transform, the phase information is usually removed to perform learning solely on the amplitude part. Hence, we first developed a model based on the amplitude information (Sec. 3.2.3). Then, we have tried to learn with the same model by concatenating the amplitude and phase information (Sec. 3.2.4).

3.2.2 Architecture

Our proposed model is based on a VAE with convolutional layers. The encoder is defined as a CNN with l layers of processing. Each layer is a 2-dimensional convolution followed by a batch normalization and a ReLU activation. These layers are followed by traditional fully-connected layers, in order to map the convolutional transforms to a given size d_z of the latent space (leading to a final dimensionality of $2 * d_z$ as we need to model the *means* and *variances* of our distributions). Here we selected a latent space of size $d_z = 128$. The decoder network is defined as almost a mirror to the encoder, so that they have a similar capacity. However, we change the convolution to a deconvolution operation and adjust the parameters so that the output size matches that of the input.

All models have been trained using the ADAM optimizer [30]. The initial learning rate was fixed to 10^{-4} and a scheduler was set to divide the learning rate by a factor of 5 each time the validation loss did not decrease for 100 epochs.

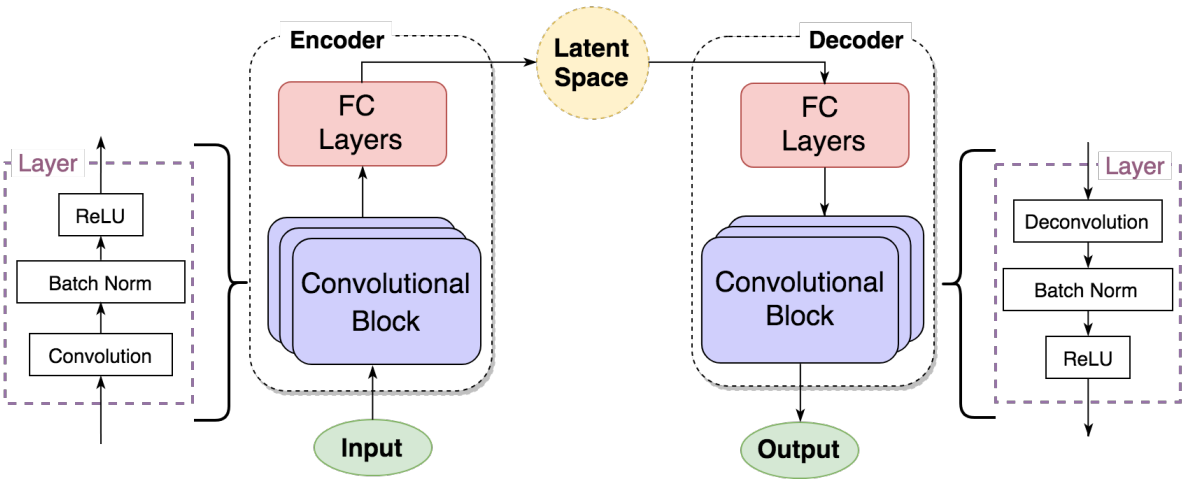


Figure 15: The convolutional VAE model is composed of a convolutional encoder followed by fully-connected layers and a mirrored decoder architecture.

In order to evaluate the impact of different components of our proposed architecture, we tested several parameters configurations (number of layers, number of kernels, stride parameters, capacity of the decoder). We will detail the results and parameters for each architecture in the corresponding sections.

3.2.3 Amplitude only

First, we detail the results based on learning the audio time-frequency distributions while retaining only the amplitude information. Learning is performed following three distinct parameters configurations that are detailed in Appendix A.

First, we analyze the ability of different models to reconstruct the distributions of audio samples from the evaluation set (Fig. 16). As we can see, the VAE is able to perform an accurate reconstruction from different evaluation samples. Hence, we can easily distinguish the kick drum and the clap representations based on the reconstructed frequency content (the clap containing more high frequencies and the kick drum more low frequencies). However, there is a striking difference in the overall blurriness of the reconstructions. This can be explained as the known consequence of using a variational mean-field approximation in the generative model [27].

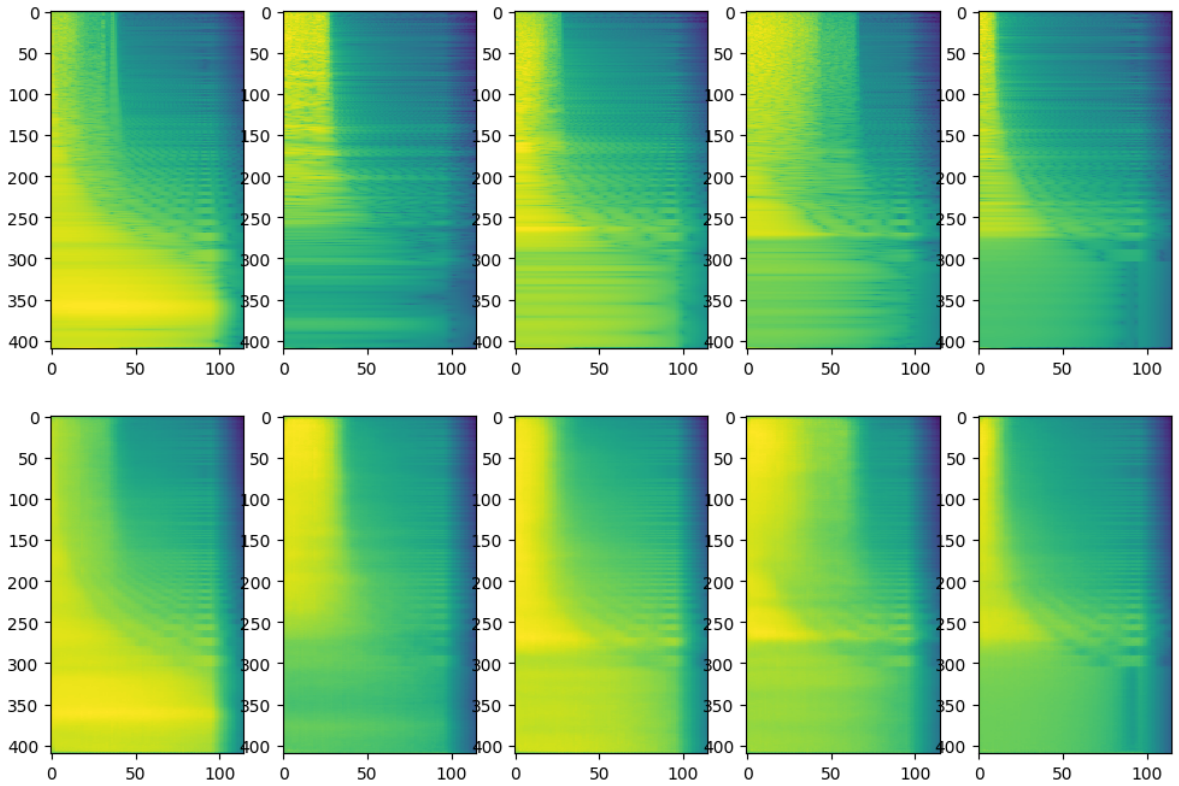


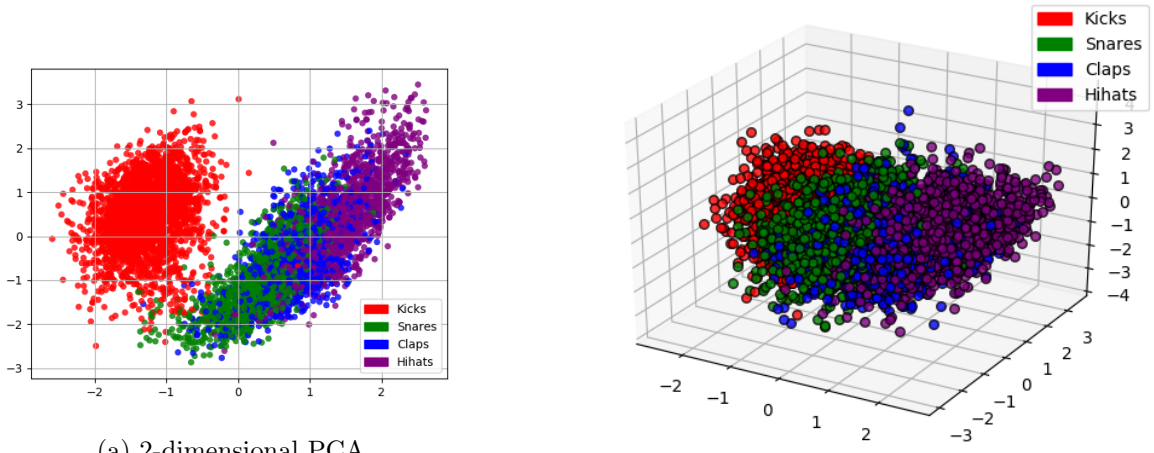
Figure 16: Original amplitude (up) and reconstructions (down) from random entries in the validation set, with configuration 2C-3L-3D. While being blurry, the reconstructions can be considered as good since we can distinguish them from each other and associate them with the original.

In order to quantitatively compare different models, we compute their overall reconstruction accuracies on the whole evaluation set (see Tab. 1). We denote each configuration as a triplet *C-L-D* where C is the number of convolutional layers, L is the number of fully connected layers in both the encoder and decoder, and D is the number of deconvolutional layers. For example, configuration 2C-3L-3D has 2 convolutional enconding layers, 3 fully connected layers in both the encoder and the decoder and 3 deconvolutional layers. Finally, to obtain a reference baseline, we use MSE loss to compare our results with those obtained with a standard deterministic autoencoder using configuration 2C-3L-3D.

As we can see, two of our models obtain better reconstruction results than those obtained with a standard deterministic AE and configuration 2C-3L-3D. Overall, the worse performing model is the 2C-2L-2D configuration. This can be explained by a too low capacity of our model that features only two fully-connected layers. Also, the stride parameter is very high in all of our layers which means that the step used to move the kernels is too large. This stride prevents this model from precisely reconstructing the inputs. The best performing model is obtained with the 2C-3L-3D configuration . Compared to the 4C-4L-4D, this configuration features bigger kernels as well as equally

Table 1: Quantitative evaluation of reconstructions on the validation set (amplitude only)

Configuration	Reconstruction Loss	KL Divergence	Total Loss	MSE
Convolutional AE	-	-	-	360.56
2C-3L-3D	**-116.2**	29.2	**-87.0**	**360.21**
2C-2L-2D	-87.2	27.0	-60.2	360.82
4C-4L-4D	-95.2	**20.3**	-74.9	360.40



(a) 2-dimensional PCA

(b) 3-dimensional PCA

Figure 17: Principal component analysis on the latent space obtained with configuration 2C-3L-3D ($z_{dim} = 64$). The network performs a good discrimination of the 4 classes. Moreover, the x axis seems to encode a feature related to the spectral centroid which is a very interesting feature to use for control.

small stride, which may allow to detect more global patterns, thus making the final audio distributions more coherent. Compared to the standard AE with the same configuration, it obtains better results, as predicted when we introduced VAEs (Fig. 9b).

As discussed in the previous sections, we aim to obtain models which provide both a high quality of synthesis, but also an intuitive latent control space. We recall that this space will be analogous to a synthesis parameters space (as any point in this space can be decoded into a complete time-frequency distribution). Hence, we want this space to be well organized, enabling intuitive exploration. We study the structure of the latent space built with configuration 2C-3L-3D, which provides the most efficient synthesis.

As the original latent space is high-dimensional ($z_{dim} = 64$), we use Principal Components Analysis (PCA) to visualize it. This ensures to have a linear transform of the original latent space that we map to a 3-dimensional space and display the results in Figure 17. As we can see, the classes are well discriminated in the latent space and seem to be intuitively organized. Indeed, the group of kick sounds and hi-hats occupy clear regions of the space, while claps and snares are mixed, which appears logical. Indeed,

	Momentum	m = 0				m = 0.99			
	Iterations	100	200	500	1000	100	200	500	1000
	Time (s)	8.7	17.2	43.9	87.2	8.7	17.2	44.0	87.2
ODG	Kicks	-2.579	-2.578	-2.576	**-2.574**	-2.579	-2.579	-2.578	-2.577
	Claps	-2.575	-2.573	-2.571	**-2.569**	-2.576	-2.575	-2.575	-2.575
	Snares	-2.577	-2.576	**-2.575**	**-2.575**	-2.579	-2.579	-2.578	-2.578
	Hihats	-2.553	-2.546	-2.539	**-2.536**	-2.555	-2.552	-2.543	-2.540

Table 2: Mean generation time and ODG for different GLA hyper-parameters (results on amplitude only). Time is for a CPU computation. Our model uses configuration 2C-3L-3D

the separation between these classes is not easy even for a trained ear, with the snares' spectral centroid usually considered lower than that of claps. Hence, a very interesting result is that the latent space that we learned seem to organize sounds with respect to their centroid. Thus, our model allows to generate hybrid sounds from this space while providing intuitive control to the user.

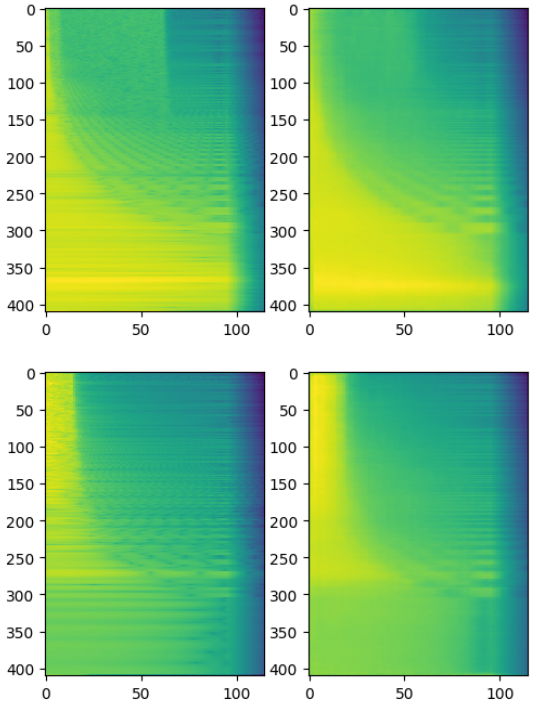
Up to now, our model provide both a good NSGT reconstruction quality and an interesting latent space structure. However our final goal is to generate audio waveforms. Here, because the phase information has been removed, the inversion of the NSGT is performed with the Griffin-Lim Algorithm (GLA) [31]. This algorithm estimates a signal from its amplitude spectrum by first initializing the phase as a Gaussian noise and, then, iteratively refines it through the successive computations of the inverse and forward transforms. Finally, the signal is regenerated by inverse transform of the original amplitude and the latest phase found by the algorithm.

To evaluate the quality of generated audio, we computed the average Objective Degradation Grade (ODG) between the original audio and the reconstruction for random samples from the evaluation set (100 samples per class). The ODG is a score coming from the Perceptual Evaluation of Audio Quality (PEAQ) algorithm [32] which simulates perceptual properties of the human ear and then integrates multiple model output variables into a single metric named ODG which ranges from 0 (imperceptible degradation) to -4 (very annoying perturbation). We compare different configurations for GLA in terms of reconstruction quality but also computation time and summarize these in Table 2. As we can see, our model is able to generate medium-quality percussive audio samples. As predicted, the number of iteration has a role in the quality of the audio samples but apparently, 1000 iterations still are not enough, imposing very long computation times to get high quality audio. To cut this time, we tried to use the fast version of GLA (with $m = 0.99$ as described in [33]), but this does not work in our situation. Therefore, this prohibits the use of our models for a real time application.

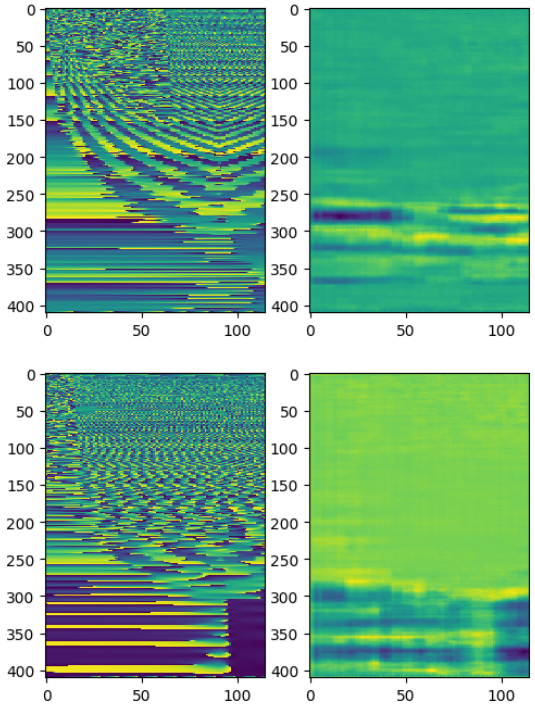
3.2.4 Amplitude and phase

Due to the long computation times required by GLA to obtain high-quality audio from our generated amplitude spectra, we propose to use the same models to learn the phase information jointly with the amplitude. Indeed, this would allow us to apply the inverse NSGT directly on the output of the network and thus to generate the signal in the easiest and fastest way possible. To do so, learning is performed using two groups of convolutions. First, we pass the phase and amplitude through two different convolutional networks (in order to avoid convolving informations of different natures). Then, we join the outputs of these networks before passing the resulting vector through the linear block. This allows us to get a single latent code despite having separated the first part of both trainings. The same principle is used for decoding. Once again, we evaluate several configurations, which are the same as those tested in the previous section.

Looking at the reconstruction plots (Fig. 18), results on amplitude seem to be similar to those with the phase discarded. However, when considering the phase reconstruction, it seems that the model is not able to perform an accurate reconstruction. Indeed, the original phase information varies much faster than the amplitude and, by nature, is very sharp. Hence, the blurriness of the phase reconstruction is way more problematic than that of the amplitude.



(a) Original amplitude (left) and their reconstructions (right)



(b) Original phase (left) and their reconstructions (right)

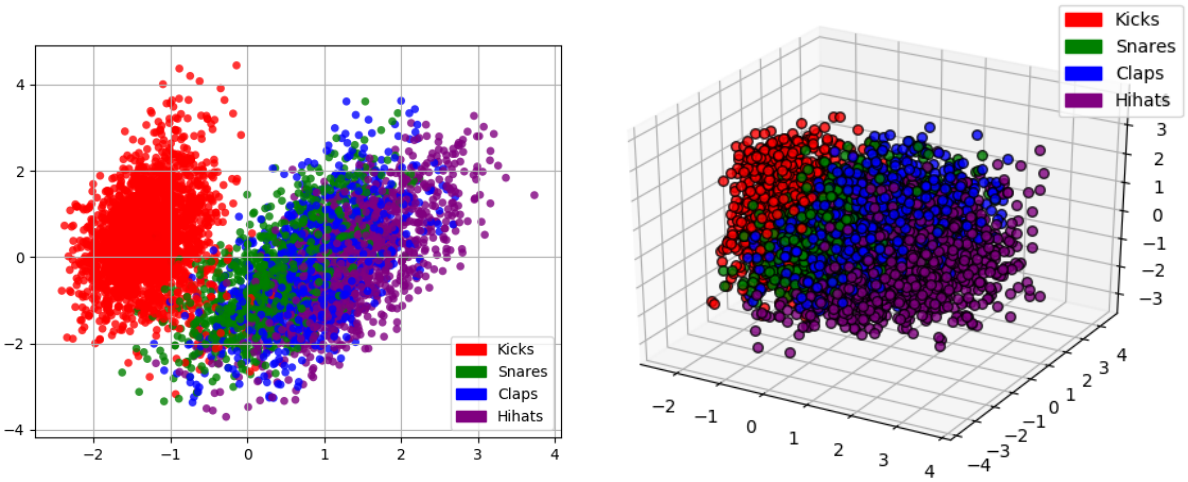
Figure 18: Reconstruction results from the validation set, obtained with configuration 2C-3L-3D. While the amplitude is pretty well reconstructed, the phase information is severely degraded

Configuration	Reconstruction Loss	KL Divergence	Total Loss
2C-2L-2D	165.4	30.6	196.0
4C-4L-4D	168.6	**18.8**	187.4
2C-3L-3D	**153.3**	26.7	**180.0**

Table 3: Average losses on the validation set (amplitude and phase)

Now we compare our models in terms of mean losses. Table 3 presents the mean losses computed from the validation set for each of these configurations.

The loss values are way higher which can be explained by the introduction of the phase information. Nevertheless, the performance order has not changed and the configuration 2C-3L-3D is still the most efficient in terms of average and reconstruction losses. We develop the results it yielded by first studying the structure of the latent space.



(a) 2-dimensional PCA

(b) 3-dimensional PCA

Figure 19: Principal component analysis on the latent space obtained with configuration 2C-3L-3D ($z_{dim} = 128$). This model also performs a decent discrimination between kick sounds and and noisy sounds (hi-hats, claps and snares)

Regarding the latent space, the PCA leads to the same conclusions as before. The phase information seems to have not perturbed the organization of this space and the centroid-related axis is still present.

When it comes to generation, using the blurry phase output by the network yields low-quality samples. Indeed, as said above, the reconstruction is too blurry to represent a coherent phase. Thus, we still need to reconstruct the signal using the Griffin-Lim algorithm. To still try to take advantage of the learning on phase, we chose to initialize the GLA with the phase output by the network hoping that it provides a better initialization

	Momentum	m = 0				m = 0.99			
	Iterations	100	200	500	1000	100	200	500	1000
	Time (s)								
ODG	Kicks	-3.897	-3.900	-3.902	-3.903	-3.902	-3.905	-3.906	-3.907
	Claps	-3.854	-3.856	-3.857	-3.860	-3.892	-3.894	-3.893	-3.894
	Snares	-3.891	-3.886	-3.875	-3.868	-3.902	-3.901	-3.900	-3.899
	Hihats	-3.823	-3.841	-3.855	-3.857	-3.878	-3.885	-3.887	-3.889

Table 4: Generation time and ODG for different GLA hyper-parameters. Here, the initialization of GLA has been done with the phase output by the network.

than gaussian noise. To compare with the previous case, we tested all previous GLA configurations. The results will be shown in table 4 in an updated version of this report

A surprising result is that increasing the number of GLA iterations is worsening the resulting audio. Moreover, comparing these results with those obtained with the Gaussian noise initialization shows that the network-predicted phase does not provide a better initialization. Finally, the phase retrieval problem is leaving us unable to perform real-time audio synthesis. Indeed, the computation time needed to get a convincing audio from GLA is too long. To tackle this issue, we developed a model which aims at performing a faster NSGT-to-signal transformation.

3.3 Transform inversion model

Based on our convolutional model, we are able to learn a latent space that can synthesize novel NSGT distributions. However, we have seen that the phase information is neither learnable nor retrievable in a reasonable time. To tackle this issue, we developed an inversion model aimed at directly estimating a signal from an amplitude spectrum.

3.3.1 Data processing

Because of the complexity of its computation, we have decided not to train the inversion models on CQ-NSGT. Rather, we train the model on CQT whose frames are corresponding to a fixed number of samples (the detailed parameters for CQT computation can be found in appendix). Thus, we construct a dataset composed of joint slices of the CQT and the associated waveform, as described in Figure 20

As shown in Figure 20, the waveform is not encoded in its original 16-bit resolution. As done for SampleRNN [3], we encode sounds with a 8-bit μ-law encoding. This encoding resolution is higher around zero than around -1 and +1. This is more suited to audio than a linear 8-bit encoding since the distribution of samples value is closer to a zero-centered

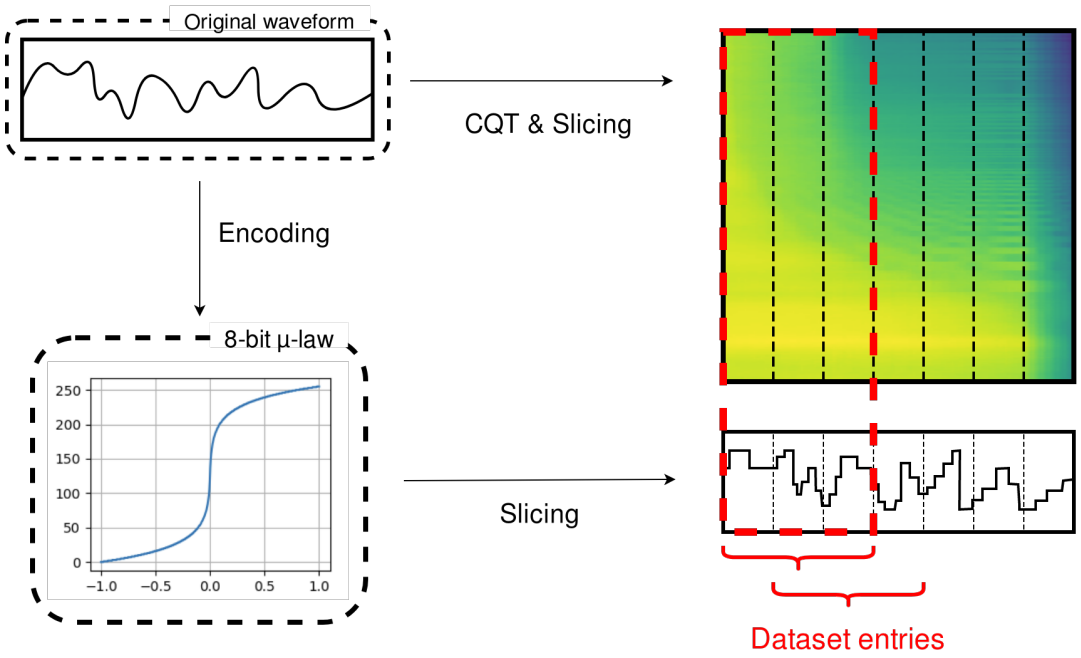


Figure 20: From waveform to dataset entry. The CQT is computed and sliced, producing n_{frames} CQT slices. The waveform is encoded in 8-bit μ-law and sliced in n_{frames}

Gaussian distribution than to a uniform distribution. Finally, one entry in the dataset is composed of 8 CQT frames with the corresponding waveform of 1016 samples.

3.3.2 Architecture

The model we are focusing on is based on RNNs that are hierarchically stacked similarly to the SampleRNN model. First, we designed the model depicted in Figure 21), composed of three RNNs. The frame-level RNN (Tier 1 on Figure 21) processes the 8 CQT frames. The output of this first RNN, which length is equal to that of the CQT sequences is then up-sampled to a 64 vectors sequence, processed by the RNN in tier 2. The output of the RNN in tier 2 is once again up-sampled to be a 1016-long vector. Finally, this last sequence is processed by the sample-level RNN to output a set of 1016 predictions over the 256 μ-law classes. Indeed, because we have encoded our waveform in 8-bit μ-law, the values it takes are integers between 0 and 255 which allows us to perform a classification task rather than a regression task. Thus, the loss function used to train the model is a Categorical Cross-Entropy loss. In our experiments, we have tried to set our RNNs as being either LSTMs or GRUs with a hidden space of size 512.

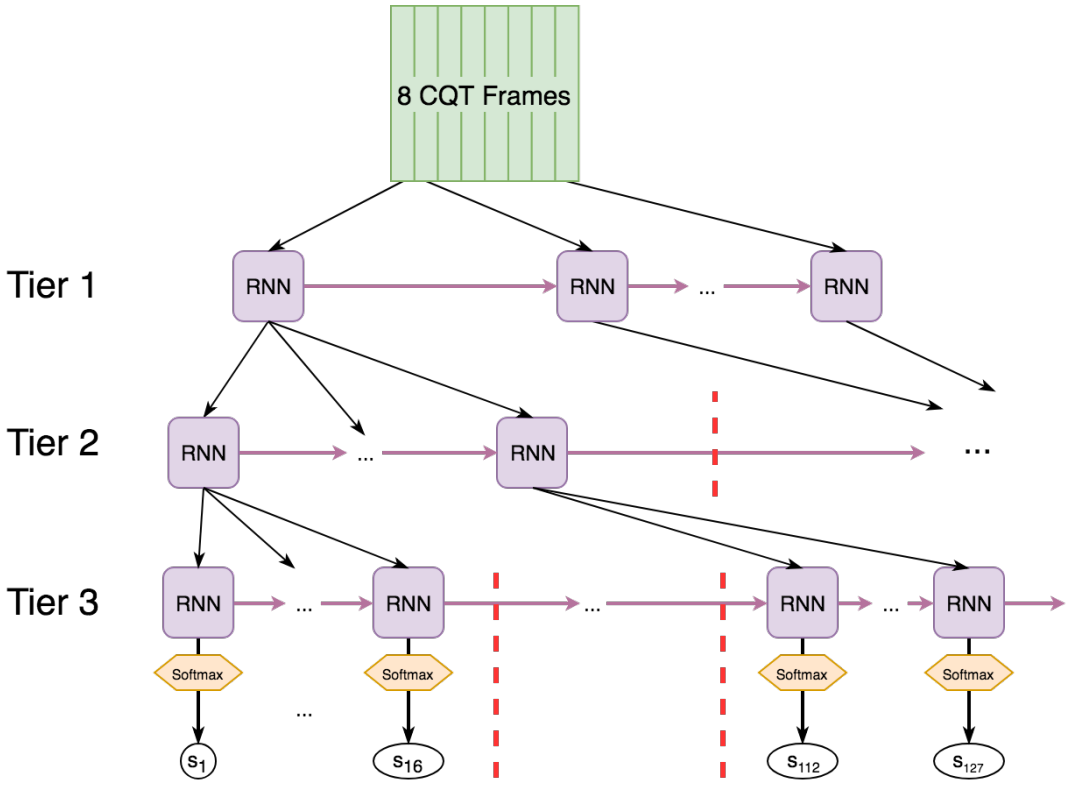


Figure 21: The first transform inversion model. RNNs first process the CQT frames and upsmaple the output. This upsampling step is repeated in tier 2. Red lines denote a change of input for the RNN

3.3.3 Model architecture experimentations

This first model was unable to learn anything and yielded no result with both the training and evaluation datasets. It produced white noise or silence outputs, with a sample prediction accuracy of 0.003, corresponding to a random guess. We then tried many improvements in order to make this model learn. Here, we describe the successive improvements, their motivations, and their effects on the results.

- First, we chose to replace the sample-level RNN with a MLP, as done in [3]. This was notably motivated by the will to cut the long training times imposed by three RNNs. The MLP used is made of two hidden layers of size 1024, followed by ReLUs. This did reduce training time but did not improve accuracy.
- Then, we removed the first RNN layer as we thought that it might be redundant with the second one. Furthermore, we also wanted to strengthen the influence of the CQT information so we added conditioning from the CQT frames at the sample-level MLP by concatenating these frames to the input of the MLP at each time step. Once again, that did not improve the accuracy.
- The next modification we made is to condition each layers with the samples output by the network. We thought it could be beneficial, especially for the last RNN

which thereafter had a knowledge of the context. To do so, the samples we want to condition upon are first embedded through an embedding layer which transforms the 0-255 integer range into a range of 64-dimensional vectors. First, we tried to condition only the last RNN with the three preceding samples. Then, we tried to condition all RNNs with a decreasing number of samples as done in [3]. None of these substantially improved the results. This may be due to the fact that we condition with the samples output by the network, rather than the ground truth. Then, because the network outputs wrong samples in early training, it is difficult for it to learn to use the conditioning information.

- To overcome the issue of conditioning on wrong samples in early training, we have implemented **teacher forcing** [34]. Teacher forcing consists in randomly replacing some samples in the conditioning (that previously came from the output of the network) with those coming from the ground truth. The replacement of one sample has a probability p to be performed. If $p = 1$, the network is exclusively conditioned on ground truth. If $p = 0$ the model is trained as before.

All these modifications led the model to the architecture presented in Figure 22.

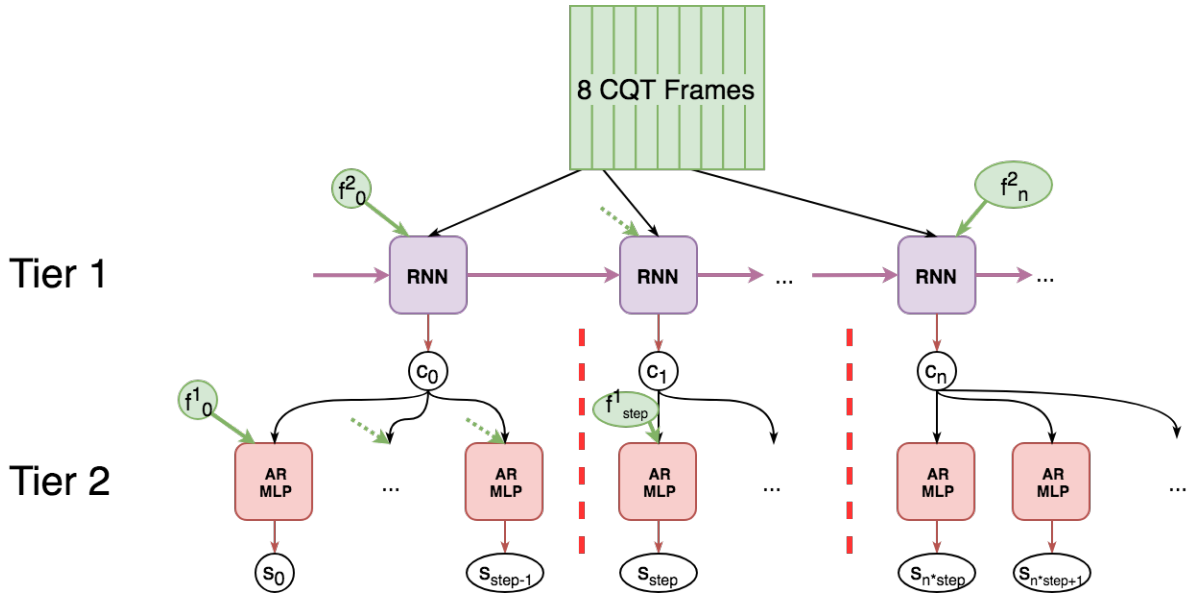


Figure 22: The latest transform inversion model. Green arrows represent CQT and samples conditioning.

The effect of introducing teacher forcing is largely beneficial. With a coefficient $p = 0.5$, the model now reaches approximately 61% accuracy on the training dataset (see Fig. 23). However, no changes have been observed on the evaluation dataset (see Fig. 24). In fact, during evaluation, teacher forcing is disabled since we do not have access to the ground truth of samples values. First, we thought that this asymmetry was the reason for the failure of evaluation results. To counter this, we tried to progressively reduce the teacher-forcing probability p down to zero but this did not yield better results.

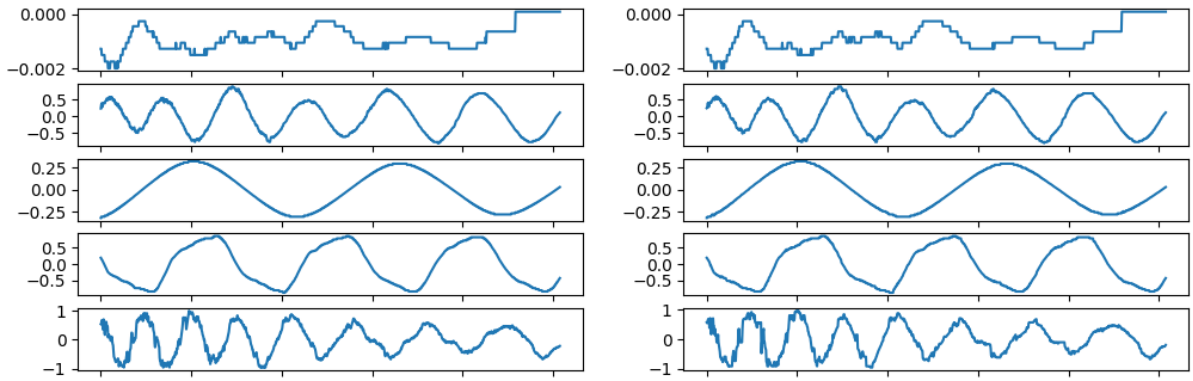


Figure 23: Waveform reconstruction results from CQT in the training set

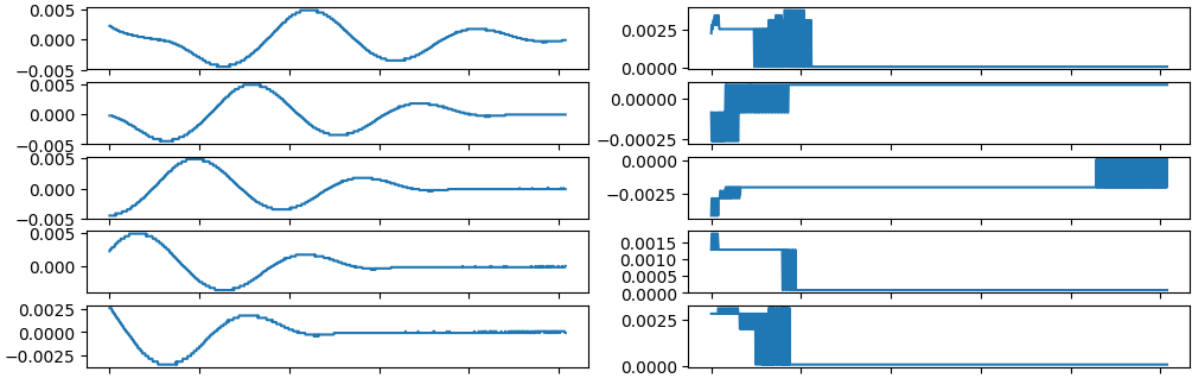


Figure 24: Waveform reconstruction results from CQT in the validation set

This last experiment pointed to the possibility that the network was actually learning to recover samples only in an autoregressive way, without relying on the CQT input. This could explain its overall failure at evaluation time. To test this hypothesis, we decided to train the model while replacing the CQT with an array made of white noise. Thus, if the model was able to usefully use the information from the CQT, its accuracy should have collapsed since we did not provide this information anymore. The results from this last test is that the model fed with wrong CQTs still obtains around 61% accuracy on the training set. This explicitly proves that the network was not using the information given by the CQT. Hence, this explains why our model was unable to provide the same accuracy on the training and evaluation datasets.

3.4 Convolutional-Temporal VAE

As discussed earlier, despite giving convincing results on amplitude generation, learning on time-frequency representations has the drawback of not allowing to generate audio signals in real-time (because of the phase retrieval problem). Hence, our last model aims at bypassing the inverse transformation issue by directly learning from the waveform

information. This would allow learning a control space from which decoding to the waveform would be straightforward, hopefully improving generation time.

3.4.1 Data processing

Because of the high-dimensionality of raw audio, it is impossible to train our model on the waveform without slicing the example files into smaller vectors. Thus, all files are sliced in 1024 samples windows, with a step size of 512 samples. Then, as done in the previous section, we encode all slices in 8-bit *mu*-law encoding, transforming the all waveform windows into sequences of 1024 one-hot vectors. Finally, in order to keep the release of the sounds, the last slices are zero-padded rather than dropped.

3.4.2 Architecture

In the original WaveNet autoencoders article [26], the authors explicitly mention a failure of the variational framework to create a stochastic and robust latent space. The WaveNet decoder actually ignored the latent code and performed the next sample prediction task based on the previous samples only. Indeed, WaveNet decoder is so powerful that it can perform well while ignoring the latent code. However, this is a known issue in variational encoder, discussed notably in [27]. Thus, we have designed a model that tries to use the principle of dilated convolutions in a simpler fashion, in order not to have a decoder with limited capacity. Our model is thus a VAE whose encoder is made of a convolutional block and a hidden FC layer, mimicking the temporal encoder structure found in [26].

The convolutional block of the encoder is defined as a stack of multiple convolutional layers. Each layer is defined as a ReLU, a dilated convolution and a second ReLU. The output of each layer is the addition between the input and the output of the last ReLU (this direct link from the input to the output is called a *residual connection*). The decoder is designed as the perfect mirror of the encoder, with the convolutional layers being replaced with deconvolutional layers.

Inside a block, the dilation factor starts from 1 and is doubled at each layer, as done for WaveNets. Regarding hyper-parameters, our model features 2 blocks of 10 layers in both the encoder and decoder. The kernel size for convolutions is 3, the number of kernels is always 64, so as the embedding size. For the fully-connected hidden layer, its size is 512 and is followed by a ReLU. Finally, the latent space is 128-dimensional.

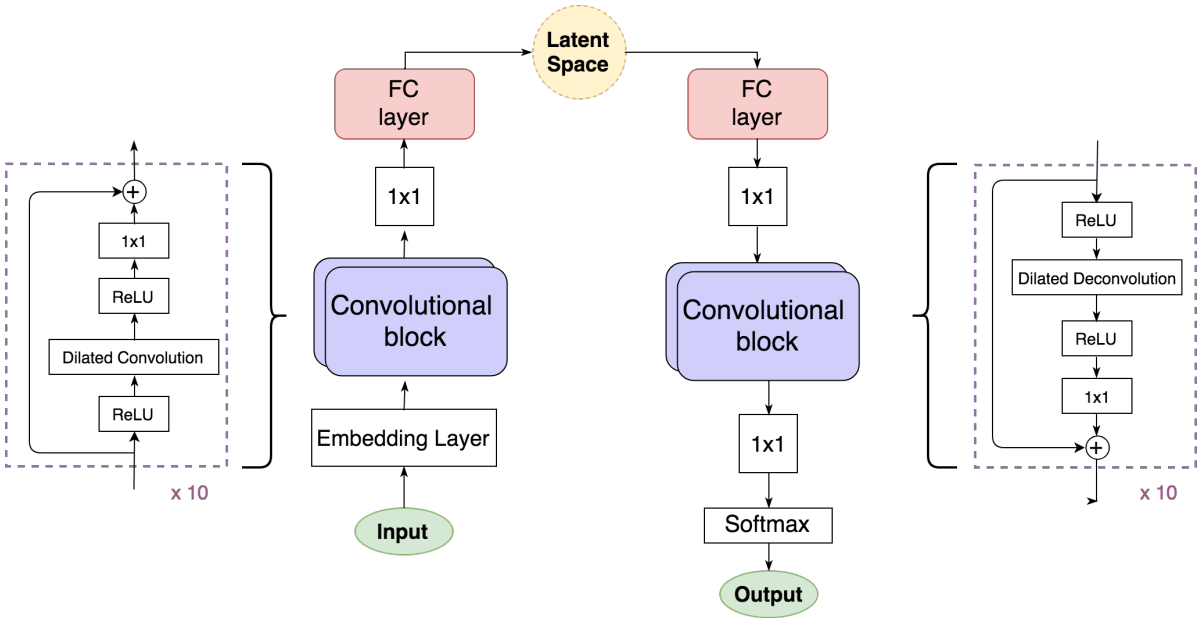


Figure 25: Our dilated convolutions-based temporal model. The stacked convolutional blocks are made of 10 layers each. The 1x1 blocks denote convolutional layers with a kernel size of 1. They are used with only one kernel when we want to linearize data, like the output of the encoder's and decoder's convolutional blocks. With 64 filters, they are used to un-linearize data, for example the output of the FC layer in the decoder.

3.4.3 Results

As in the convolutional model (Sec. 3.2), we first applied our model to learn all types of samples simultaneously (kicks, snares, claps and hi-hats). Indeed, this would allow generating hybrids by interpolating between various types of sound. The reconstructions results from random entries of the validation dataset are presented in Figure 26.

First, we notice that the reconstruction quality is uneven among classes. A closer look at the reconstruction plots shows that the reconstruction is good for the slowly-varying parts of the signal. This excludes the high frequency components and noise. Once again, this could be attributed to the natural blurriness of VAEs we already mentioned. The high-speed variations and randomness in noisy sounds such as claps, snares and hihats make it difficult for the VAE to learn something. Thus, we can see that the slices that are better reconstructed are coming from kick drums. Indeed, they have the lowest frequencies and are quite sinusoidal and not very noisy (apart from the attack). Moreover, as a result, the projections of latent codes in the PCA (Fig. 27) is well distributed for kick drums, whereas for other classes, we observe very a small distribution, meaning that the model struggled to find an underlying structure. Obviously, we cannot expect a decent decoding of this information which seems to have been badly encoded.

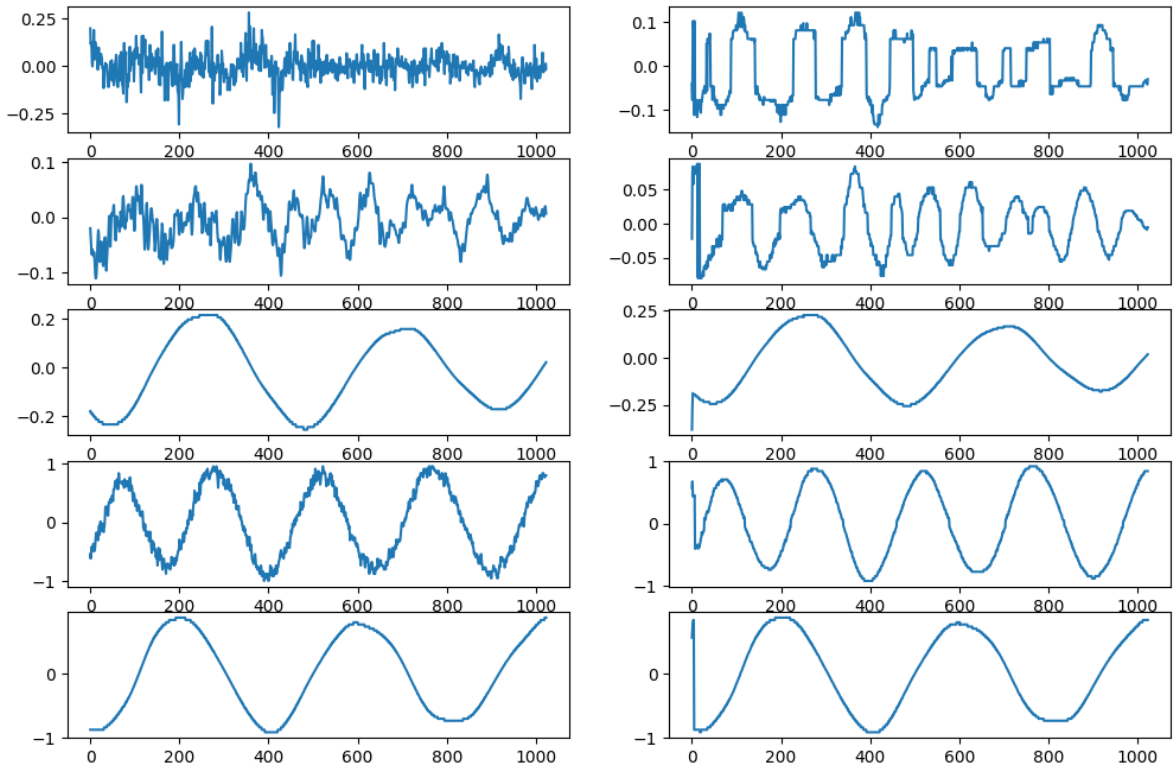


Figure 26: Random waveforms from the evaluation dataset and their reconstructions. Noisy components are less well reconstructed than sinusoidal ones

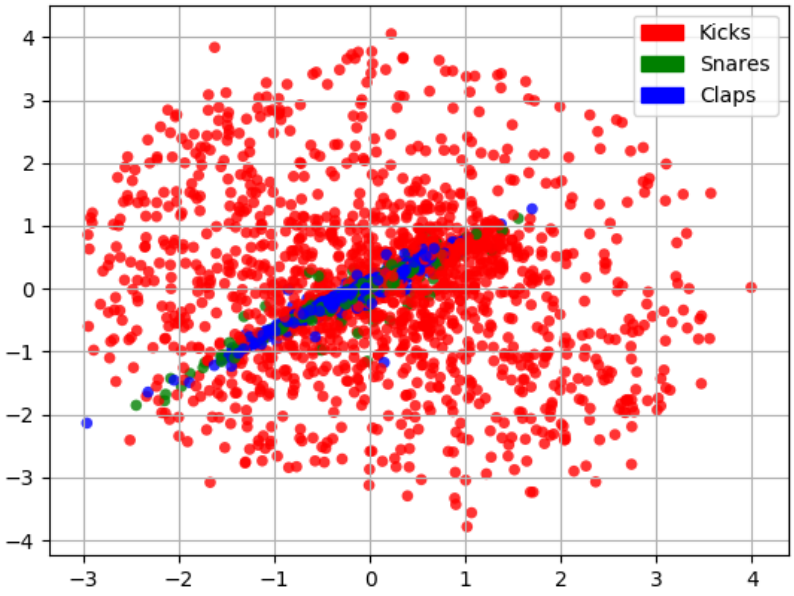


Figure 27: Principal component analysis on the latent space (z = 64). The claps/snares are always projected in the same dense zone, meaning that the encoding is performed badly.

Quantitatively, this model yields 8.5% accuracy on the evaluation set. This must be caused by the noisy entries of the dataset that the network fails to model. Indeed, the training phase is interesting to study: In early training, the network tries to do optimize the output for all situations equally, giving plots similar to Figure 28. Then, the network seems to privilege the reconstruction of kicks (which represent half of the slices) to minimize the error.

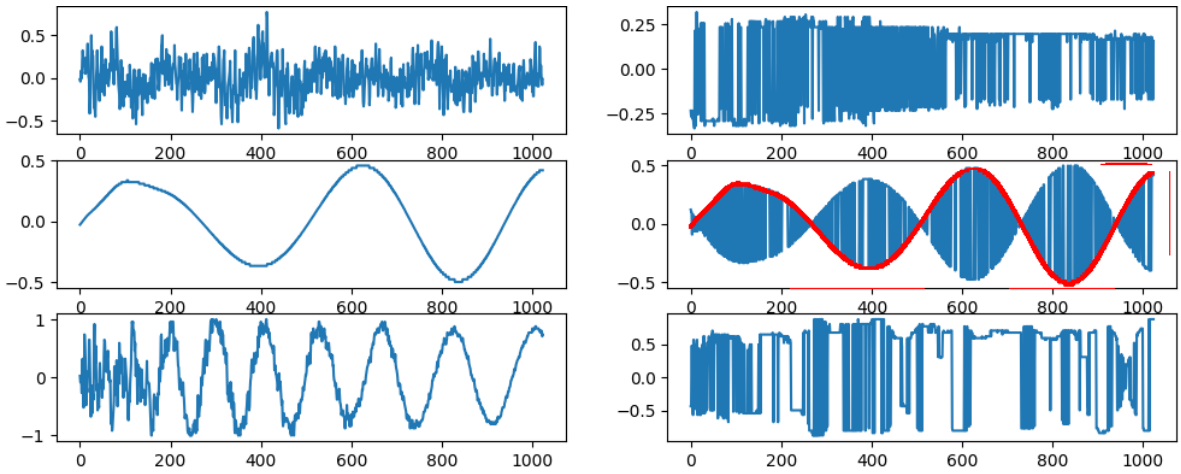


Figure 28: Results on validation set in early training. On the second row, we can see that the network tries to mix high speed variations with the sinusoidal shape, yielding a sort of kick envelope (the red curve is just the original curve (left) that we superposed)

Thus, despite a failure on noisy sounds, this models has proven somewhat successful on kicks. Thus, we have trained a model on kicks only to get rid of the information that could not be learned, thus interfering with training.

3.4.4 Kicks only

Trained on kick drums only, the model now yields 25.3% accuracy, proving that the noisy sounds were causing a drop in the performances of the model. Figure 29 shows reconstruction plots for the model that has been trained on kick drums only.

Looking at the reconstruction plots, it could seem like the model is performing better than 25%. However, the categorical accuracy measure we use does not distinct between a very wrong and slight wrong prediction. Thus, if 232 is the value to predict, a 233 will be considered just as a 50. Therefore, we assume that our model is often very close to the truth, but not exact.

Another interesting remark is that the first sample seem to always be wrong. In fact, having no previous context, the network cannot be able to predict this sample. For future use of the model, we could just delete it.

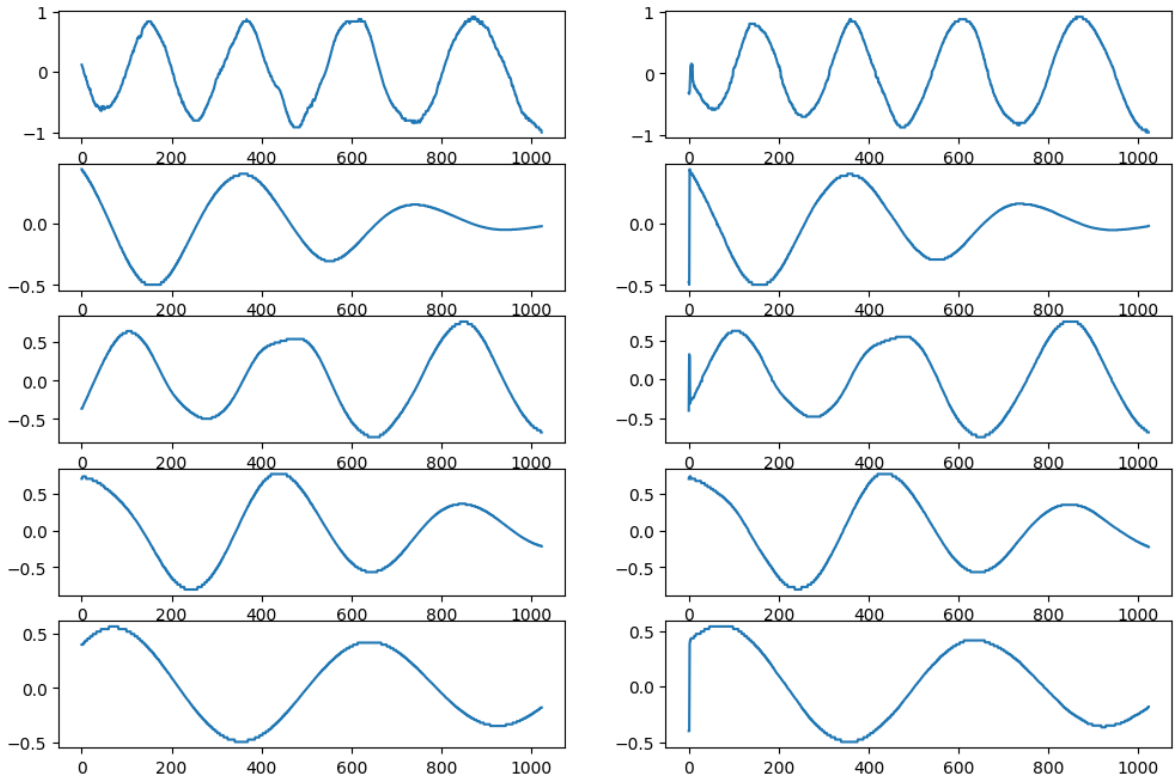


Figure 29: Random kick drum waveforms from the evaluation dataset and their reconstructions.

Through a PCA (Fig. 30), we can see that the latent space is well distributed, with no depleted zone. This is a good result for generation since it almost guarantees to have meaningful decoding from every point in the latent space

A last criteria we use to evaluate the structure of the latent space is how well it discriminates different types of slices (attack, release). To do so, rather than labeling each slice (which could be laborious), we chose to plot the sequence of latent positions corresponding to slices from a single audio file. If a clear mean path appears for different files, we could separate the space in regions containing a single type of slice and recombine a given number of slices from each zone to get a entire sound.

Looking at the plots on figure 31, no particular structure emerges. The trajectories are always located in the middle of the space and not very spread. At first sight, it will not be easy to synthesize coherent sounds as we might place the attack after a slice of release. Another way to think about it is that it greatly expands the recombination opportunities so we are currently exploring some recombination strategies to synthesize from this space, to decide whether this shape is desirable or not.

Sound results are not convincing yet, but we can already say that such a model can produce approximately 110 slices of 1024 samples per second, allowing real-time generation at 22050 Hz.

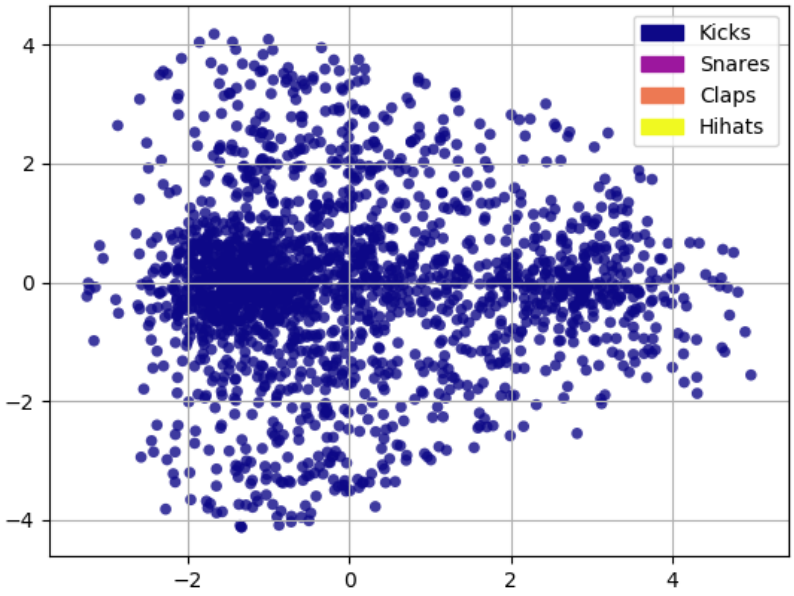


Figure 30: Principal component analysis on the latent space (z = 64). The space is well-distributed

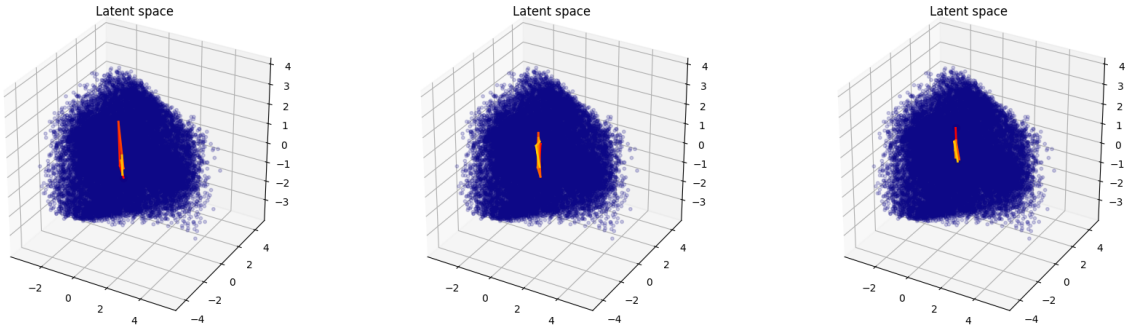


Figure 31: Trajectories representing the sequence of latent points associated with the sequence of slices for three kick sounds. Yellow is beginning, red is ending. Trajectories are always narrow and we can extract no representative mean.

4 Discussion and conclusion

In this internship, our main objective was to develop a generative model for audio synthesis allowing for an intuitive sound synthesis. To design such a model, we took advantage of variational autoencoders that allow to model a sound as a data generated by a space of latent parameters. Also, we benefited from recent advances in the use of convolutional neural networks for audio. Therefore, our contribution consists in the development of two variants of a CNN-VAE model combining both approaches for audio processing and generation.

First, we proposed a model able to process and reconstruct time-frequency representations (namely th CQ-NSGT). The results it yields are very encouraging. Indeed we have shown that it is able to reconstruct and generate high-quality amplitude spectra. Moreover, a study of the latent space structure revealed that the training procedure allowed to extract important perceptual features from these amplitude spectra. This latent space organization, is an essential need for the future development of intuitive user interfaces. However, this first model is limited by its inability to generate a decent phase information. Indeed, amplitude alone cannot be inverted back to the signal domain This implies that generation has to be done through many iterations of the Griffin-Lim algorithm, which greatly increases generation time.

This opens interesting avenues for future work. A first solution could be to generate sharper phase representations, which would allow to generate waveforms through a simple inversion of the complex CQ-NSGT. To achieve this, some papers have shown that changing the training loss of VAEs is beneficial. Indeed, the L2 loss might be mostly responsible for the blurriness of generated data [27].

In this report, we also presented a second variant of our CNN-VAE model which directly processes and generates waveform. This model handles time information through 1D dilated convolutions in a WaveNet fashion. Despite its inability to generate convincing samples of noisy sounds such as claps, this model is suitable for real-time kick drums generation. With more time, we would investigate on interpolation strategies to reconstruct a sound from the latent space. We think that being able to tweak the trajectories could lead to interesting sound generation. Also, forcing the space to clusterize attack or release slices could allow more coherent trajectories and enhance the ease-of-use

From a practical point of view, a comparison between both variants let us think that the first one might be the most promising. Indeed, the fact that we can generate one entire sound from a single point in the latent space is a very desirable feature that enables a quite simple interactivity between the user and the system. Such an approach could lead to the development of user interfaces where parameter knobs are assigned to coordinates in the latent dimensions with the highest variance (e.g. through a PCA). Thus, tweaking the knobs would allow to explore a wide range of new sounds. Moreover, the fact that it can process all types of sounds at once is a great feature leading to the possibility to generate hybrid sounds.

References

[1] M. V. Mathews, "The digital computer as a musical instrument," *Science*, vol. 142, no. 3592, pp. 553–557, 1963.

[2] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv preprint arXiv:1609.03499*, 2016.

[3] S. Mehri, K. Kumar, I. Gulrajani, R. Kumar, S. Jain, J. Sotelo, A. Courville, and Y. Bengio, "Samplernn: An unconditional end-to-end neural audio generation model," *arXiv preprint arXiv:1612.07837*, 2016.

[4] J. O. Smith, "Viewpoints on the history of digital synthesis," in *Proceedings of the International Computer Music Conference*, pp. 1–1, INTERNATIONAL COMPUTER MUSIC ACCOCIATION, 1991.

[5] D. Schwarz, "State of the art in sound texture synthesis," in *Digital Audio Effects (DAFx)*, pp. 221–232, 2011.

[6] K. Karplus and A. Strong, "Digital synthesis of plucked-string and drum timbres," *Computer Music Journal*, vol. 7, no. 2, pp. 43–55, 1983.

[7] G. Eckel, F. Iovino, and R. Caussé, "Sound synthesis by physical modelling with modalys," in *Proc. International Symposium on Musical Acoustics*, pp. 479–482, 1995.

[8] J. M. Chowning, "The synthesis of complex audio spectra by means of frequency modulation," *Journal of the audio engineering society*, vol. 21, no. 7, pp. 526–534, 1973.

[9] C. Roads, "Automated granular synthesis of sound," *Computer Music Journal*, vol. 2, no. 2, pp. 61–62, 1978.

[10] B. Truax, "Real-time granular synthesis with a digital signal processor," *Computer Music Journal*, vol. 12, no. 2, pp. 14–26, 1988.

[11] P. Esling, A. Bitton, *et al.*, "Generative timbre spaces with variational audio synthesis," *arXiv preprint arXiv:1805.08501*, 2018.

[12] C. M. Bishop and T. M. Mitchell, "Pattern recognition and machine learning," 2014.

[13] A. Graves, "Generating sequences with recurrent neural networks," *arXiv preprint arXiv:1308.0850*, 2013.

[14] A. Karpathy, "The unreasonable effectiveness of recurrent neural networks," *Andrej Karpathy blog*, 2015.

[15] D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes," 12 2013.

[16] S. Grossberg, "Nonlinear neural networks: Principles, mechanisms, and architectures," *Neural networks*, vol. 1, no. 1, pp. 17–61, 1988.

[17] R. J. Schalkoff, *Artificial neural networks*, vol. 1. McGraw-Hill New York, 1997.

[18] X. Han, Y. Zhong, L. Cao, and L. Zhang, "Pre-trained alexnet architecture with pyramid pooling and supervision for high spatial resolution remote sensing image scene classification," *Remote Sensing*, vol. 9, no. 8, p. 848, 2017.

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

[20] N. Boulanger-Lewandowski, Y. Bengio, and P. Vincent, "Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription," *arXiv preprint arXiv:1206.6392*, 2012.

[21] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[22] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[23] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and composing robust features with denoising autoencoders," in *Proceedings of the 25th international conference on Machine learning*, pp. 1096–1103, ACM, 2008.

[24] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe, "Variational inference: A review for statisticians," *Journal of the American Statistical Association*, vol. 112, no. 518, pp. 859–877, 2017.

[25] C. Doersch, "Tutorial on variational autoencoders," *arXiv preprint arXiv:1606.05908*, 2016.

[26] J. Engel, C. Resnick, A. Roberts, S. Dieleman, D. Eck, K. Simonyan, and M. Norouzi, "Neural audio synthesis of musical notes with wavenet autoencoders," *arXiv preprint arXiv:1704.01279*, 2017.

[27] X. Chen, D. P. Kingma, T. Salimans, Y. Duan, P. Dhariwal, J. Schulman, I. Sutskever, and P. Abbeel, "Variational lossy autoencoder," *arXiv preprint arXiv:1611.02731*, 2016.

[28] J. C. Brown, "Calculation of a constant q spectral transform," *The Journal of the Acoustical Society of America*, vol. 89, no. 1, pp. 425–434, 1991.

[29] N. Holighaus, M. Dörfler, G. A. Velasco, and T. Grill, "A framework for invertible, real-time constant-q transforms," *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 21, no. 4, pp. 775–785, 2013.

[30] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[31] D. Griffin and J. Lim, "Signal estimation from modified short-time fourier transform," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 32, no. 2, pp. 236–243, 1984.

[32] T. Thiede, W. C. Treurniet, R. Bitto, C. Schmidmer, T. Sporer, J. G. Beerends, and C. Colomes, "Peaq-the itu standard for objective measurement of perceived audio quality," *Journal of the Audio Engineering Society*, vol. 48, no. 1/2, pp. 3–29, 2000.

[33] N. Perraudin, P. Balazs, and P. L. Søndergaard, "A fast griffin-lim algorithm," in *Applications of Signal Processing to Audio and Acoustics (WASPAA), 2013 IEEE Workshop on*, pp. 1–4, IEEE, 2013.

[34] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer, "Scheduled sampling for sequence prediction with recurrent neural networks," in *Advances in Neural Information Processing Systems*, pp. 1171–1179, 2015.

A Model parameters configurations

2C-3L-3D		Input channels	Number of filters	Kernel size	Stride	Padding
Encoder	Conv 1	1 (2 with phase)	16	(15,10)	(5,3)	-
	Conv 2	16	32	(15,10)	(5,3)	-
	FC 1	Input size, output size = 5376, 1024				
	FC 2	Input size, output size = 1024, 512				
	FC 3	Input size, output size = 512, 128				
Decoder	FC 1	Input size, output size = 128, 512				
	FC 2	Input size, output size = 512, 1024				
	FC 3	Input size, output size = 1024, 5376				
	Deconv 1	32	16	(15,10)	(3,3)	-
	Deconv 2	16	8	(20,10)	(3,3)	-
	Deconv 3	8	1 (2 with phase)	(5,5)	(2,1)	-

2C-2L-2D		Input channels	Number of filters	Kernel size	Stride	Padding
Encoder	Conv 1	1 (2 with phase)	16	(20,10)	(10,5)	(2,2)
	Conv 2	16	32	(10,5)	(4,4)	(0,2)
	FC 1	Input size, output size = 1344, 512				
	FC 2	Input size, output size = 512, 128				
Decoder	FC 1	Input size, output size = 128, 512				
	Linear 2	Input size, output size = 512, 1344				
	Deconv 1	32	16	(10,5)	(5,4)	(1,1)
	Deconv 2	16	1 (2 with phase)	(16,9)	(11,5)	-

4C-4L-4D		Input channels	Number of filters	Kernel size	Stride	Padding
Encoder	Conv 1	1 (2 with phase)	8	(10,10)	(2,2)	(1,1)
	Conv 2	8	8	(7,7)	(2,2)	(3,3)
	Conv 3	8	16	(3,3)	(2,2)	-
	Conv 4	16	32	(2,2)	(2,2)	-
	FC 1	Input size, output size = 4000, 2048				
	FC 2	Input size, output size = 2048, 1024				
	FC 3	Input size, ouptut size = 1024, 512				
	FC 4	Input size, output size = 512, 128				
Decoder	FC 1	Input size, output size = 128, 512				
	FC 2	Input size, output size = 512, 1024				
	FC 3	Input size, output size = 1024, 2048				
	FC 4	Input size, output size = 2048, 4000				
	Deconv 1	32	16	(11,8)	(3,3)	(3,3)
	Deconv 2	16	8	(7,4)	(3,3)	(2,4)
	Deconv 3	8	8	(3,3)	(2,2)	(4,6)
	Deconv 4	8	1 (2 with phase)	(3,2)	(1,1)	(3,5)