Audio Music Generation using Deep Learning in an End-to-End Approach

Master Thesis

Aleix Claramunt Molet

Aalborg University Copenhagen
Sound and Music Computing

This document has been designed in LaTeX. The scripts have been implemented using *Python*. For the audio analysis and the plots the *librosa* library has been used. Github has been used to mantain the code. SSH protocol has been used to communicate with the server. 3 Nvidia Titan X have been used to perform the experiments. *Jupyter Notebook* has been used to keep track of the experiments and show the results.

Sound and Music Computing
Aalborg University Copenhagen
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Audio Music Generation using Deep Learning in an End-to-End Approach

Theme:
Deep Learning

Project Period:
Spring Semester 2018

Project Group:

Participant(s):
Aleix Claramunt Molet

Supervisor(s):
Hendrik Purwins

Copies: 1

Page Numbers: 62

Date of Completion:
May 31, 2018

Abstract:

This master thesis wants to address the task of synthesising new sounds using deep learning in an end-to-end approach. That means that when the system is fed with raw audio, it generates new audio samples without any additional information. Although the use of deep learning is quite new in the field, sound synthesis have always seized the interest of researchers. Synthesizers first, and the use of signal processing techniques to model physical systems later, have been studied deeply over the last decades. A breaking point in the field came in 2016 when Oord et al. presented WaveNet [1]. The network used a deep learning architecture to generate one sample at a time when conditioning it by all the previous ones. In this thesis, different architectures have been designed to generate audio samples in an end-to-end approach. WaveNet has been selected over other architectures and a deep exploration has been done. After seeing relevant results by using global conditioning, the network was extended to perform local conditioning. The benefits of local conditioning have been studied, presenting a final tool that is able to automatically distinguish and generate specific piano and panflute sounds conditioning them on the mel spectrum and MFCCs.

Contents

Preface

This report summarises the work done during the last 4 months of my master degree in Sound and Music Computing.

Aalborg University, May 31, 2018

Aleix Claramunt Molet
<aclara16@student.aau.dk>

Chapter 1

Introduction

Nowadays, Artificial Intelligence (AI) is a fast growing field where the number of publications and interests increase every day. Even though it gained most of its popularity just a few years ago, the history of AI started a long time ago. Originally AI was designed to automatically solve problems that were intellectually difficult to humans. Most of this programs could be modelled by applying different rules, and that make a computer more accurate and faster than a human. In contrast, AI was failing in simple human tasks as speech recognition and image detection among others. The problem is that all this data is more complicated and complex and sometimes is hard to find a model. In these cases, the way of passing this information to a computer became the key factor. Due to the impossibility of passing all the information with simple models, machine learning was invented.

Machine learning, was first described as the capability of AI systems to acquire their own knowledge by extracting patterns from data. With this new approach, computers were able to solve more complex problems. In detail, deep learning looks to perform better than other methodologies this task. Deep learning gives his name to the fact that the network is composed by a large amount of different layers. That allows the network to learn different features of the input data in each layer, and tackling the problem with different steps, solve a complex problem. Despite of the popularity of AI in these days, it is important to mention that it appears on the literature in the 40's [2] [3], and became a field of interest in the late 50's [4]. However, in those days, the technology was still not ready to demonstrate the real power of deep learning. 3 key factors helped in establishing Deep Learning in the state-of-the art in almost every field. First, the amount of data available has increased considerably. It has been demonstrated that deep learning architectures work better when using a large amount of training data. The creation of these datasets, however, was not possible until recent years due to advances on technology. That gives us to the second key point: technology. Deep learning have been

characterised by its elevated computational cost. Even these days, training a well optimised network and using last technological tools could take more than days. However, the apparition of GPU's in the market and the decreasing in price of these tools together with others components solved this problem. Finally, with the tools and the data available, deep learning started to be applied in a lot of different fields, moving away from the historical typical problems. The results achieved with most of these algorithms motivate researchers to use deep learning in broader and broader fields [5].

Nowadays, similar architectures could be used to tackle completely different problems. Actually, that is one of the advantages of deep learning over other architectures. With a good dataset it could perform better than more specific systems. One of the types of architectures used in deep learning is called Convolutional Neural Networks (CNN), that owes this name in the way that the system learns. A CNN or DCNN (Deep Convolutional Neural Network) 1.1 are a specific kind of AI in which the system use convolutions between layers to create a model. One of the breaking points in the industry was produced in 2012 [6], when for the first time a CNN won ImageNet [7], the largest contest in image recognition. Is in that moment when CNN started gaining popularity over other architectures and start performing better and better in this field together with others.

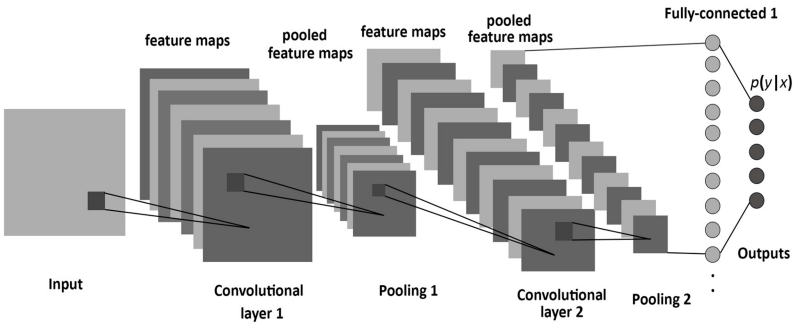


Figure 1.1: Convolutional Neural Network [**Albelwi2017**]

Arts in general are in this group of topics that humans can handle in a very intuitive way, but is hard for computers to learn. Obviously, the reason of that is that when we want to create something new, we need to take in consideration a lot of parameters that we have learnt during our life. There is not an exact model of the information that we use to create something new, it is in that cases that machine learning algorithms could, again, help us. Magenta, a research project led by Google is a good example of how machine learning could be used to create art. All of their demos, that range from creating new musical instruments [8], new ways of creating music [9], [10], [11] or new ways of creating drawing from pictures [12],

are available as an intuitive app hosted in his website. The code for all the projects is also fully available.

Deep learning research is still dominated by computer vision. However, there is not much difference between how computers see images and raw audio. In one case, they see pixels, in the other they see samples. However, in both cases they see an array of limited values with a fixed length. Topics of high research interest as speech recognition, natural language processing (NLP) or music information retrieval (MIR) have been explored with the use of deep learning. The results achieved, created a new state of the art in these fields.

Music generation, as could not be otherwise, has been in the interest of researchers over the years. Although it looks a recent topic, the first computational model for algorithmic composition dates back to 1959 [13]. It is true, however, that not until last years neural networks have demonstrated their ability in audio generation. Lots of new papers has been presented over the last years [14] Some of them working in the audio domain [1] [15] [8] [16], others, working in some audio transformations and finally others working in symbolic sequences as scores [14], [17]. A deep explanation of some of these networks is presented in section 2.

This thesis is centered in the use of deep learning techniques to generate audio in the audio domain, what is also called as an end-to-end approach. In particular, the candidate selected has been WaveNet [1], a new neural network presented by Google Deepmind in 2016 that showed better results than the state of the art methods both in generating English speech and Chinese mandarin speech. This system, that is well described in 3, uses a DCNN to generate one sample when conditioned by all the previous ones. The system also showed the capabilities of WaveNet of learning and generating music. Thanks to the online community, and mostly by Igor Babusckin, an unofficial open source implementation has been released [18].This thesis used this open source implementation as a starting point in order to recreate music in and end-to-end approach. However, due to the lack of details given by the author, the needed time, and the resources needed, the results achieved by this network are not as realistic as the original WaveNet [1]. Nowadays, WaveNet network although it has not been released as an open source software, has been explored by many authors and companies. This architecture, have also been used to solve new problems a part from music generation. Some of them are a Wavenet Autoencoder [8], WaveNet for speech denoising [19], Wavenet for Time Series Forecasting, [20], a music style translator [21], a speaker dependent vocoder [14], a neural text to speech synthesizer [22] among others.

1.1 Motivation and Goals

Seeing deep learning as the current trend in audio synthesis, this project wants to explore some of the available tools and resources. Among different possibilities, WaveNet and specifically Igor Babusckin's implementation [18], has been selected as the tool to explore. The reason to choose this tool has has been motivated by two main factors: First, the results achieved on speech synthesis has been evaluated as more natural and realistic than other current methods. Second, as we commented before, seems that WaveNet is creating a new tendency while working with audio in an end-to-end approach.

This thesis uses Igor Babusckin's implementation [18] as an starting point, as it is accepted as the most "official repository" having 3.658 recommendations on `Github`. The main goal of the thesis is to contribute with the online community with something new. At the very begging of our research some lacks were detected. WaveNet papers [1] [15] came with a bit of secrecy. In spite of presenting the architecture and the results, there are some specific details that were not commented. Examples of that are the number of channels for most of the filter, the number of dilation layers, or the number of iterations. Online community did a big effort on finding the best parameters for that [23]. One of the main drawbacks of Igor Babusckin's implementation [18] is its complexity and its lack of information. Due to that, one of the goals of the thesis is to create a more understandable guide on how to use and how to modify this implementation. Another drawback of this network is that is mostly tuned to generate high quality speech. This leads to work with a large network that took several hours to generate a few seconds of audio when using specially designed GPUs. Several datasets have been presented and a bunch of experiments have been performed in order to understand the behaviour of the network. Finally, Igor Babusckin's implementation [18] had one missing feature. Local conditioning, that is the capacity of the network to learn different phonemes and produce the specific one, was not implemented. In the original paper [1] local conditioning is briefly explained. There is also mentioned that the condition comes from linguistic features in a TTS model. However, there are not more details about how to extract this features. Moreover, local condition seems to be the key components in new implementations of Wavenet as Nsynth [8], MidiNet [24] or the Universal Music Translation Network [21]. Global conditioning (the capability of the system to distinguish among other speakers), instead, was already implemented. Following this approach, and some posts on the Github repository [25], [26] we implemented and tested two kinds of local condition. One conditioned with the Mel spectrum and another conditioned on 12 Mel Frequency Cepstral Coefficients.

1.2 Structure

This project is organised as follows: Chapter 2 deals with the state of the art of using deep learning for audio synthesis. It analyse some of the most popular systems explaining their basic characteristics. Chapter 3 goes in detail in the methods used. Is mostly centred in WaveNet [1] and its unofficial implementation by Igor Babusckin [18]. This chapters also presents Nsynth [8] as it has also been used during this thesis and has become an inspiration for the implementation part. Chapter 4 explains in detail the implementation and the results achieved with different architectures. Finally, chapter 5 concludes the thesis reviewing the most relevant aspects of it. Also some ideas to improve or continue this work are presented during the last part of this chapter.

Chapter 2

Related Works

The use of computers to process sound as humans do has always been a challenging task. Simple tasks as identifying a musical genre, the speaker identity, the structure of a song, or the meaning of a simple sentence have been concerning topics of research. Also, the use of machines to generate music and speech in a natural and realistic way have been a highly explored tasks during the last decades. The techniques used in the field have been recently changed by the popularisation of Artificial Intelligence (AI) techniques. The good results achieved by AI, mostly in image classification and image synthesis, are being used in the audio field giving promising results. In this section, some of the most common approaches and famous architectures will be discussed.

Music, which is the topic chosen for this thesis, has the particularity that could be represented in two completely different ways: first, it can be seen as an audio signal (or any transformation of it) and second it can be represented with a musical score (text, MIDI, piano roll, ABC data, etc.). Both representations can achieve realistic results, always depending on the architecture of the network. In [27] the author presented "The Continuator", a new instrument made use of Markov chains to learn in real time the style of the player and reproduce it at the same time, creating a duet with the musician. In that case, the input data was a midi sequence (pitch, amplitude, velocity and time). Knowing that music could be also represented as a sequence of characters, is logic to think that methods designed to learn and produce long sequences of data would also work with music. An example of this assumption is presented in [28], where a recurrent neural network (RNN) is used to generate new scores. The audio representation used was ABC notation [29], a notation that allows to represent a score using characters. This notation could be easily converted to MIDI or other formats. The RNN chosen was a modification of *char-rnn* created by Andrej Karpathy [30] [31]. In general, a recurrent neural network can be thought of as multiple copies of the same network passing a message

to each neighbour. This architecture makes them a feasible option when the data that has to be processed is intimately related to sequences and lists. However, one of the main drawbacks of RNN is that they don't process well long-term dependencies. A solution to this problem was presented in [32] where Long Short Term Memory architecture (LSTM) was described. In *char-rnn* [30], the author trained an LSTM network to generate text character by character. The results showed that the network was not only able to learn the words, but also the structure of the sentences in a report. A similar implementation of this network was used by Sturm in [28] to generate folk music.

Interesting results have also appeared working in the audio signal domain instead of any kind of music representation. One of the main benefits of working directly on the raw audio signal is that more data is available, and it is easier to find or create a good dataset. However, the complexity of the system and the network is also higher. This is due to the complexity of raw audio. Usually sound is stored in a quantisation factor q of $q = 2^16$ bits and sample frequency f_s of $f_s = 44100Hz$. That is translated to 44100 different samples per one second of audio and 65536 different possible categories per each sample. Although most of the systems try to reduce the input data decreasing q and f_s to achieve realistic results, the system use to be more complex than score representation systems.

In 2016, Oord et. al presented pixelCNN [33] and pixelRNN [34]. Two different architectures that were able to generate new pictures pixel by pixel. In the same year WaveNet [1] was also presented. This network was a reimplementation of pixelCNN in the audio domain. It uses dilated causal convolutions to achieve a large receptive field while mantaining a good compromise with the computational cost. The results achieved with music and speech were better than previous state of the art. WaveNet, that is the architecture used for this thesis and it is well described in section 3, became popular in the field. Different studies that made use of WaveNet are being presented. In [16] Tacotron 2 is described. It is a network that directly translate text to speech by using a modified version of WaveNet conditioned on mel-scale spectograms. Deep Voice [22] also used a variant of WaveNet to generate speech in an end-to-end approach. The system consisted of 5 blocks (a segmentation model for locating phoneme boundaries, a grapheme-to-phoneme conversion model, a phoneme duration prediction model, a fundamental frequency prediction model, and an audio synthesis model) and a reduced version of WaveNet was used for the synthesis part. The results showed realistic results while reducing the generation time over existing implementations by a factor of 400. Another use of WaveNet is presented in [19]. Here, the author uses non causal dilated convolutional layers with a dilation of three to remove the noise of a signal. An experiment with 33 participants and 20 audio samples showed that WaveNet was

preferred over other common denoising methods. Nsynth [8] used a conditioned WaveNet together with a WaveNet encoder to create new sounds in a similar way than analogue synthesizers do. Nsynth has been also explored during this thesis and is presented in section 3.

The use of RNN in end-to-end audio approach have been also explored. Assuming that WaveNet it is an specific implementation for audio of the pixelCNN, a logic step was to explore how pixelRNN would work in the audio domain. With this idea, sampleRNN [17] was created. As we commented before, RNN are good to model sequential data, but use to don't scale well at high temporal resolutions. The authors solve this problem creating a network formed by different modules each operating at a different temporal resolution. The lowest module processes individual samples, and each higher module operates on a longer timescale and a lower temporal resolutions. Each module also conditions the module below it, with the lowest module outputting sample-level predictions. Trained in three different datasets, results showed that sampleRNN was preferred over other methods, including a reimplementation of WaveNet.

Chapter 3

Methods

3.1 Wavenet

Wavenet [1] is a Deep Convolutional Neural Network (CNN) that is able to generate new audio wave-forms operating directly on the raw audio domain. It was released in 2016 by Oord et. al, members of the Google Deepmind. Before WaveNet, pixelCNN [33] and pixelRNN [34] were presented. Both architectures achieved realistic results in the image synthesis field. Both systems uses all the previous samples to predict the current one. WaveNet [1], used this approach to generate audio. Generating one sample at time while conditioning it with all the previous ones it achieved more natural and realistic results than the best current available systems on the market. Not only thanks to this results, but only to the disruptive methods used, a large number of researchers explored the opportunities that WaveNet architecture could benefit in other topics.

Although WaveNet was mostly designed for speech synthesis, in the first paper [1] also showed the possibilities of using it for music synthesis. A set of new generated piano samples were presented. Despite no perceptual evaluation was conducted for the music samples, it showed to the community the possibilities of using CNNs for audio synthesis.

The methodolody used in WaveNet has been considered a breaking point in speech synthesis for different reasons. Taking a look into popular Text-To-Speech (TTS) techniques, can be seen that there exist mainly two different approaches: Concatenative Systems and Parametrical Systems. On the one hand Concatenative Systems, make use of large databases containing short speech fragments that are combined between them to create an utterance (word or vowel sound). Even though the result achieved with this methods is more natural, is still difficult to modify some parameters of the generated signal (speaker identity, emphasis, emo-

tions, etc.) without recording a whole new dataset. Parametrical Systems, on the other hand, use different parameters that contain information about the desired output and are used to control the model, usually formed by different signal processing algorithms as the Vocoder. In these cases, the stiffness given by Concatenative Systems is solved, but its sound tends to be less natural. [35]. The use of machine learning techniques combined with TTS techniques is also present in the literature. However, WaveNet presented a new state-of-the art in two key points: first, because it works directly on raw audio, withouth any external dependencies. Second, because it uses a deep CNN instead over architectures more common in the literature as RNN.

Wavenet, moreover solved some of the drawbacks of Parametrical Systems and Concatenative Systems. Firstly, the perceptual evaluation performed in [1] showed that the generated speech was even more natural than Concatenative Systems. Secondly, by training the network with different speakers and using global conditioning, the generated speech could be controlled in several domains. However, WaveNet also has its own drawbacks.

One of the main weaknesses of WaveNet was his high computational cost. Due to its large receptive field 1 second was needed to generate 0.02 seconds of audio in specially designed GPUs [1]. However, the authors focused on solving this problem and in 2017 they presented a new WaveNet architecture Oord2017 that could parallelize most of its operations. The generation time was reduced by a factor of 1000 times, making it possible to generate 20 seconds of audio with just one second. Finally, this year Google adopted WaveNet in his voice assitant projects, making it work in real time [36].

Even the big technological advance that WaveNet suppose for the field, Google didn't publish any open version of it, and the available information was very limited. Due to that, the community, and mainly Igor Babuschkin (ibab), implemented a similar neural network from the available papers [18].

Igor Babuschkin's implementation is difficult to follow from a novice perspective with the available literature. Even it is as close as possible to the original implementation is difficult to relate the original paper [**oord2016**] and the open source implementation [18]. Due to that, in the following sections first the architecture from the original paper [1] will be presented. Later, Igor Babuschkin's implementation will be explained in detail while referencing also to the original paper.

3.1.1 Original Network

Wavenet is presented as a fully probabilistic and autoregressive model where each audio sample is conditioned in all the previous ones. In that way, they treated the problem as if it was a classification problem. The joint probability of a waveform $x = x_1, ..., x_T$ is factorised as a product of conditional probabilities as:

$$p(x) = \prod_{t=1}^{T} p(x_t | x_1, ..., x_{t-1}) \tag{3.1}$$

This conditional probability distribution of each sample could be learned by a neural network. To understand the overall network is useful to divide the system in 6 different sections: Pre-processing, dilated causal convolutions, gated activations units, residual and skip connections, post-processing and conditioning. A full overview of the network could be seen in figure 3.1

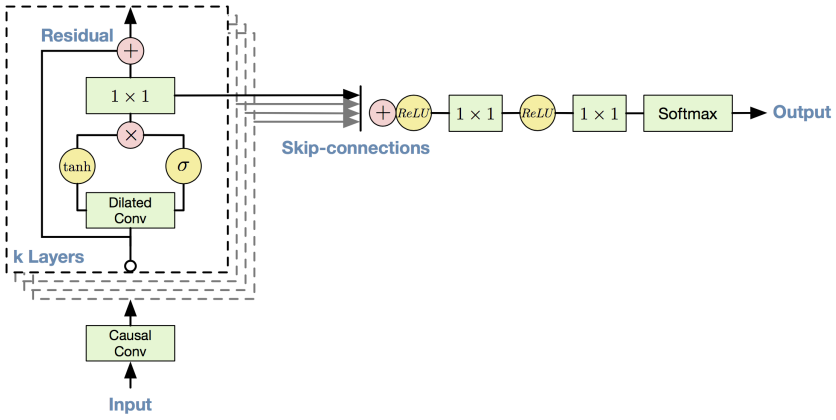


Figure 3.1: Overview of the residual block and the entire architecture [1]

Pre-processing

Typically raw audio is generated with a sampling frequency of 44100 Hz and a quantization of 16 bits. The sampling frequency came by the human hearing range (20Hz - 20000 Hz) and the *Nyquist-Shannon sampling theorem* that says that in order to avoid aliasing the sampling frequency has to be at least the double of the frequency that has to be reproduced. However, not all the audio signals work in this range. Speech for example tends to go from 300 Hz to 3400 Hz. Due to the significant cost of working with this amount of samples, the first step was to reduce the sampling frequency from the input. Researchers decided to use a sampling frequency of 16000 Hz instead of 44100.

When converting an analog signal to a digital one a quantization factor has to be decided. The quantization, is the number of fractions that the amplitude of the signal is divided. A common convention is to use 16 bits to code each possible value. That means that the total possible values per each sample are $2^16 = 65536$. As we commented before and could be seen in equation 3.1 each sample is conditioned by all the previous ones. That means, that this value that comes from the quantization will be used to predict all the following samples. Due to that, one could understand that predicting one value among 65536 values, could not be an optimal option. However, reducing the quantization of a signal also reduce the quality of it. A compromise between the quality of the generated signal and the quantization channels has been found by the researchers in 9 bits. This, reduce the possible values to 256. Is also important to mention, that even it would look like regression problem, the problem is tackled as a classification problem. That means that each new sample, could be seen as a category to predict while having all the previous samples as a features. This reduction, then is more important than ever, as the network has only to distinguish among 256 categories instead of 65536.
In order to reduce the number of quantization channels, Oord et al. decided to a apply a μ-law transformation [37] to the data and later quantize it to the 256 possible values. Researchers found that this non-linear transformation produced significantly better results than linear quantization methods. A μ-law transformation is computed as:

$$f(x_t) = sign(x_t)\frac{ln(1+\mu|x_t|)}{ln(1+\mu)} \tag{3.2}$$

where -1 < x_t < 1, μ = 255 and sign represents the Sign function.

Finally, one-hot encoding is applied to each sample. One-hot encoding consist in turning this 256 values into a form that would be understood better for the neural network. In detail, each category will be represented by an array of 0's and 1's as showed in table 3.1.

Category	**One-Hot Vector**			
0	1	0	...	0
1	0	1	...	0
...				
255	0	0	...	1

Table 3.1: One-Hot Encoding Example

An array formed by all this concatenated one-hot encoded vectors would be the data used to train the network, and the data generated by the network. At the

end of all the process, this one-hot vector will be decoded again to create an audio waveform.

Dilated Causal Convolutions

Dilated causal convolutions are the main ingredient of WaveNet. The word *causal* means that the network will only look at the previous and the current samples to predict a new one (as a causal filter) [20]. In this specific case, the use of causal convolutions is justified because the output is not known. That means that is not possible to use a future sample inside the convolution. However, other architectures (i.e. WaveNet for speech denoising [19] or even during the training process, where the future samples are known, the use of non-causal convolutions will also work. In figure 3.2 two WaveNet architectures are presented. The first one shows a network consisting of a stack of causal convolutional layers [1]. The image below presents re-implementation of WaveNet using non-causal convolutions [19]. The architecture also differs in the dilation factor. First figure doesn't have diltions layers. On the other hand, the architecture presented in Pons2018 has a dilation factor that increase at the power of 3.

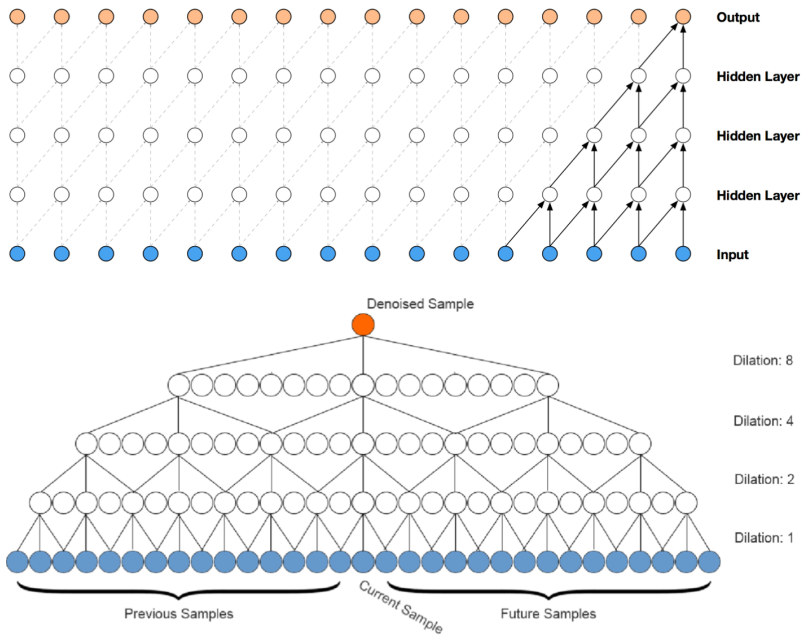


Figure 3.2: Top: Network with 4 layers of Causal Convolutions [1]. Bottom: Network with three dilated non Causal Convolutions [19]

The word *dilated* means that the filter is applied by skipping certain elements in the input [20]. That allows the network to create a big receptive field that grows

exponentially with less layers than non-dilated networks. One of the main advantage of CNN over RNN is that they are typically faster to train as they don't need recurrent connections. However, when a big receptive field is needed so many layers are also required. The fact of adding more layer to the network increase considerably the computational cost. Dilated convolutions tends to solve this problem by assuring a big receptive field with just a few layers. Also, the input resolution through the network is well maintained. In figure 3.3 we can see the same network as in figure 3.2 but when dilated convolutions applied. If we compare figure 3.2 (top) and figure3.3 we could see that the receptive field of the non-dilated network is only 5. For a non-dilated convolution the receptive field could be calculated as:

$$r = n + l - 1 \tag{3.3}$$

where n is the number of layers and l is the filter length. However, using dilated convolutions the receptive field of the output grows up to 16 while keeping the same number of layers. In this case the receptive field size can be calculated as:

$$r = (l - 1) \cdot \sum_{i=1}^{n} d(i) + 1 \tag{3.4}$$

where l is the filter length, n is the number of layers, i is the current layer and $d(i)$ reffers to the dilation factor of each specific layer.

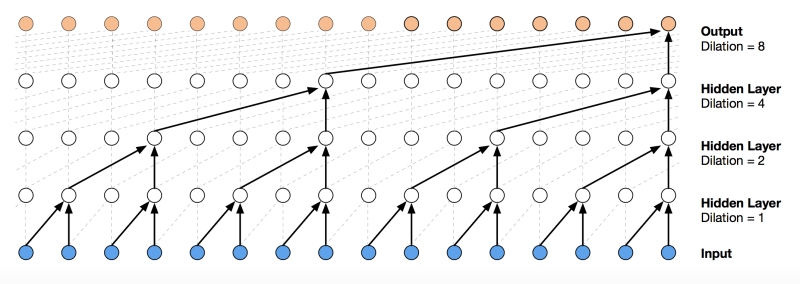


Figure 3.3: Top: Network with 4 layers and dilation of 2[1]

The paper proposes to start with a dilation of 1 and increase it exponentially by a factor of 2 every time until we arrive to 512. Later, repeat this several times. The network created then, will look like this: 1,2,4,...,512,1,2,4,...,512,1,2,4,...,512. Applying equation 3.4 the receptive field for each block would be $1 + 2 + 4 + ...512 + 1 = 1024$. However, the receptive field of the full network won't be a multiple of 512, it will be $3 * (1 + 2 + 4 + ...512) + 1 = 3070$

Gated Activation Units

The gated activations unit used were the same as in the gated PixelCNN [33]:

$$z = \tanh\left(W_{f,k} * x\right) \odot \sigma(W_{g,k} * x) \tag{3.5}$$

where * denotes a convolution, $\odot$ denotes an element wise multiplication, $\sigma(\cdot)$ is the sigmoid function, k is the layer index, f denotes the filter, g denotes the gate, and W is the learneable convolution filter. Gated activation units are in charge of mapping the output of the dilated convolution in resulting values from 0 to 1. In deep learning, is common to use a Relu function instead of other functions. However, in pixelCNN [33] was demonstrated that this particular case of activation units performs better than Relu.

Residual and Skip Connections

A full overview of the entire architecture is presented in figure 3.1. It can be seen that a causal convolution occurs first with the pre-processed data. Later the result of this convolution enters the first layer of the dilated stack. In this block the gated activation units are calculated. Finally, in this block a 1 by 1 convolution is performed. Later, the residual connections and the skip connections must be calculated. To calculate the residual we need to add the output from the first causal convolution and the output of the 1 by 1 convolution. This residual connection will be used to feed the following layer. The skip connection is directly the value that comes from the 1 by 1 convolution. This value is stored, until all the layers are computed. Later all this values are summed and finally the post-processing part is performed.

Post-processing

As could be seen in figure 3.1 the post processing part consist of a ReLU function followed by a 1 by 1 convolution, another ReLu followed by a 1 by 1 convolution, and finally a softmax layer, that is used to calculate the loss.

Conditioning

Finally the neural network could be extended adding a condition both during the training and during the generation part. Adding a condition allows the system to learn different parameters during the training and generate more specifics sounds during the generation. For example, in the case of speech synthesis the system could be conditioned on the speaker identity or also in the specific phoneme to produce. To do that two different types of conditionality are presented: global condition and local condition.

Global condition is used to force the system to learn the speaker identity. This allow us to train the network with different speakers (different gender, race, etc.)

and choose which one to use during the generation. To use global conditioning we need to modify the activation function from equation 3.5 by:

$$z = \tanh\left(W_{f,k} * x + V_{f,k}^{T}\mathrm{h}\right) \odot \sigma(W_{g,k} * x + V_{g,k}^{T}\mathrm{h}) \tag{3.6}$$

where $V_{*,k}$ is a learnable linear projection, and the vector $V_{*,k}^{T}\mathrm{h}$ is broadcast over the time dimension.

Local condition is implemented in a similar way. The use of local condition is what allows the network to produce exactly a specific sentence desired by the user. The main difference between local and global condition is that now the condition have to be updated every sample. When we want to train the system using local conditioning equation 3.5 becomes:

$$z = \tanh\left(W_{f,k} * x + V_{f,k}\mathrm{y}\right) \odot \sigma(W_{g,k} * x + V_{g,k}\mathrm{y}) \tag{3.7}$$

where $V_{f,k} * y$ is now a 1 by 1 convolution and $y = f(h)$ is the transformed information containing the sample labels. In speech synthesis, it contains the linguistic information coming from a TTS model.

3.1.2 Igor Babuschkin Implementation *(ibab)*

When WaveNet was released for the first time in 2016, the authors didn't want to publish any code. Moreover, the paper Oord2016 didn't give all the details to replicate the model. However, due to the relevant results achieved, the online community started a research project led by Igor Babuschkin (ibab) [18]. After several months of discussing the results and the capabilities of this new implementation of waveNet, it started to be considered as the open source "un-official" version. Currently has more than 3000 stars on Github ans has been the starting point for a lot of projects that used a part of WaveNet or even the full implementation.The results, however, were not as realistic as the ones achieved by Oord et.al. That could be for several reasons as the available resources, the time spent to train the network, or other points that have not been implemented because they are not mentioned in the paper. This implementation is written in Tensorflow, an interface for implement machine learning algorithms developed by Google. It runs over Python and it could be executed both in a computer, mobile devices or large-scale distributed systems as GPUs [38]. Although this implementation is relatively easy to execute both for the training and the generating part, the code is very complex for beginners and the published information is not enough to have a good understanding of the project. Due to that, one of the goals of this thesis has also been to create a useful documentation for future researchers and students.

In this section Igor Babuschkin's implementation is presented in detail. This section often refers to concepts seen in section 3.1.1. But is important to understand the relation between the original paper and this implementation.

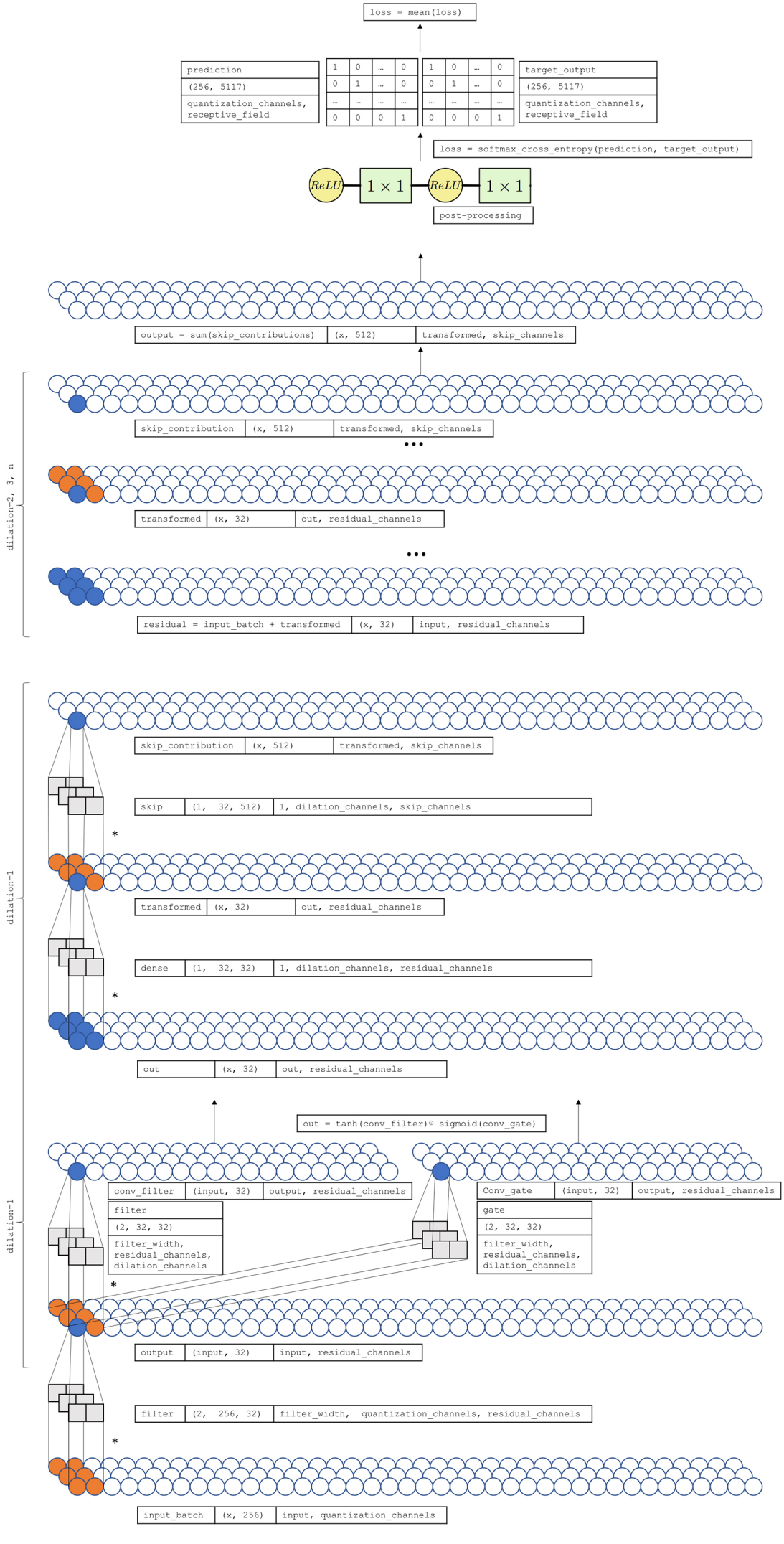
System overview

As explained in section 3.1.1 the input of the network is a directory containing different raw audio files. The system trains the network file by file (by default `batch_size=1`, but could be changed), or splitting it in fragments of 100000 samples (6.25 seconds) is the file is bigger than that. This happens during the preprocessing block, when also a mu-law encoding (see section 3.1.1) is performed, and a tensor is created.Later The tensor created is later one-hot encoded.

The next step is to feed the network that has been previously created. To understand the full architecture of network a clearer representation is presented in figure 3.1.2. First, the one-hot encoded tensor is convolved with a causal filter of length 2 (by default `filter_width=2`) and the output is sent to the dilated_stack block. In this block is where the residual and the skip connections are calculated explained in section 3.1.1 are calculated. Later, as it is explained in section 3.1.1, the post-processing is performed. Finally, the loss is calculated using the softmax cross-entropy between the output of each timestep and the input at the next timestep.

Figure 3.1.2 shows in a more clearer way WaveNet architecture and could ba good help when trying to understand the code. Is important to mention that the number of channels and the length of the tensors doesn't correspond to the reality. On one side or below each element, different information is also displyed. This names corresponds with the variable names used in the code, and are also described in the following section. In the case of the filters, we followed a TensorFlow nomenclature, where each filter is defined with 3 parameters: filter length, filter depth and output channels. The number of ouput channels will condition the size of the next batch, and the number of put channels and depth of the next filter must be the same to perform the convolution.

Explore how this parameters affect to the network have been out of the scope of this project as it was already done by the community. In [23] a long discussion about this could be found. Most of this parameters come with new papers, conferences and experiments. However, we needed a reduced version of WaveNet, and we explored different ways of reducing it.


loss = mean(loss)
prediction
(256, 5117)
quantization_channels, receptive_field
target_output
(256, 5117)
quantization_channels, receptive_field
loss = softmax_cross_entropy(prediction, target_output)
ReLU
1 × 1
ReLU
1 × 1
post-processing
output = sum(skip_contributions)
(x, 512)
transformed, skip_channels
skip_contribution
(x, 512)
transformed, skip_channels
transformed
(x, 32)
out, residual_channels
residual = input_batch + transformed
(x, 32)
input, residual_channels
dilation=2, 3, n
skip_contribution
(x, 512)
transformed, skip_channels
skip
(1, 32, 512)
1, dilation_channels, skip_channels
transformed
(x, 32)
out, residual_channels
dense
(1, 32, 32)
1, dilation_channels, residual_channels
out
(x, 32)
out, residual_channels
dilation=1
out = tanh(conv_filter)⊙ sigmoid(conv_gate)
conv_filter
(input, 32)
output, residual_channels
filter
(2, 32, 32)
filter_width, residual_channels, dilation_channels
Conv_gate
(input, 32)
output, residual_channels
gate
(2, 32, 32)
filter_width, residual_channels, dilation_channels
output
(input, 32)
input, residual_channels
filter
(2, 256, 32)
filter_width, quantization_channels, residual_channels
input_batch
(x, 256)
input, quantization_channels
dilation=1

Parameters

The number of channels could be easily modified given as an input by an external file called `wavenet_params.json`. Part of this thesis has also been to explore with some of these parameters, mainly in order to reduce the computational cost of the network. Some of the most relevant parameters are:

- `filter_width:2`: width of all the filters. Both in the causal layer and in the dilated block.
- `quantization_channels`: number of quantization channels.
- `dilations`: it defines a new layer with the dilation specified.
- `residual_channels`: output channel for the causal filter.
- `skip_channels`: output channels for the skip connection
- `dilation_channels`: output channels for all the filters in the dilation block except for the skip connection.

Network and Variables

The network is defined in `model.py`. Inside this file we can find `WavenetModel` class, a class instanced in `train.py` that is used to create the network. Is in this class where the network is created and where all the weights are calculated and updated during backpropagation. All the variables that are used to create the network are called by the method `_create_variables`. This method basically creates a Python dictionary with the parameters of the filter in each layer. In Tensorflow a filter is defined by 3 parameters: filter width, filter depth, and output channels. The Python dictionary created by `_create_variables` looks like:

- `causal_layer`: first causal convolution. Before entering to the dilation block.

Name	Shape	Parameters
Filter	2, 256, 32	initial_filter_width, quantization_channels, residual_channels

Table 3.2: Parameters of the filter in the causal layer

- `dilated_stack`: All of this variables are created for each dilated layer. This are being append to the dictionary. See table 3.3
- `post-processing`: When all the variables used by the dilated layers are created, the post-processing block defined in 3.1.1 is created. See table 3.1.2.

Name	Shape	Parameters
filter	2, 32, 32	initial_filter_width, residual_channels, dilation_channels
gate	2, 32, 32	filter_width, residual_channels, dilation_channels
dense	1, 32, 32	1, dilation_channels, residual_channels
skip	1, 32, 512	1, dilation_channels, skip_channels
filter_bias	32	dilation_channels
dense_bias	32	residual_channels
skip_bias	512	skip_channels

Table 3.3: Parameters of the filters in the dilated layers

Name	Shape	Parameters
postprocess1	1, 512, 512	1, skip_channels, skip_channels
postprocess2	1, 512, 256	1, skip_channels, quantization_channels
postprocess1_bias	512	skip_channels
postprocess2_bias	256	quantization_channels

Table 3.4: Parameters of the filter for the post-processing block

After computing all the variables needed by the network, the input data is pre-processed as explained in 3.1.1. Function `mu_law_encode` from `wavenet\ops.py` and the method `_one_hot` defined inside this class are the responsible of encoding the data. After that, the network is created in a similar way as defined in 3.1.2. The methods used for creating the network are `_create_network` and `_create_dilation_layer`.

The last step is to calculate the loss function. The loss is calculated as a cross-entropy softmax of all the samples within the receptive field. The target is the input signal, and the prediction are all the predicted samples. During the first steps the predicted signal is smaller than the receptive field, as it is generating one sample at a time. In these cases the predicted samples needed to calculate the loss are feed with 0s.

Global Conditioning

Global conditioning, presented in section 3.1.1, can be understood with figure 3.4. Global conditioning could be understood as adding a bias after the dilation convolution and before calculating the gated activation unit. This bias, consist in a 1x1 convolution by a filter and the global conditioning batch. Global conditioning batch is an array with the same number of channels of the input that contains the information about the category that is being trained. If we look figure 3.4 new parameters are defined. `gc_channels` is the depth of the filters in

this block (`gc_filtweights`) and (`gc_gateweights`). It is also the channels of the `global_conditioning_batch`. It must be 32, because it needs to have the same shape as `conv_filter`. The `cardinality` is automatically calculated by the system and it refers to the number of different categories. It will be explained more in detail in section 3.1.2.

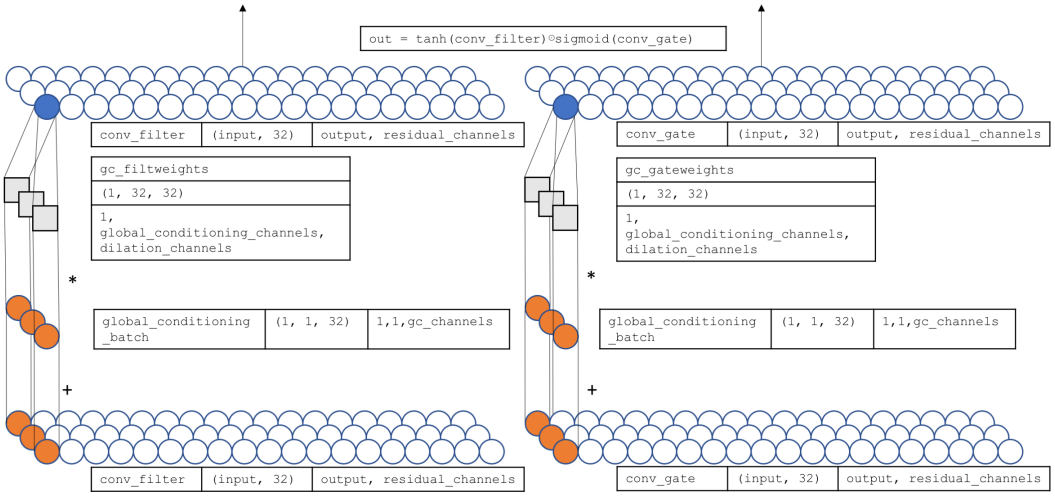


Figure 3.4: Global Conditioning Block

When global condition is detected these new variables are created:

Name	Shape	Parameters
gc_embeddings	x, 32	cardinality, gc_channels
gc_gateweights	1, 32, 32	1, global_condition_channels, dilation_channels
gc_filtweights	1, 32, 32	1, global_condition_channels, dilation_channels

Table 3.5: Parameters of the filter for global conditioning block

So as it was mentioned before, global conditioning is performed by adding a particular bias in each layer and just before calculating the gated activation unit.

Local Conditioning

Local condition basically force the network to generate a specific group of samples belonging to the same class (i.e., vowel, word, pitch...). To achieve this a similar conditioning than the explained before is performed. The main difference, is that this condition won't be the same for all the file. Instead, it has to be automatically extracted by the raw audio. In [1] they used a tool that automatically extracted some linguistic features from the input data. However, what the tool that they exactly use is not mentioned in the paper, and due to that has not been implemented.

To understand how local condition works one could review how global conditioning is implemented. The key difference among them, is that global conditioning is performed file by file. So, `global_conditioning_batch` is fixed during the same file. Local conditioning instead, is changing inside the same file. To achieve that, a new variable called `local_conditioning_batch` needs to contain the information for each sample. As it was mentioned, this information could be something as complex as linguistic features, but could also be something easier like a category used to distinguish two classes in a sound. In that case `local_conditioning_batch` will be 0 every time that the sample belongs to the first category and 1 when it belongs to category. This is what is represented in figure 3.5. Obviously, this one-dimensional vector must be broadcasted to all the channels of the input. Using just one channel to locally condition, however, doesn't give the best results. The intention of showing a diagram with just one channel is merely to make it easier to understand. Part of this thesis has also been to explore different strategies (one-hot, previous, current and future sample, etc.) to improve this quality. This will be described in section 4

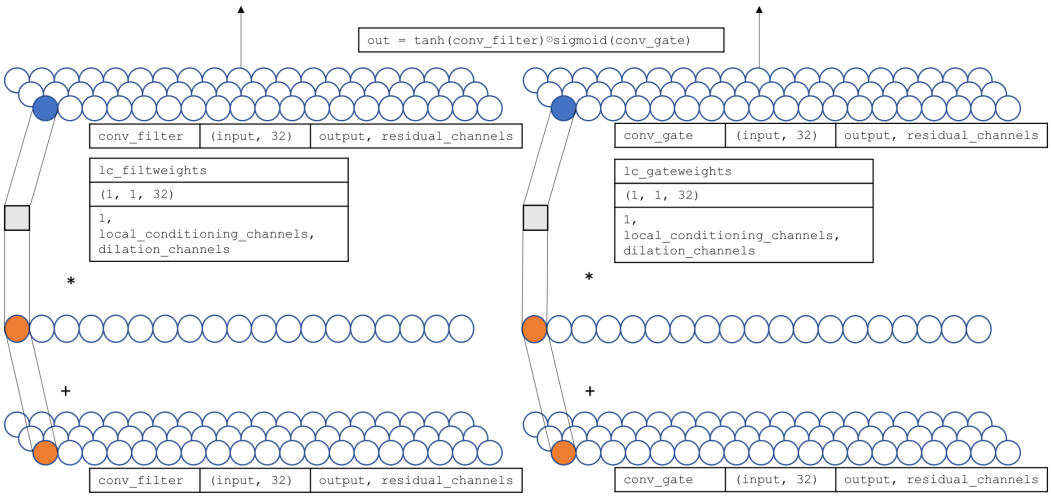


Figure 3.5: Local Conditioning Block

Small Tutorial

In this part a small tutorial on how to use this particular extension ow WaveNet is presented. It follows the official tutorial presented in ibab, which is also a useful resource.

The system is basically formed by 5 important files:

- `wavenet_params.json`: it contains the values for most of the parameters to create the architecture of the network
- `wavenet\audioreader.py`: in charge of the pre-processing part.
- `wavenet\model.py`: it creates the network and performs most of its operations.
- `train.py`: trains the network.
- `generate.py`: generate new audio samples.

The following lines will describe how to train the network, how to generate new audio samples, how to use global condition and finally, how to use local condition.

In order to generate some audio first, the system needs to be trained. That could be done executing the following line in the terminal:

```
python train.py --data_dir=corpus
```

where `--data_dir` refers to the directory where the training data is stored. The data have to be a .wav file. Some parameters can be also added in this execution line. They are well described in the first lines of `train.py` but can be also viewed by executing `python train.py --help`. A recommendation is to add `--silence_threshold=0` and reduce the number of steps `--num_steps` if the dataset is smaller than the VCTK. Edit the architecture of the network is also possible. This is done by editing the parameters in `wavenet_params.json`.

As we commented before is also possible to condition the network. For conditioning it globally we need to add `--gc_channels=32` to the execution line. This tells to the system that global condition is enabled and also defines the number of channels for the condition. If instead of using global conditioning, we want to train the system using local conditioning, the following parameter has to be passed in the previous execution line `--lc_channels=True`.

When the system is trained and the model has been stored, the systemn is ready to generated new audio samples. To achieve that, the following line has to be called:

```
python generate.py --samples 16000 --wav_out_path=generated.wav
logdir/train/2017-02-13T16-45-34/model.ckpt-80000
```

where `--samples` is the number of samples to generate, `--wav_out_path` is the path to save the generated file and `logdir` is the model used for the generation. When running the generation part, both the model and the parameters in `wavenet_param.json` must be the same. As before, it is also possible to condition both locally and globally. To generate a sequence globally conditioned we must add `--gc_cardinality`, `-gc_channels=32` and `--gc_id`. `--gc_cardinality` is the number of different categories and is printed in the console before starting the training and `--gc_id` is the category to generate.

In order to synthesise a new sound locally conditioned more parameters need to be added to the execution line. First, the system needs to know to the values that goes with each sample. We did that by passing .txt file or a .json file, depends of the network used that basically contains the information needed per each sample. That will be described in section 4. However, to add this file, we need to add to the execution line `--labels=labels_path` where labels is the name of the file that contain the local condition. As in the case of global condition, `--lc_channels=...` and `--lc_cardinality=...` have to be added to the execution line. These values are printed during the training and must be the same as it defines the architecture of the network.

3.2 Nsynth

Nsynth [8] was presented in 2017 by Magenta, a research project that explores the role of machine learning in the process of creating art and music. They presented several artistic projects with the purpose of generating new songs, images, drawings and other materials. All the code is always available on his GitHub respository [39].

For this project, and given the good results that Wavenet achieved in audio synthesis, they decided to use Wavenet for a most artistic purpose. In particular, they were inspired by classical synthesizers, in the way that they could create new sounds with specific tone and timbre just modifying some signals as the pitch, the velocity or some filter parameters. Wavenet, although it showed realistic results in speech and music synthesis, was not able to create new sounds. Neither was its purpose. In this project, instead, the purpose was to generate new sounds but, at the same time, preserving some of the characteristics of the training data. In this project they presented two works: first, a Wavenet autoencoder that was able to create new sounds by combining different sounds for the training, and second, a large-scale and high-quality dataset of musical notes.

To understand first how WaveNet works, a brief understanding of autoencoders

is needed. An autoencoder (figure 3.6) is basically a neural network that attempts to copy its input to its output. The network could be seen as consisting of two parts: and encoder function $h = f(x)$ and a decoder who produces the reconstruction $r = g(h)$. Between these two parts there is a hidden layer, the code h, that is used to represent the input in a reduced representation. Traditionally, autoencoders were used mostly for dimensionality reductions, however, nowadays an autoencoder designed to copy perfectly the input is not especially useful. Instead, they only copy specific characteristics of the input data to generate something new. That, have brought autoencoders to the forefront of generative modelling. [40]. This is exactly the case of Nsynth, where an autoencoder has been used to condition the network in order to create new types of expressive and realistic instrument sounds.

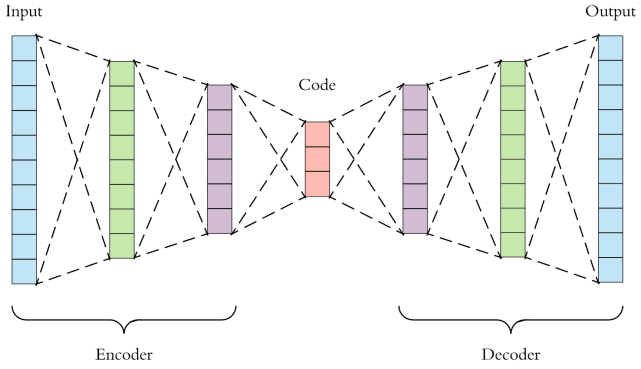


Figure 3.6: Autoencoder

Nsynth works in a similar way as WaveNet. It generates new audio waveforms from raw audio. However, WaveNet relies on external conditions to control the desired output. In Nsynth these external conditions, have been removed and are automatically controlled by a temporal encoder. The temporal encoder (figure **??**) is a 30-layer non linear residual network of dilated convolutions followed by 1x1 convolutions. Each convolution has 128 channels and is followed by a ReLU function and a 1x1 convolution. The temporal encoder output is finally feed into a 1x1 convolution and is downsampled by a factor of 512 (32ms). The time resolution depends on the stride of the pooling layer. A good compromise was found when using 16 dimension per timestep and stride of 512 (32ms when $fs = 16000$). The WaveNet decoder is similar to the original WaveNet model [1], however, every layer is conditioned by the upsampled output of the temporal encoder.

The realistic results achieved by this network is not only thanks to its architecture, but also for the specific dataset used. This dataset was created using commercial sample libraries and it contained 1006 different instruments, 87 dif-

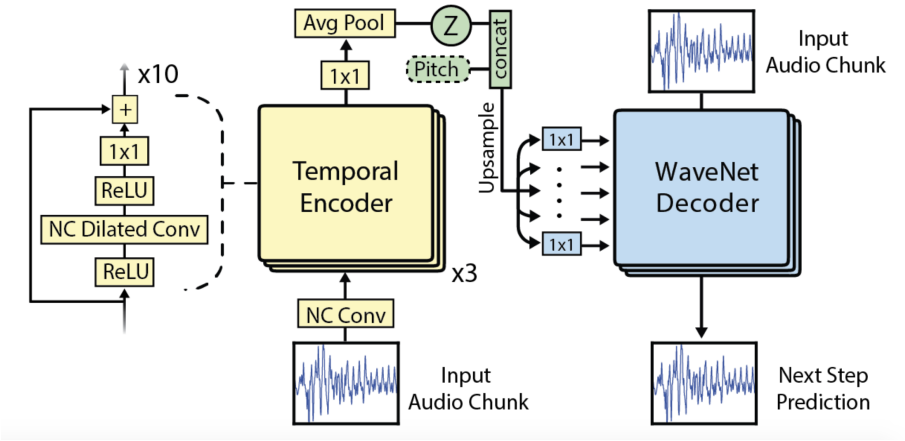


Figure 3.7: Wavenet Autoencoder [8]

ferent MIDI notes and 5 MIDI velocities. The use of this specific dataset allow the encoder to condition the WaveNet decoder, not only with its pitch, but only with its timber. However, producing the same as the input was not the goal of this project. To achieve new realistic sounds the authors modified the embedding in the generation part combining the encoding result from different instruments. As an example, if the conditioned passed in the generation is $(c_1 + c_2)/2$ being c_1 a piano samples and c_2 a flute sample, the result would be a new instrument combining some characteristics of the piano and some of the flute.[41]

Chapter 4

Implementation and Experiments

During this project, different experiments have been done in order to understand the performance of Igor Babuschkin's implementation of WaveNet [18]. All of these experiments are well described in [42], and are also presented in the appendix.

This section starts describing most of these experiments, and analysing it results. Later, two new architectures are presented. Both extend Igor Babuschkin's implementation [18] adding local conditioning. In the first one, local conditioning is used to generate a specific signal from a dataset conditioning it on the mel spectrum. To analyse the dataset, mel spectrum have been used. The other network, also uses local conditioning to generate a specific sound with an specific fundamental frequency but it uses MFCCs as the condition.

4.1 Reducing the Network

The original paper on WaveNet [1] revealed few details about the specifics of the implementation. An online community with interest in the WaveNet explored implementation details. [43].

Most of the challenges that keep the interest of the comunity were how to achieve a realistic speech sound, or a realistic audio sound. However, this thesis differs from most of these posts because we want to understand how WaveNet works, and see what could it learn. For this thesis, the computational cost is a limiting factor, as we only have 3 avilable GPUs to share among all the interested students. The time also was a limiting factor, as we had a short period to present the results. In this first section we explored how reducing the network affected the quality of the generated signal, and how it reduces the computational cost.

We started by reducing the dataset. For that we used one sinuoid sampled at 16000 Hz and duration of 1 second. We have seen that good results were achieved

with a receptive field of 16. 2 stacked layers of 3 dilations increasing by a factor of 2 (2x(1,2,4). Reducing the quantization channels also resulted in a faster computation withouth affecting the generated signal.Training the network during 100 epochs lasted 262 seconds. To generate 1 second of audio (16000 samples) took 38.8 seconds. Figure 4.1 shows the generated signal, and the loss function achieved during the training.

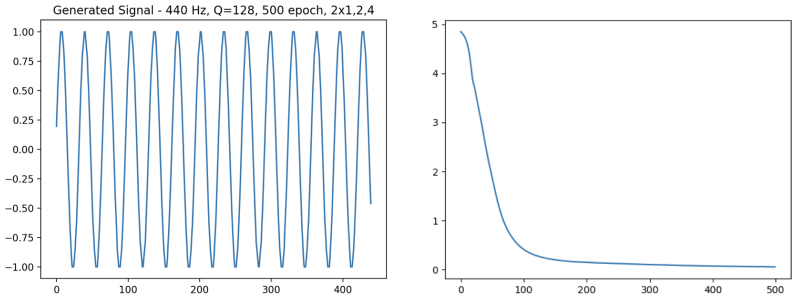


Figure 4.1: Generated signal and Loss function for the reduced network

4.2 Global Conditioning

As we described in section 3.1.1, conditionallity was implemented in order to copy the characteristics of one specified speaker. In Igor Babuschkin's implementation, the condition is specifically designed to work with the VCTK corpus dataset [44]. Each different speaker is treated by the network as a different category, so different filters are used in the conditioning part for each category. The category is taken from the file of the name, and that is performed in the function `get_category_cardinality` from audio_reader.py. In order to use global conditioning with our datasets, this function has to be modified according to the name of our files.

In this section, different datasets have been created to explore global conditioning. Below some of the results are presented. A full description of the experiments could be find in [42].

First, we tried to make the system learn different frequencies. To achieves that, we train the network using 7 different frequencies (440, 493.88, 523.25, 587.33, 659.25, 698.46, 783.99) Hz, and we train the system during 100 epoch. The results, that could be seen in Figure 4.2 showed that the system is able to learn different frequencies.

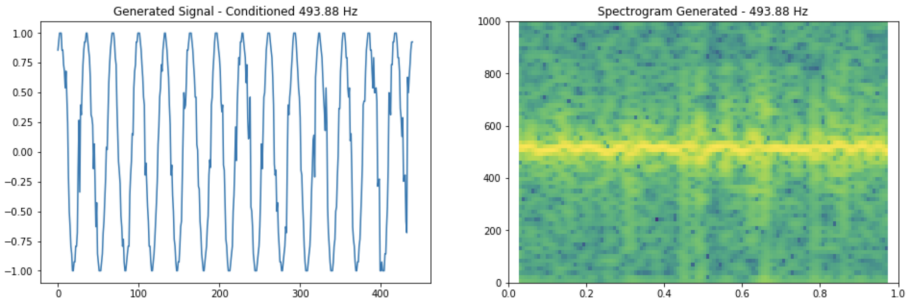


Figure 4.2: Signal generated when conditioned at 493.88 Hz

Second, we investigated if the network was able to learn different amplitudes of the same signal, so we conditioned the network to different amplitudes. We created a dataset containing a sinusoidal signal with frequency $f = 440Hz$ with 4 different amplitudes (0.25, 0.5, 0.75, 1). Using the same network than before and training with 250 epoch, the system was able to learn and generate 4 different signals.

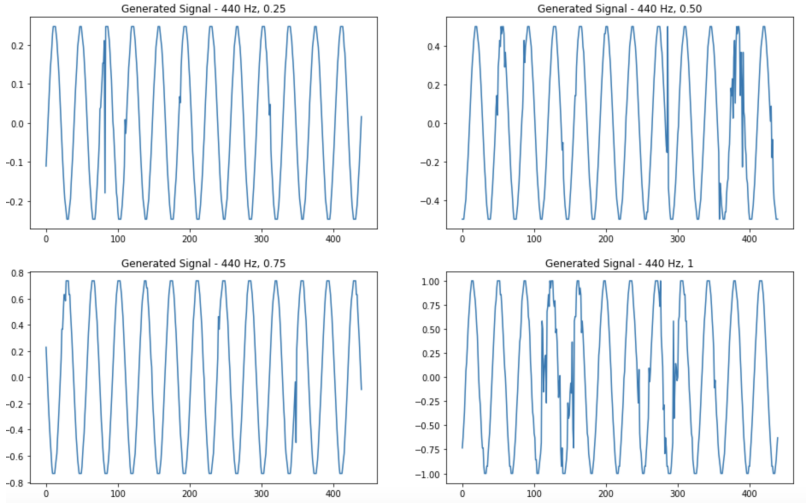


Figure 4.3: Signals generated when trained with 4 different amplitudes

After demonstrating that the network was able to recreate precisely the input, and also was able to distinguish between different sound characteristics, we experimented how the network would behave with a more complex dataset. For that experiment, a dataset containing 900 signals with 100 different frequencies, 3 different shapes (triangle, sinus, sawtooth), and random amplitudes was created. The system was conditioned to the fundamental frequency. The results showed that the network was able to generate a new signal with a new shape, and random

amplitudes, but that was maintaining the frequency. In figure 4.4 we could see the result of a generated signal when the condition was 3928 Hz. This sound could be listened in [42]. Even the signal maintains the frequency, the signal was not peridic anymore. It was periodic for a few samples, and it changes constantly from one shape, or one amplitude to another.

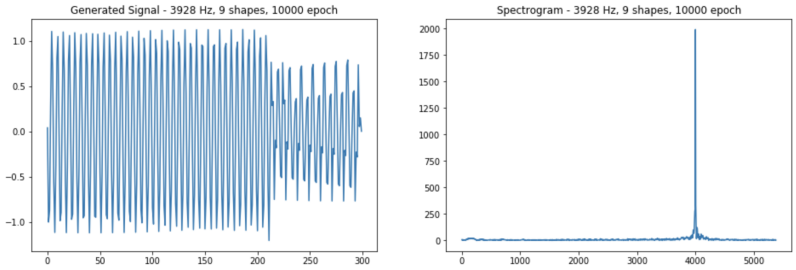


Figure 4.4: Conditioned in frequency with different shape and amplitude

4.3 Local Conditioning

As it is explained in section 3.1.1 local conditionally is the ability of the network to be conditioned inside the same signal. In speech synthesis, for example it is used to generate a specific phoneme. Local condition is not available on Igor Babuschkin's implementation. Moreover, the use of local conditioning for music has not been discussed in the original paper. However, looks that is the key component of new architectures as [8] and [21].

The community involved in the project has also been interested in implementing local conditioning and that motivated us to implement our particular local conditioning. Inspired by [25], [26], [45] and [46] we presented two different models of local Conditioning.

First model first pre-process the input data creating a file where each sample has assigned a category. That category came from analysing all the dataset and extracting the fundamental the frequency. The second network, uses the first 12 MFCCs to condition the network. The results showed that both networks could be used to condition the sequence to generate.

4.3.1 Mel Spectrum

When working on simple group of signals the most distinguishable characteristic among them is their frequency. According to this, we started this part trying to locally condition different signals according to their frequency. The tool presented

is an automatic tool that automatically differentiates different frequencies in the dataset, and create the necessary number of channels to train the network. To use local condition the parameter `--local_condition=True` has to be added on the execution line.

Pre-processing the data

In this step the system looks to all the files in the directory an analyse them using a mel spectrum. We should mention that mel spectrum was selected over other descriptors because we have been working with simple signals, like sinusoids. In that case, mel spectrum could work. However, if this would be extended as a real application this descriptor wouldn't work. When all the different frequencies are taken, they are translated into categories starting from 0 to n, where n is the number of different frequencies. Different experiments showed that passing only this category number as it was done when globally condition the system wouldn't work. Some of the ideas discussed in [25] proposed to use one-hot encoding to define the category. Applying that, the network started to behave as expected and a desired sequence was created. However, a lot of noise was produced during the transitions. To solve that, we added to each sample also the one-hot encoded information about the previous sample and the future sample. This fact removed the noise created during the transitions.

As an example imagine that we have a dataset containing three frequencies (440, 880, 1320 Hz). The conditioning vector will be a 1x9 array where the first three digits represent the previous frequency category one-hot encoded, the following three digits represent the current frequency category one-hot encoded, and the last three digits represent the next sample frequency category one-hot encoded. According to this, the architecture of the network will be totally related with the number of frequencies detected. Similar approaches have been discussed in [25] [26] and [45].

Architecture

An overview of the local conditioning architecture is presented in figure 3.5. Nevertheless, is important to mention that this architecture will be conditioned by the number of different frequencies. As it is commented before, the local conditioning embedding vector (category_id_local in the code) is a one-hot encoded version of the frequency category of the previous, current and next sample. One-hot encoding will depend on the number of categories, and due to that, the embedding vector shape will depend on this. lc_filtweights and lc_gateweights will also have a different shape depending on the number of frequencies detected. The shape

of these two filters will be (1, n, 32) where n is the size of the local conditioning vector.

Generate

During the generation, the network needs to be conditioned per each sample, that means that together we need to pass a new parameter on the execution line. This parameter is called `-labels` and has to contain the path of the labels .txt file. This file has to be at least as long as the number of samples that we want to generate. The format of this file, has to be one of the labels used during the training separated by a coma. Each label has to be created in a new line. Also we have to add the number of categories and the number of channels. This values are printed during before starting the training, and are called --lc_channels and `--lc_cardinality`. An example of an execution line would be:

```
CUDA_VISIBLE_DEVICES=3 python generate.py --samples 24000
--wav_out_path=train0.wav ./logdir/train/2018-05-13T06-14-23/model.ckpt-
2999 --lc_channels=9 --lc_cardinality=3 --labels=./corpus/localTrain/
lc_train0.txt
```

When the networks detects that local conditioning is enabled, it will automatically convert the labels from the .txt file to one-hot vector of previous-current-next sample. We could generalize that the size of each label will be $3xc$ where c is the cardinality, and it refers to the number of different frequencies. Moreover, the size of this vector will also condition the shape of the filters in the local conditioning part (`lc_filtweights` and `lc_gateweights`).

Results

This section presents the most relevant results while working with locally conditioning the network using the mel spectrum. Hypothetically, the system should be able to automatically detect different frequencies on the training data, create the labels and condition the network. After that, we should be able to tell the system which specific sequence of signals we want to generate and it should be able to create this sequence. In order to understand how the network behaves and also considering the available resources, it was decided to work with simple datasets and increase the complexity when the results are favourable. Due to the lack of this kind of datasets, a different set of datasets have been created.

Different datasets are presented in this section, as well as the results and conclusions extracted. A full detailed explanation of the experiments could be seen in the jupyter notebook [42] of the project and also in appendix **??**.

Random Sinusoids with Normalized Amplitude

This dataset consists of 100 signals formed by randomly concatenations of three frequencies (440 Hz, 880 Hz and 1320 Hz). Using a small receptive field of 16 ($2x1, 2, 4$) and training the network 500 epoch the results shows that the network is perfectly able to generate the sequence desired. Figure 4.5 shows the generated signal (right) when the network is locally conditioned with the signal on the left.

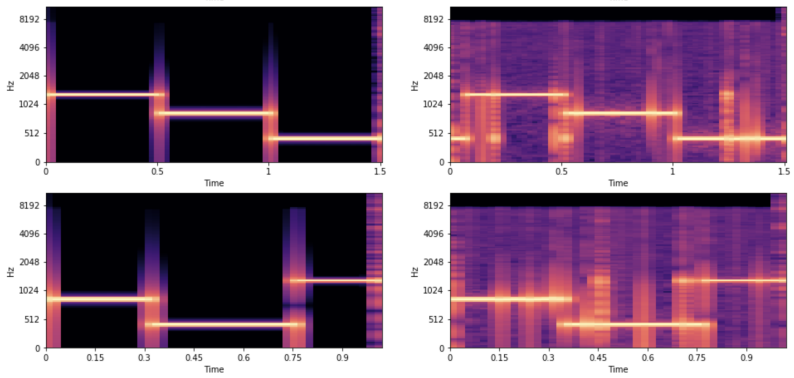


Figure 4.5: Expected signal (left) and generated signal (right)

The training took 35 minutes, and the loss achieved was around 0.6. In order to remove the noise in the generated signal, we tried to train the network by using more iterations. Effectively, the noise was removed from the signal, but it affected the temporal quality of the condition.

Random Sinusoids with Random Amplitudes

A logical step forward to increase the complexity of the dataset, would be to use different frequencies and different amplitudes. In this dataset we created 100 signals formed by the concatenation of the same frequencies as before with random amplitudes. Figure 4.6 shows the generated files together with the labels passed by the input when trained with 500 epoch and 3000 epoch. As it could be observed, increasing the number of iterations remove the noise of the generated signal but then the condition is not learnt. This experiment has been run with different iteration, but we couldn't achieve a good compromise between the temporal resolution of the condition and the absence of noise. We couldn't remove neither the noise applied to the first frequency (440 Hz) which is much superior than in the other two frequencies. That could be due to the length of the period and the short receptive field.

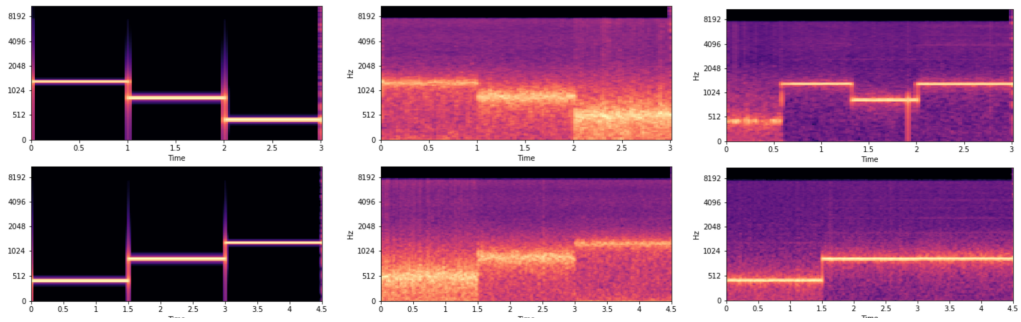


Figure 4.6: Expected signal (left), 500 epoch (middle), 5998 epoch (right)

Random Shapes and Random Amplitudes

This dataset added different signal shapes on the last dataset. It contains 100 signals with three frequencies, 4 random amplitudes and three different shapes (triangular, sinus, sawtooth). To obtain the results showed in figure 4.7 we needed to train the system during 2999 epoch to get this results. It took 3 hours and 20 minutes to train, and the loss value was around 1.21. Less iteration gave worse results in terms of the quality of the sound (noise added) and how the condition is generated.

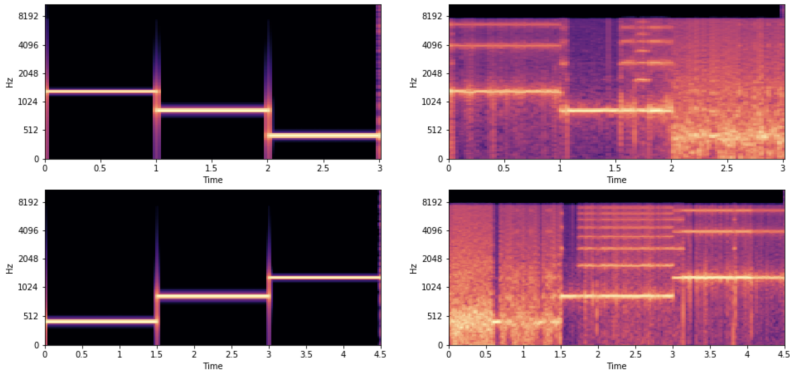


Figure 4.7: Labels when the signal is a sinus (left), generated signals (right)

In terms of the shape, for the second and third frequency is usually constant with a square or a sawtooth. However, for the first frequency is changing constantly from one shape to another, giving a signal with more noise. Figure **??** shows the results for two of this signals. As in the previous dataset the first frequency present more noise than the other. Some harmonics are present due to the square and the saw-tooth shapes.

Panflute Dataset

In this dataset we selected 7 examples from the dataset [47] containing three different frequencies. Then, we randomly concatenated these signals with different lengths creating 100 signals. The dataset was created in order to demonstrate that wavenet is able to work with real instruments. The selection of a panflute over other instruments is because of its timbre and it is shape. As could be appreciated in figure 4.8, it presents only a few harmonics and the waveform is constant over time. In this picture, also we could see how the network is able to recreate the timbre of a panflute and that it could differentiate over different frequencies. Even with most of the frequencies the signal generated is clear, some noise is presented in specific frequencies.

We investigated the reason of why some notes were more noisy than others. First, we looked if it was because the receptive field. But increasing it didn't solve the problem. Later, we explored if it could be because this frequency has less training the data in the dataset, but the dataset was well distributed. So, we still need to explore why some frequencies are more noisy than others.

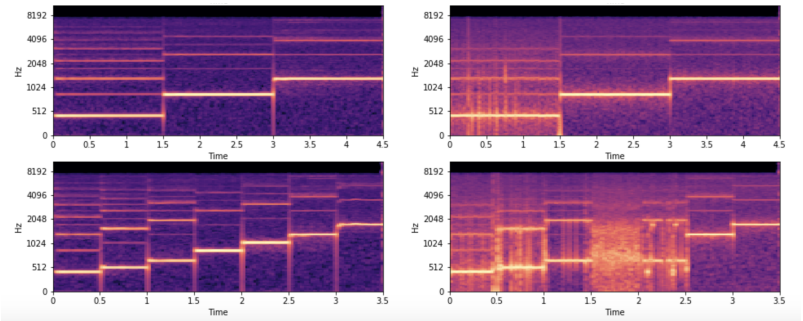


Figure 4.8: Expected signals (left) and generated signals (right)

Piano Dataset

The last dataset presented consists of 100 signals formed with three different frequencies from a piano dataset [47]. The timber of a piano is richer in harmonics than a panflute, and also the waveform is more complex as it presents attack, decay, sustain and release. Due to that, we need to increase the receptive field of the network. An architecture that worked has been using 4 stacked layers of dilations 1,2,4,8, providing a receptive field of 62 samples. Training the system during 3000 epochs took 3 hours and 15 minutes in a NVIDIA TITAN X GPU. Figure 4.9 shows two examples of the generated sounds. As it could be seen the network is able to recreate some of the harmonics of the piano. The local condition is also well learned. As it could be seen also some noise is added to the signal. That

is something common with this implementation. We tried to remove that noise but increasing the number of the iterations. Doing that, however, the noise was reduced but the condition was not learned and the network was not able to create and specific sequence.

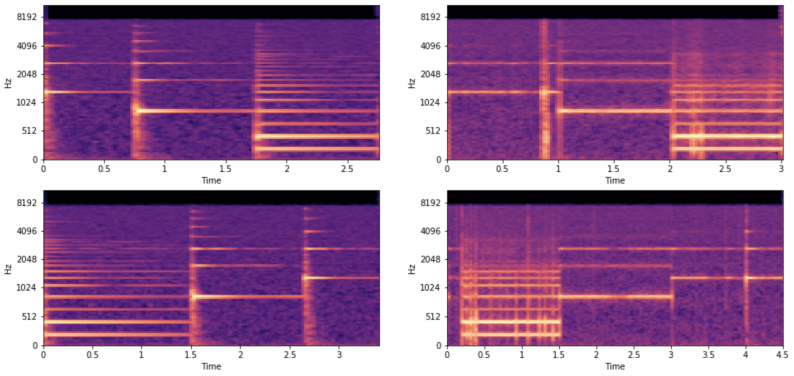


Figure 4.9: Piano labels (left) and generated signals (right)

4.3.2 MFCC

Inspired by Nsynth[8], Tacotron2[16], and the Universal Music Translator Network [21], we decided to use Mel-frequency cepstral coefficients to locally condition the network. In the previous network we used the mel spectrum to classify different frequencies and use the local condition to generate an specific sequence. However this analysis presented different problems. First, mel spectrum is not a good descriptor for detecting the fundamental frequency or the pitch. [48] Other algorithms as pitch detection should be used. Second, using this network with a big dataset probably wouldn't give good results as the number of channels in the local conditioning batch will be too large. Finally, the use of the mel spectrum as the local condition is useful when we want the output to be as similar as possible to the input. However, if we prefer to combine the characteristics of different sounds to create new sounds, as Nsynth [8] is doing, is not a good descriptor.

MFCCs are one of the most common descriptors in audio processing and speech recognition. The steps to compute MFCCs are the following: first, the input signal is windowed in frames of 20-40 ms. Later, for each frame the power spectrum is calculated. After that, this is sent into a bank of K_mel mel frequency filters. Mel filters are a group of triangular filters and unity responds at its centre that treats the signal in a similar way as our cochlea does. The first filter, gives information about how much energy exists around 0 Hertz and is much narrow than last filter, due to that as higher is the frequency, less we perceive the differences in frequency. After computing the mel filter, we take the logarithm of them. Finally, the cosine

discrete transform (CDT) is computed. [49] The selection of MFCCs over other audio descriptors was inspired by Tacotron2 [16] that already used MFCCs.

In order to use MFCCs the architecture of the network has to be modified. In the following lines this structure will be described. First, MFCCs are computed per each signal during the pre-processing.To calculate the MFCCs we used the built in function from the `librosa` library with 20 MFCCs, frames of 2048 samples and a hop size of 512. We selected only 12 MFCCs, from the 2nd to the 13rd as they are the ones that give the most relevant information. This information is upsampled using nearest neighbour and stored in a .json file as it could be converted directly to a python dictionary. When all the files have been analysed, the network will start the training. In that case, `local_condition_batch`, `lc_filtweights` and `lc_gateweights` from figure 3.5 will have always 12 channels. Is important to mention that here MFCCs are directly used to train the network and to generate new samples. That means that during the generation a .json file containing the 12 MFCCs per sample have to be passed to the network.

As before experiments with different datasets have been run. Below some of these experiments are presented and more results are available in the jupyter notebook [42].

Different Sinusoids with Random Amplitudes

Same dataset as presented in 4.3.1, different amplitudes and three frequencies. If we observe the results presented in figure4.10 we could see that the spectrum is as clean as in the last method. However, when conditioning on the fundamental frequency using different amplitudes, we got probelms getting the right sequence. In that case, this is also solved. In this particular example we used 3000 epoch achieving a loss of 0.84.

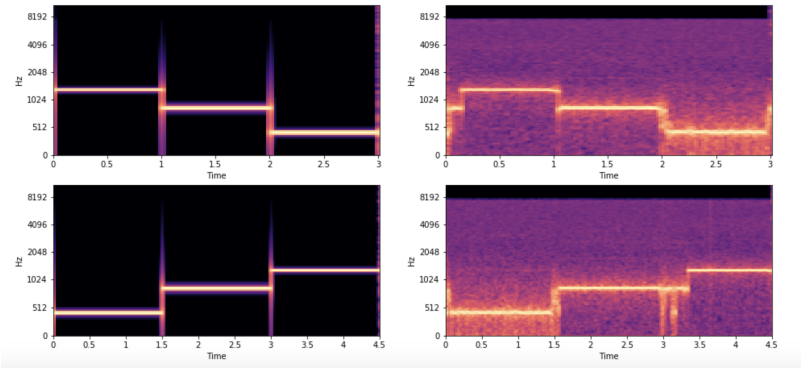


Figure 4.10: Labels (left) and generated signals (right)

Piano Dataset

Seeing the good results achieved with the last dataset, we decided to finally train with real sounds again. We used the pianoDataset, described in section 4.3.1. Results show that the network is able to recreate better the timbre of the selected instrument. Also the condition is learned well, and it changes at the desired sample conditioned by the labels.

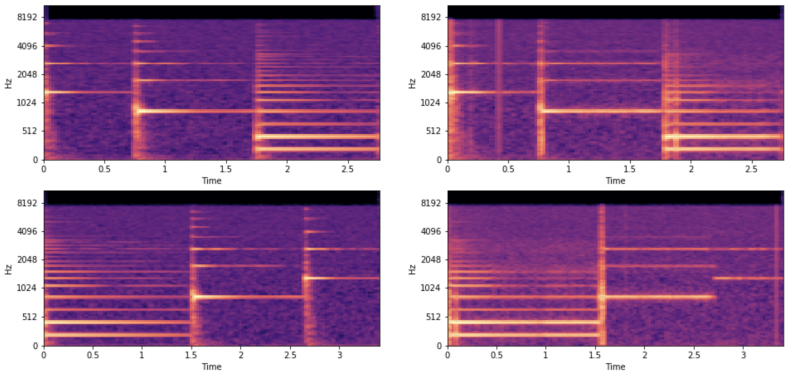


Figure 4.11: Piano labels (left) and generated signals (right)

Chapter 5

Conclusion

In this thesis the state of the art of audio synthesis using artificial Intelligence has been investigated. First, different networks have been presented showing their basics implementations and results. Later, WaveNet [1] has been selected over networks due to its breaking results compared with other methods and its popularity. Although Wavenet [1] is a well-known architecture in the field, it comes with a bit of secrecy by the authors. This lack of information motivated the online community to recreate the network by using the available data and tuning the system by trial and error. This network [18] has been accepted by the community as the starting point to create new projects. However, this implementation is still too complex for novices researchers. And the information available is neither abundant. In this thesis inverse engineering has been applied in order to understand well how the network is working and a more intuitive guide has been provided. Although Igor Babuschkin's implementation has become very popular, the results achieved are far away from Google implementation. That could be due to the limited resources, the time, or some parameters that are not identically tuned.

Being realistic about the time and the resources available in Aalborg University, a reduced architecture of WaveNet has been presented for this thesis. Also different datasets has been created and the behaviour of the network has been explored. First results showed that global conditioning, a part from the purpose of reproducing different speakers, could be used to condition the network to generate a specific frequency, a specific shape or a desired amplitude.

One of the missing parts of Igor Babuschkin's implementations was the local conditioning part. This part was not implemented because when using for speech synthesis a system to extract the linguistic features in a TTS model was not presented in the paper. However, local conditioning could be used in different approaches. In order to understand and see its potential, a simple model that

automatically learns the fundamental frequency of a signal and uses it in local conditioning has been created. After seeing good results both with simple signals and realistic sounds (piano and pan-flute) a more general tool has been also presented. This tool uses 12 MFCCs to condition the network and later it uses it to generate a specific sequence of notes. The use of MFCCs instead of mel spectrum demonstrated cleaner results when the dataset was more complex.

5.1 Future Work

This thesis could be extended in several different ways. First, the system could be trained with larger datasets containing more complex data. Even this implementation is still far from real recordings the use of denoising systems at the end of the network could help on achieving realistic sounds.

The use of MFCCs in the local conditioning part showed that the network reproduced well the pitch and the timber of the input. However, synthesizing a new sound was out of the scope of this work. Secondly, the project could be extended in this way, using MFCCs to create new musical instruments. Finally, two audio descriptors have been explored to locally condition the networ: mel spectrum and MFCC. This thesis could also be extended by finding a more adequate descriptor to use with the local condition.

Bibliography

[1] Aäron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew W. Senior, and Koray Kavukcuoglu. "WaveNet: A Generative Model for Raw Audio". In: *CoRR* abs/1609.03499 (2016). arXiv: 1609.03499. URL: http://arxiv.org/abs/1609.03499.

[2] Warren S. McCulloch and Walter H. Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity". In: *Bulletin of Mathematical Biophysics* Vol. 5 (1943), pp. 115–133.

[3] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

[4] F. Rosenblatt. "The Perceptron: A probabilistic model for information storage and oganization in the brain". In: *Psychological Review* Vol. 65 (1958).

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016, pp. 1–4.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in neural information processing systems* (2012), pp. 1097–1105.

[7] J. Deng, W. Dong, R. Socher, L. J. Li, Kai Li, and Li Fei-Fei. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[8] Jesse Engel, Cinjon Resnick, Adam Roberts, Sander Dieleman, Douglas Eck, Karen Simonyan, and Mohammad Norouzi. "Neural Audio Synthesis of Musical Notes with WaveNet Autoencoders". In: *CoRR* abs/1704.01279 (2017). arXiv: 1704.01279. URL: http://arxiv.org/abs/1704.01279.

[9] A. Roberts, J. Engel, C. Raffel, C. Hawthorne, and D. Eck. "A Hierarchical Latent Vector Model for Learning Long-Term Structure in Music". In: *ArXiv e-prints* (Mar. 2018). arXiv: 1803.05428 [cs.LG].

[10] Adam Roberts, Jesse Engel, Sageev Oore, and Douglas Eck, eds. *Learning Latent Representations of Music to Generate Interactive Musical Palettes*. 2018. URL: http://ceur-ws.org/Vol-2068/milc7.pdf.

[11] Anna Huang, Sherol Chen, Mark Nelson, and Doug Eck. "Mixed-Initiative Generation of Multi-Channel Sequential Structures". In: 2018.

[12] D. Ha and D. Eck. "A Neural Representation of Sketch Drawings". In: *ArXiv e-prints* (Apr. 2017). arXiv: 1704.03477.

[13] Lejaren Hiller and Leonard M. Isaacson. *Experimental music; composition with an electronic computer*. Psychology Press, 2005.

[14] L.-C. Yang, S.-Y. Chou, and Y.-H. Yang. "MidiNet: A Convolutional Generative Adversarial Network for Symbolic-domain Music Generation". In: *ArXiv e-prints* (Mar. 2017). arXiv: 1703.10847 [cs.SD].

[15] A. van den Oord, Y. Li, I. Babuschkin, K. Simonyan, O. Vinyals, K. Kavukcuoglu, G. van den Driessche, E. Lockhart, L. C. Cobo, F. Stimberg, N. Casagrande, D. Grewe, S. Noury, S. Dieleman, E. Elsen, N. Kalchbrenner, H. Zen, A. Graves, H. King, T. Walters, D. Belov, and D. Hassabis. "Parallel WaveNet: Fast High-Fidelity Speech Synthesis". In: *ArXiv e-prints* (Nov. 2017). arXiv: 1711.10433 [cs.LG].

[16] J. Shen, R. Pang, R. J. Weiss, M. Schuster, N. Jaitly, Z. Yang, Z. Chen, Y. Zhang, Y. Wang, R. Skerry-Ryan, R. A. Saurous, Y. Agiomyrgiannakis, and Y. Wu. "Natural TTS Synthesis by Conditioning WaveNet on Mel Spectrogram Predictions". In: *ArXiv e-prints* (Dec. 2017). arXiv: 1712.05884 [cs.CL].

[17] 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". In: *ArXiv e-prints* (Dec. 2016). arXiv: 1612.07837 [cs.SD].

[18] *A TensorFlow implementation of DeepMind's WaveNet paper*. https://github.com/ibab/tensorflow-wavenet. Accessed: 2018-05-30.

[19] D. Rethage, J. Pons, and X. Serra. "A Wavenet for Speech Denoising". In: *ArXiv e-prints* (June 2017). arXiv: 1706.07162 [cs.SD].

[20] A. Borovykh, S. Bohte, and C. W. Oosterlee. "Conditional Time Series Forecasting with Convolutional Neural Networks". In: *ArXiv e-prints* (Mar. 2017). arXiv: 1703.04691 [stat.ML].

[21] N. Mor, L. Wolf, A. Polyak, and Y. Taigman. "A Universal Music Translation Network". In: *ArXiv e-prints* (May 2018). arXiv: 1805.07848 [cs.SD].

[22] S. O. Arik, M. Chrzanowski, A. Coates, G. Diamos, A. Gibiansky, Y. Kang, X. Li, J. Miller, A. Ng, J. Raiman, S. Sengupta, and M. Shoeybi. "Deep Voice: Real-time Neural Text-to-Speech". In: *ArXiv e-prints* (Feb. 2017). arXiv: 1702.07825 [cs.CL].

[23] *Generating good audio samples*. https://github.com/ibab/tensorflow-wavenet/issues/47. Accessed: 2018-05-30.

[24] K. Choi, G. Fazekas, K. Cho, and M. Sandler. "A Tutorial on Deep Learning for Music Information Retrieval". In: *ArXiv e-prints* (Sept. 2017). arXiv: 1709.04396 [cs.CV].

[25] *Global condition and Local conditioning*. https://github.com/ibab/tensorflow-wavenet/issues/112. Accessed: 2018-05-30.

[26] *Trying to condition on F0 (problem with embedding)*. https://github.com/ibab/tensorflow-wavenet/issues/198. Accessed: 2018-05-30.

[27] Francois Pachet. "The continuator: Musical interaction with style". In: *Journal of New Music Research* 32.3 (2003), pp. 333–341.

[28] B. L. Sturm, J. F. Santos, O. Ben-Tal, and I. Korshunova. "Music transcription modelling and composition using deep learning". In: *ArXiv e-prints* (Apr. 2016). arXiv: 1604.08723 [cs.SD].

[29] *ABC Notation*. http://abcnotation.com/. Accessed: 2018-05-30.

[30] *char-rnn*. https://github.com/karpathy/char-rnn. Accessed: 2018-05-30.

[31] *The Unreasonable Effectiveness of Recurrent Neural Networks*. http://karpathy.github.io/2015/05/21/rnn-effectiveness/. Accessed: 2018-05-30.

[32] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: 9 (Dec. 1997), pp. 1735–80.

[33] A. van den Oord, N. Kalchbrenner, O. Vinyals, L. Espeholt, A. Graves, and K. Kavukcuoglu. "Conditional Image Generation with PixelCNN Decoders". In: *ArXiv e-prints* (June 2016). arXiv: 1606.05328 [cs.CV].

[34] A. van den Oord, N. Kalchbrenner, and K. Kavukcuoglu. "Pixel Recurrent Neural Networks". In: *ArXiv e-prints* (Jan. 2016). arXiv: 1601.06759 [cs.CV].

[35] Simon King. "A Beginners Guide to Statistical Parametric Speech Synthesis". In: (Jan. 2010).

[36] *Cloud text-to-speech*. https://cloud.google.com/text-to-speech/. Accessed: 2018-05-30.

[37] *Pulse Code Modulation (PCM) of voice frequencies*. ITU-T. Recommendation G. 711.

[38] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: A system for large-scale machine learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[39] *Magenta: Music and Art Generation with Machine Intelligence*. https://github.com/tensorflow/magenta. Accessed: 2018-05-30.

[40] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016, p. 499.

[41] *NSynth: Neural Audio Synthesis*. https://github.com/tensorflow/magenta/tree/master/magenta/models/nsynth. Accessed: 2018-05-30.

[42] *Wavenet Experiments*. http://nbviewer.jupyter.org/github/aleixcm/tensorflow-wavenet/blob/master/wavenetExperiments.ipynb. Accessed: 2018-05-30.

[43] *A TensorFlow implementation of DeepMind's WaveNet paper - Issues*. https://github.com/ibab/tensorflow-wavenet/issues. Accessed: 2018-05-30.

[44] *English multi-speaker corpus for CSTR voice cloning toolkit*. hhttp://homepages.inf.ed.ac.uk/jyamagis/page3/page58/page58.html. Accessed: 2018-05-30. 2012.

[45] *Local Conditioning on F0 Working (Kind of)*. https://github.com/ibab/tensorflow-wavenet/issues/233. Accessed: 2018-05-30.

[46] *Which features to implement now?* https://github.com/ibab/tensorflow-wavenet/issues/189. Accessed: 2018-05-30.

[47] Masataka Goto, Hiroki Hashiguchi, Takuichi Nishimura, and Ryuichi Oka. "RWC music database: Music genre database and musical instrument sound database". In: (2003).

[48] Alain de Cheveign© and Hideki Kawahara. "YIN, a fundamental frequency estimator for speech and music". In: *The Journal of the Acoustical Society of America* 111.4 (2002), pp. 1917–1930. DOI: 10.1121/1.1458024. URL: http://link.aip.org/link/?JAS/111/1917/1.

[49] Manuel Davy. "An Introduction to Statistical Signal Processing and Spectrum Estimation". In: *Signal Processing Methods for Music Transcription*. Ed. by Anssi Klapuri and Manuel Davy. Boston, MA: Springer US, 2006, pp. 21–64. ISBN: 978-0-387-32845-4. DOI: 10.1007/0-387-32845-9_2. URL: https://doi.org/10.1007/0-387-32845-9_2.

Appendix A

Jupyter Notebook Experiments

This appendix contains all the experiments performed to achieve the results previously presented in section 4 . It aims to be a useful guide to understand some of the decisions taken during this project. This Jupyter Notebook is also available in [42]

To perform these these experiments the use of specific GPUs has been required. Thanks to Aalborg University and the Machine Learning Lab we could make use of three Nvidia Titan X.

Series of experiments performed with ibab implementation of wavenet

Input data

Import and plot input data

In [1]:

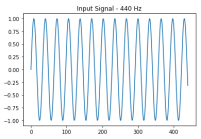
```
# Put these at the top of every notebook, to get automatic reloading and inline plotting
%reload_ext autoreload
%autoreload 2
%matplotlib inline
```

In [2]:

```
import matplotlib.pyplot as plt
import scipy.io.wavfile as wavfile
import librosa, librosa.display
import IPython.display as ipd
import numpy as np
import scipy as sc
```

In [3]:

```
signal = wavfile.read('corpus/oneSin/sinus16000.wav')
signal = signal[1]
plt.figure(1)
plt.title('Input Signal - 440 Hz')
reduced = signal[:440]
plt.plot(reduced)
plt.show()
```



In this first experiment we used the recommendations from the paper. That means 5 stacked layers with 10 dilation layers per stack. That give a receptive field of **5116**.

```
receptive_field = (filter_width - 1) * sum(dilations) + 1
```

where

```
filter_width = 2

dilations = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512,
             1, 2, 4, 8, 16, 32, 64, 128, 256, 512,
             1, 2, 4, 8, 16, 32, 64, 128, 256, 512,
             1, 2, 4, 8, 16, 32, 64, 128, 256, 512,
             1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

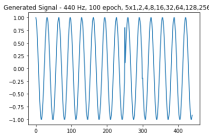
100 epoch were used.

Reducing wavenet

First experiments consisted of sinusoidal signal input with sample rate 16000 Hz and frequency 440 Hz. Some of the parameters have been modified in order to reduce the complexity and the computational cost of the network.

In [4]:

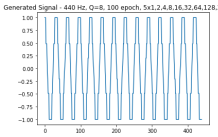
```
signal = wavfile.read('generatedSignals/longSin.wav')
signal = signal[1]
plt.title('Generated Signal - 440 Hz, 100 epoch, 5x1,2,4,8,16,32,64,128,256,512')
reduced = signal[-440:]
plt.plot(reduced)
plt.show()
```



Reducing the quantization give the following results:

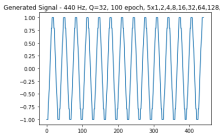
In [5]:

```
signal = wavfile.read('generatedSignals/quantization8.wav')
signal = signal[1]
plt.title('Generated Signal - 440 Hz, Q=8, 100 epoch, 5x1,2,4,8,16,32,64,128,256,512')
reduced = signal[-440:]
plt.plot(reduced)
plt.show()
```



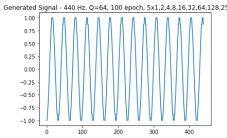
In [6]:

```
signal = wavfile.read('generatedSignals/quantization32.wav')
signal = signal[1]
plt.title('Generated Signal - 440 Hz, Q=32, 100 epoch, 5x1,2,4,8,16,32,64,128,256,512')
reduced = signal[-440:]
plt.plot(reduced)
plt.show()
```



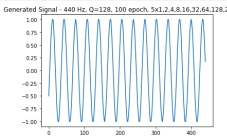
In [7]:

```
signal = wavfile.read('generatedSignals/quantization64.wav')
signal = signal[1]
plt.title('Generated Signal - 440 Hz, Q=64, 100 epoch, 5x1,2,4,8,16,32,64,128,256,512')
reduced = signal[-440:]
plt.plot(reduced)
plt.show()
```



In [8]:

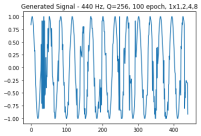
```
signal = wavfile.read('generatedSignals/quantization128.wav')
signal = signal[1]
plt.title('Generated Signal - 440 Hz, Q=128, 100 epoch, 5x1,2,4,8,16,32,64,128,256,512')
reduced = signal[-440:]
plt.plot(reduced)
plt.show()
```



Changing the quantization to **128** good results are achieved. Now, the number of layers will be decreased.

In [9]:

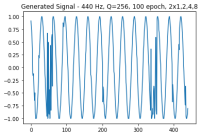
```
signal = wavfile.read('generatedSignals/dilations1x1248.wav')
signal = signal[1]
plt.title('Generated Signal - 440 Hz, Q=256, 100 epoch, 1x1,2,4,8')
reduced = signal[-440:]
plt.plot(reduced)
plt.show()
```



In [10]:

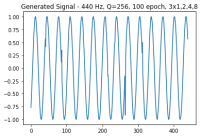
```
signal = wavfile.read('generatedSignals/dilations2x1248.wav')
signal = signal[1]
plt.title('Generated Signal - 440 Hz, Q=256, 100 epoch, 2x1,2,4,8')
```

```
reduced = signal[-440:]
plt.plot(reduced)
plt.show()
```



In [11]:

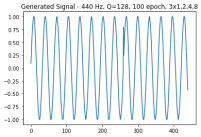
```
signal = wavfile.read('generatedSignals/dilations3x1248.wav')
signal = signal[1]
plt.title('Generated Signal - 440 Hz, Q=256, 100 epoch, 3x1,2,4,8')
reduced = signal[-440:]
plt.plot(reduced)
plt.show()
```



Finally a combination of the best results obtained with reducing the number stacked layers and the quantization steps is presented. Q = 128 and dilations = 3x[1,2,4,8]. The receptive field is equal to 47. This configuration will be used during the following experiments.

In [12]:

```
signal = wavfile.read('generatedSignals/dilations3x128q128.wav')
signal = signal[1]
plt.title('Generated Signal - 440 Hz, Q=128, 100 epoch, 3x1,2,4,8')
reduced = signal[-440:]
plt.plot(reduced)
plt.show()
```



Global Conditionality

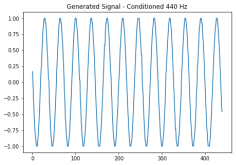
In the original paper conditionality is used to control the speaker or the speech. Let's see how it behaves with different sinusoidal signals.

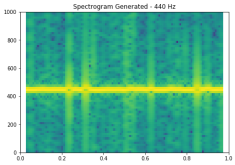
Two frequencies

First we'll use two signals named sinus1.wav and sinus2.wav with frequencies 440Hz and 493.88

In [13]:

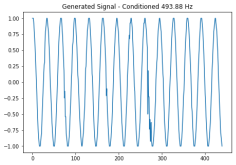
```
signal = wavfile.read('generatedSignals/consin1.wav')
signal = signal[1]
reduced = signal[-440:]

fig, (ax1, ax2) = plt.subplots(1,2,figsize=(16,5))
ax1.plot(reduced); ax1.set_title('Generated Signal - Conditioned 440 Hz')
NFFT = 1024; Fs = 16000;
ax2.specgram(signal, NFFT=NFFT, Fs=Fs, noverlap=900);
ax2.set_title('Spectrogram Generated - 440 Hz')
plt.axis([0,1,0,1000]);
```

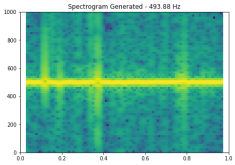




In [14]:

```
signal = wavfile.read('generatedSignals/consin2.wav')
signal = signal[1]
reduced = signal[-440:]

fig, (ax1, ax2) = plt.subplots(1,2,figsize=(16,5))
ax1.plot(reduced); ax1.set_title('Generated Signal - Conditioned 493.88 Hz')
NFFT = 1024; Fs = 16000;
ax2.specgram(signal, NFFT=NFFT, Fs=Fs, noverlap=900);
ax2.set_title('Spectrogram Generated - 493.88 Hz')
plt.axis([0,1,0,1000]);
```





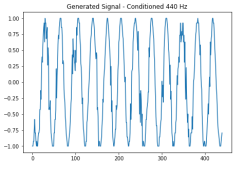
One Scale

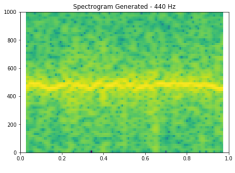
Now, the system was conditioned to learn one scale. That means that we conditioned that with 7 categories. The frequencies used were:

```
f = [440, 493.88, 523.25, 587.33, 659.25, 698.46, 783.99] #array of frequencies
```

In [15]:

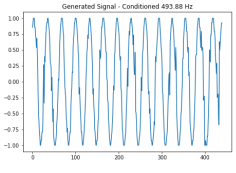
```
signal = wavfile.read('generatedSignals/scaleSin0.wav')
signal = signal[1]
reduced = signal[-440:]

fig, (ax1, ax2) = plt.subplots(1,2,figsize=(16,5))
ax1.plot(reduced); ax1.set_title('Generated Signal - Conditioned 440 Hz')
NFFT = 1024; Fs = 16000;
ax2.specgram(signal, NFFT=NFFT, Fs=Fs, noverlap=900);
ax2.set_title('Spectrogram Generated - 440 Hz')
plt.axis([0,1,0,1000]);
```

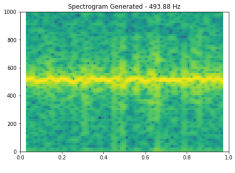




In [16]:

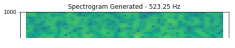
```
signal = wavfile.read('generatedSignals/scaleSin1.wav')
signal = signal[1]
reduced = signal[-440:]

fig, (ax1, ax2) = plt.subplots(1,2,figsize=(16,5))
ax1.plot(reduced); ax1.set_title('Generated Signal - Conditioned 493.88 Hz')
NFFT = 1024; Fs = 16000;
ax2.specgram(signal, NFFT=NFFT, Fs=Fs, noverlap=900);
ax2.set_title('Spectrogram Generated - 493.88 Hz')
plt.axis([0,1,0,1000]);
```

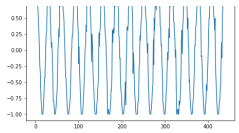
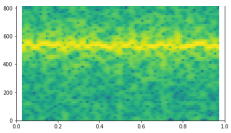




In [17]:

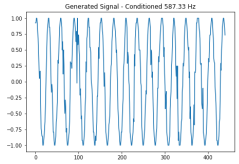
```
signal = wavfile.read('generatedSignals/scaleSin2.wav')
signal = signal[1]
reduced = signal[-440:]

fig, (ax1, ax2) = plt.subplots(1,2,figsize=(16,5))
ax1.plot(reduced); ax1.set_title('Generated Signal - Conditioned 523.25 Hz')
NFFT = 1024; Fs = 16000;
ax2.specgram(signal, NFFT=NFFT, Fs=Fs, noverlap=900);
ax2.set_title('Spectrogram Generated - 523.25 Hz')
plt.axis([0,1,0,1000]);
```

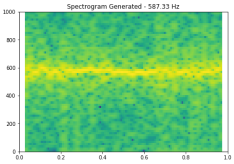


In [18]:

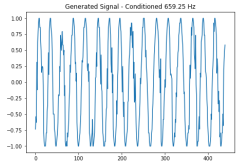
```
signal = wavfile.read('generatedSignals/scaleSin3.wav')
signal = signal[1]
reduced = signal[-440:]

fig, (ax1, ax2) = plt.subplots(1,2,figsize=(16,5))
ax1.plot(reduced); ax1.set_title('Generated Signal - Conditioned 587.33 Hz')
NFFT = 1024; Fs = 16000;
ax2.specgram(signal, NFFT=NFFT, Fs=Fs, noverlap=900);
ax2.set_title('Spectrogram Generated - 587.33 Hz')
plt.axis([0,1,0,1000]);
```

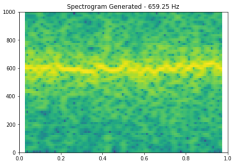




In [19]:

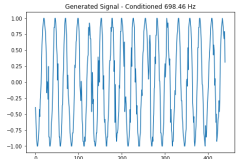
```
signal = wavfile.read('generatedSignals/scaleSin4.wav')
signal = signal[1]
reduced = signal[-440:]

fig, (ax1, ax2) = plt.subplots(1,2,figsize=(16,5))
ax1.plot(reduced); ax1.set_title('Generated Signal - Conditioned 659.25 Hz')
NFFT = 1024; Fs = 16000;
ax2.specgram(signal, NFFT=NFFT, Fs=Fs, noverlap=900);
ax2.set_title('Spectrogram Generated - 659.25 Hz')
plt.axis([0,1,0,1000]);
```

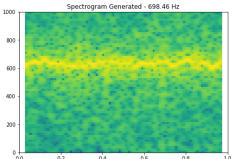




In [20]:

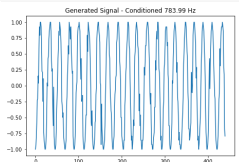
```
signal = wavfile.read('generatedSignals/scaleSin5.wav')
signal = signal[1]
reduced = signal[-440:]

fig, (ax1, ax2) = plt.subplots(1,2,figsize=(16,5))
ax1.plot(reduced); ax1.set_title('Generated Signal - Conditioned 698.46 Hz')
NFFT = 1024; Fs = 16000;
ax2.specgram(signal, NFFT=NFFT, Fs=Fs, noverlap=900);
ax2.set_title('Spectrogram Generated - 698.46 Hz')
plt.axis([0,1,0,1000]);
```

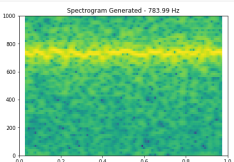




In [21]:

```
signal = wavfile.read('generatedSignals/scaleSin6.wav')
signal = signal[1]
reduced = signal[-440:]

fig, (ax1, ax2) = plt.subplots(1,2,figsize=(16,5))
ax1.plot(reduced); ax1.set_title('Generated Signal - Conditioned 783.99 Hz')
NFFT = 1024; Fs = 16000;
ax2.specgram(signal, NFFT=NFFT, Fs=Fs, noverlap=900);
ax2.set_title('Spectrogram Generated - 783.99 Hz')
plt.axis([0,1,0,1000]);
```





As we can see the system is able to learn all the frequencies. However, more noise is added than before. This could be solved increasing the number of epochs, or increasing the complexity of the network.

Changing shape, frequency and amplitude

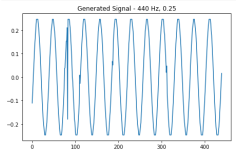
Following experiments analyze the behaviour of the network with a training dataset consisted in different shapes, frequencies and amplitudes.

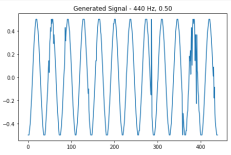
Different Amplitudes

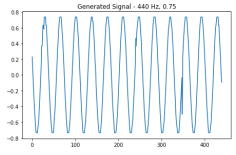
Here we conditioned wavenet with four different amplitudes (0.25, 0.5, 0.75,1). Each amplitude is a category. 100 epoch was not enough, but increasing the number of epoch clearly increases the generated file. I used 250 epoch. From here we can see that wavenet is able to learn the amplitudes.

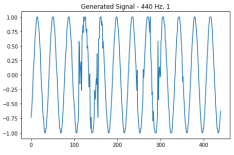
In [22]:

```
signal0 = wavfile.read('generatedSignals/amp0.wav')
signal0 = signal0[1]
reduced0 = signal0[-440:]
signal1 = wavfile.read('generatedSignals/amp1.wav')
signal1 = signal1[1]
reduced1 = signal1[-440:]
signal2 = wavfile.read('generatedSignals/amp2.wav')
signal2 = signal2[1]
reduced2 = signal2[-440:]
signal3 = wavfile.read('generatedSignals/amp3.wav')
signal3 = signal3[1]
reduced3 = signal3[-440:]

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(nrows=2, ncols=2,figsize=(16,10) )
ax1.plot(reduced0); ax1.set_title('Generated Signal - 440 Hz, 0.25')
ax2.plot(reduced1); ax2.set_title('Generated Signal - 440 Hz, 0.50')
ax3.plot(reduced2); ax3.set_title('Generated Signal - 440 Hz, 0.75')
ax4.plot(reduced3); ax4.set_title('Generated Signal - 440 Hz, 1');
```









Different Shapes

Here we trained wavenet with 2 different frequencies 440Hz and 880 Hz. With three different shapes of waveforms (sinusoidal, square, and sawtooth). The hypothesis is that wavenet will learn f0. 1000 epoch have been used as 100 and 250 gives poor results.

In [23]:

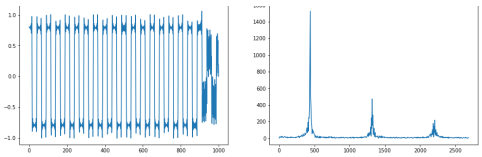
```
x, sr = librosa.load('generatedSignals/shape440_1000.wav')
ipd.Audio(x, rate=sr)
```

Out[23]:

Your browser does not support the audio element.

In [24]:

```
X = sc.fft(x[:4096])
X_mag = np.absolute(X)        # spectral magnitude
f = np.linspace(0, sr, 4096)  # frequency variable
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2,figsize=(16,5) )
ax1.plot(x[-1000:]); ax1.set_title('Generated Signal - 440 Hz, 3 shapes, 1000 epoch')
ax2.plot(f[:500],X_mag[:500]); ax2.set_title('Spectrogram - 440 Hz, 3 shapes, 1000 epoch');
```



In [25]:

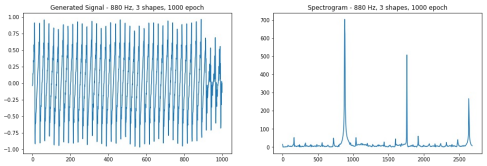
```
x, sr = librosa.load('generatedSignals/shape880_1000.wav')
ipd.Audio(x, rate=sr)
```

Out[25]:

Your browser does not support the audio element.

In [26]:

```
X = sc.fft(x[:4096])
X_mag = np.absolute(X)        # spectral magnitude
f = np.linspace(0, sr, 4096)  # frequency variable
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2,figsize=(16,5) )
ax1.plot(x[-1000:]); ax1.set_title('Generated Signal - 880 Hz, 3 shapes, 1000 epoch')
ax2.plot(f[:500],X_mag[:500]); ax2.set_title('Spectrogram - 880 Hz, 3 shapes, 1000 epoch');
```



Different shapes and Amplitudes (Big Dataset)

From previous experiments we have seen that using global conditionality on wavenet makes it possible to learn different frequencies, different shapes, and different amplitudes. Moreover, when wavenet is trained with different waveforms with the same frequency (f0) it is able to learn this frequency. In this experiment wavenet is trained with a dataset of 900 signals, containing 100 different frequencies with different amplitudes and shapes. Each frequency will be a category for the global conditionality. The hypothesis is that wavenet will learn the f0.

This first example is generated with 1000 epoch, and a frequency of 1351 Hz.

In [27]:

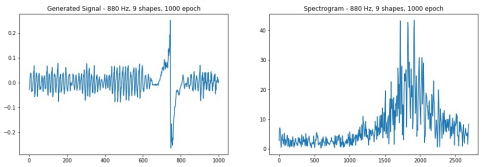
```
x, sr = librosa.load('generatedSignals/random0.wav')
ipd.Audio(x, rate=sr)
```

Out[27]:

Your browser does not support the audio element.

In [28]:

```
X = sc.fft(x[:4096])
X_mag = np.absolute(X)        # spectral magnitude
f = np.linspace(0, sr, 4096)  # frequency variable
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2,figsize=(16,5) )
ax1.plot(x[-1000:]); ax1.set_title('Generated Signal - 880 Hz, 9 shapes, 1000 epoch')
ax2.plot(f[:500],X_mag[:500]); ax2.set_title('Spectrogram - 880 Hz, 9 shapes, 1000 epoch');
```



Poor results are achieved. The loss value was 1.4533637 for the last iteration. However that value, could be improved increasing the number of iterations. If that is not possible, then the network should be increased.

Increasing the number of epoch to 10000 the results achieved are much better. In this example the f0 was 3928 Hz, and in the same category there were 6 different files with the same f0 and different shapes and amplitudes.

In [29]:

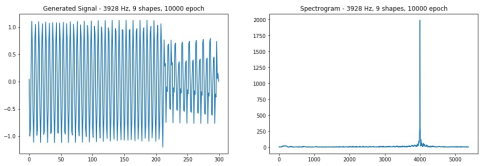
```
x, sr = librosa.load('generatedSignals/random9999_6.wav')
ipd.Audio(x, rate=sr)
```

Out[29]:

Your browser does not support the audio element.

In [30]:

```
X = sc.fft(x[:4096])
X_mag = np.absolute(X)        # spectral magnitude
f = np.linspace(0, sr, 4096)  # frequency variable
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2,figsize=(16,5) )
ax1.plot(x[-300:]); ax1.set_title('Generated Signal - 3928 Hz, 9 shapes, 10000 epoch')
ax2.plot(f[:1000],X_mag[:1000]); ax2.set_title('Spectrogram - 3928 Hz, 9 shapes, 10000 epoch');
```



F0 = 2659, 9 different shapes, 10000 epoch

In [31]:

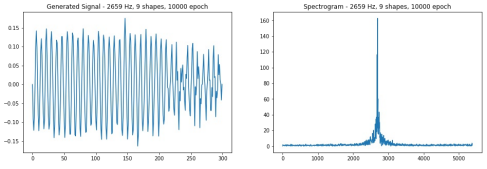
```
x, sr = librosa.load('generatedSignals/random9999_54.wav')
ipd.Audio(x, rate=sr)
```

Out[31]:

Your browser does not support the audio element.

In [32]:

```
X = sc.fft(x[:4096])
X_mag = np.absolute(X)        # spectral magnitude
f = np.linspace(0, sr, 4096)  # frequency variable
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2,figsize=(16,5) )
ax1.plot(x[-300:]); ax1.set_title('Generated Signal - 2659 Hz, 9 shapes, 10000 epoch')
ax2.plot(f[:1000],X_mag[:1000]); ax2.set_title('Spectrogram - 2659 Hz, 9 shapes, 10000 epoch');
```



F0 = 4136, 9 different shapes, 10000 epoch

In [33]:

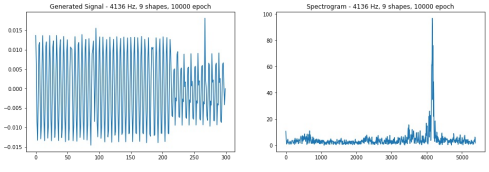
```
x, sr = librosa.load('generatedSignals/random9999_75.wav')
ipd.Audio(x, rate=sr)
```

Out[33]:

Your browser does not support the audio element.

In [34]:

```
X = sc.fft(x[:4096])
X_mag = np.absolute(X)        # spectral magnitude
f = np.linspace(0, sr, 4096)  # frequency variable
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2,figsize=(16,5) )
ax1.plot(x[-300:]); ax1.set_title('Generated Signal - 4136 Hz, 9 shapes, 10000 epoch')
ax2.plot(f[:1000],X_mag[:1000]); ax2.set_title('Spectrogram - 4136 Hz, 9 shapes, 10000 epoch');
```



F0 = 2413, 9 different shapes, 10000 epoch

In [35]:

```
x, sr = librosa.load('generatedSignals/random9999_99.wav')
ipd.Audio(x, rate=sr)
```

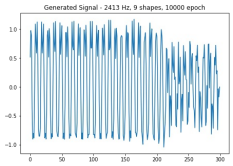
Out[35]:

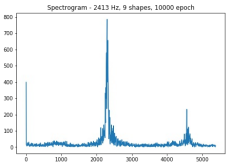
Your browser does not support the audio element.

In [36]:

```
X = sc.fft(x[:4096])
X_mag = np.absolute(X)        # spectral magnitude
f = np.linspace(0, sr, 4096)  # frequency variable
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2,figsize=(16,5) )
```

```
ax2.plot(f[:1000],X_mag[:1000]); ax2.set_title('Spectrogram - 2413 Hz, 9 shapes, 10000 epoch');
```





Conclusions

- Wavenet is able to learn the shape of the signal.
- Wavenet is able to learn the amplitude of the signal.
- When globally conditioned wavenet could learn different frequencies.
- When globally conditioned with different waveforms having the same frequency, wavenet is able to learn that frequency.

Local Conditioning

Local Conditioning on the generation part

Local conditioning was used in the original paper to generate the specific words that wavenet had to reproduce. That was done adding a second time series vector or each sample. In the speech case, that vector contained linguistic features information in a TTS model.

In ibab's implementation, local conditioning is still not implemented. The main difficulty is that the network is training the model file by file instead of sample by sample. However, similar results could be achieved training the network with Global Condition and generating with Local Condition.

For this experiment, local conditioning in the generation part has been implemented. In the following example, wavenet has been trained with three different sinusoids corresponding to the a minor chord. Each sinusoid correspond to one category. Then, in the generation part, a .txt file containing the information for each sample is passed to the system. That allows us to create different sequences.

Three results are presented below.

The code is available on the github repository under the branch **localCondition_noGlobal.**

```
--samples 24000  --wav_out_path=generatedSignals/aminor3.wav --gc_channels=32 --gc_cardinality=3 --labels=corpus/Aminor/aminor3.txt logdir/train/2018-04-13T17-51-32/model.ckpt-99
```

In [37]:

```
x, sr = librosa.load('generatedSignals/aminor.wav')
ipd.Audio(x, rate=sr)
```

Out[37]:

Your browser does not support the audio element.

In [38]:

```
x, sr = librosa.load('generatedSignals/aminor2.wav')
ipd.Audio(x, rate=sr)
```

Out[38]:

Your browser does not support the audio element.

In [39]:

```
x, sr = librosa.load('generatedSignals/aminor3.wav')
ipd.Audio(x, rate=sr)
```

Out[39]:

Your browser does not support the audio element.

Local Conditioning both on Train and Generate

If local condition is implemented in the training part it should be able to detect some information about the sample that is feeding. Our idea is too feed a signal containing three different frequencies (440Hz, 880Hz, and 1320Hz) and then choose which one to reproduce in the generation part. For that porpouse a .txt file with all the information for each sample has to be passed through the network. In our example this .txt file will be formed by 8000 x [0, 1, 2] corresponding to the three categories.

The implementation could be seen under the branch localCondition

First results with small receptive field of 47 and few iterations 300 are not good enough.

In [40]:

```
x, sr = librosa.load('generatedSignals/lc_train_0.wav')
ipd.Audio(x, rate=sr)
```

Out[40]:

Your browser does not support the audio element.

Increasing the number of iterations to 10000 we achieve a similar result.

In [41]:

```
x, sr = librosa.load('generatedSignals/lc_train_9999.wav')
ipd.Audio(x, rate=sr)
```

Out[41]:

Your browser does not support the audio element.

With a big receptive field of 5117 and 10000 epoch the results are.....

In [42]:

```
x, sr = librosa.load('generatedSignals/lc_train_9999_5117.wav')
ipd.Audio(x, rate=sr)
```

Out[42]:

Your browser does not support the audio element.

Final Version

Severals trials and configurations have been tested in order to achieve good results. Some things learned during this process are:

- One Hot Encoding gives better results during the conditioning. However noise is present during the transitions.
- Using a dataset with more than one signal varying the sequences and adding noise helps on reducing the noise between different frequencies.
- When a big dataset is used the freqüencies are not changed as good as with one signal.
- We can solve this adding the left sample and the right sample in the condition. So, in the case that we would have three frequencies (categories) the local_condition_batch that will come with each sample would look like [0 0 0, 0 0 0, 0 0 0] where the first three 0's represent the frequency of the left sample one hot encoded, the next three represent the current sample and the last ones represents the next sample.
- If we use a simple dataset (sinusoids between 1 - 3s) a smaller receptive field works better. If the receptive field is bigger, the noise is reduced but the local condition is not created. We use 16 for simple signals and 62 for more complex signals (piano).

Below I present some results achieved with an automatic tool that automatically extract the different frequencies using a mel-spectograms. Different datasets have been tried. This code, that could be find in the branch localCondition_embed, works with any given input. Is not limited to any number of different frequencies. However, we trained and generated with 3 and 7 frequencies.

Datasets

- **localTrainBigDataset_noAmp:** 100 signals combining three different frequencies (440 Hz, 880Hz, 1320Hz) with the same amplitude.
- **localTrainBigDataset2:** 100 signals combining three different frequencies (440Hz, 880Hz, 1320Hz) with random combinations and random amplitudes.
- **localTrainBigDatasetShapeAmp:** 100 signals combining three different frequencies (440Hz, 880Hz, 1320Hz) with random combinations, random amplitudes and random shapes(sin, square, sawtooth).
- **panFluteBigDataset:** 100 signals with random combinations of three different notes from the panflute dataset.
- **panFluteBigDataset7freq:** 100 signals with random combinations of 7 diffleent frequencies from the panflute dataset.
- **pianoBigDataset:** 100 signals with random combinations of three different notes from the piano dataset.
- **pianoPanFlute:** 100 signals with random combinations of panflute and piano. 3 frequencies.

Conditioning files

The same 4 files have been used to generate the conditioned signals:

- lc_train0.txt
- lc_gen0_16000.txt
- lc_gen1_24000.txt
- lc_gen2_16000.txt
- lc_gen3_48000.txt
- lc_gen4_72000.txt

The expected ouputs then are:

In [43]:

```
lc_train0, sr = librosa.load('corpus/Analysis/lc_train0.wav')
ipd.Audio(lc_train0, rate=sr)
```

Out[43]:

Your browser does not support the audio element.

In [44]:

```
lc_gen0, sr = librosa.load('corpus/Analysis/lc_gen0_16000.wav')
ipd.Audio(lc_gen0, rate=sr)
```

Out[44]:

Your browser does not support the audio element.

In [45]:

```
lc_gen1, sr = librosa.load('corpus/Analysis/lc_gen1_24000.wav')
ipd.Audio(lc_gen1, rate=sr)
```

Out[45]:

Your browser does not support the audio element.

In [46]:

```
lc_gen2, sr = librosa.load('corpus/Analysis/lc_gen2_16000.wav')
ipd.Audio(lc_gen2, rate=sr)
```

Out[46]:

Your browser does not support the audio element.

In [47]:

```
lc_gen3, sr = librosa.load('corpus/Analysis/lc_gen3_48000.wav')
ipd.Audio(lc_gen3, rate=sr)
```

Out[47]:

Your browser does not support the audio element.

In [48]:

```
lc_gen4, sr = librosa.load('corpus/Analysis/lc_gen4_72000.wav')
ipd.Audio(lc_gen4, rate=sr)
```

Out[48]:

Your browser does not support the audio element.

How to train?

```
CUDA_VISIBLE_DEVICES=3 python train.py --data_dir=corpus/pianoPanFlute/ --num_steps=3000 --silence_threshold=0 --lc_channels=True
```

How to generate?

```
CUDA_VISIBLE_DEVICES=3 python generate.py --samples 24000 --wav_out_path=generatedSignals/general/localTrainBigDatasetShapeAmp/shapeAmp_16_2999_train0.wav ./logdir/train/2018-05-13T06-14-23/model.ckpt-2999 --lc_channels=9 --lc_cardinality=3 --labels=./corpus/localTrain/lc_train0.txt
```

Where:

- `--samples` is the number of samples that we want to generate. Has to be the same lenght of the labels file, or smaller.
- `--lc_channels` comes from the analysis done during the training and is printed before starting the training.
- `--lc_cardinality`number of different categories (frequencies) that are detected. Is printed before starting the training.
- `--labels` directory containing the labels. Has to be a .txt file of this format.

Generated Files

- **LocalTrainBigDatset_noAmp**

In [49]:

```
train0, sr = librosa.load('generatedSignals/general/localTrainBigDataset_noAmp/randomBig_16_499_train0.wav')
ipd.Audio(train0, rate=sr)
```

Out[49]:

Your browser does not support the audio element.

In [50]:

```
gen0, sr = librosa.load('generatedSignals/general/localTrainBigDataset_noAmp/randomBig_16_499_gen0.wav')
ipd.Audio(gen0, rate=sr)
```

Out[50]:

Your browser does not support the audio element.

In [51]:

```
gen1, sr = librosa.load('generatedSignals/general/localTrainBigDataset_noAmp/randomBig_16_499_gen1.wav')
ipd.Audio(gen1, rate=sr)
```

Out[51]:

Your browser does not support the audio element.

In [52]:

```
gen2, sr = librosa.load('generatedSignals/general/localTrainBigDataset_noAmp/randomBig_16_499_gen2.wav')
ipd.Audio(gen2, rate=sr)
```

Out[52]:

Your browser does not support the audio element.

In [53]:

```
gen3, sr = librosa.load('generatedSignals/general/localTrainBigDataset_noAmp/randomBig_16_499_gen3.wav')
ipd.Audio(gen3, rate=sr)
```

Out[53]:

Your browser does not support the audio element.

In [54]:

```
gen4, sr = librosa.load('generatedSignals/general/localTrainBigDataset_noAmp/random_16_499_gen4.wav')
ipd.Audio(gen4, rate=sr)
```

Out[54]:

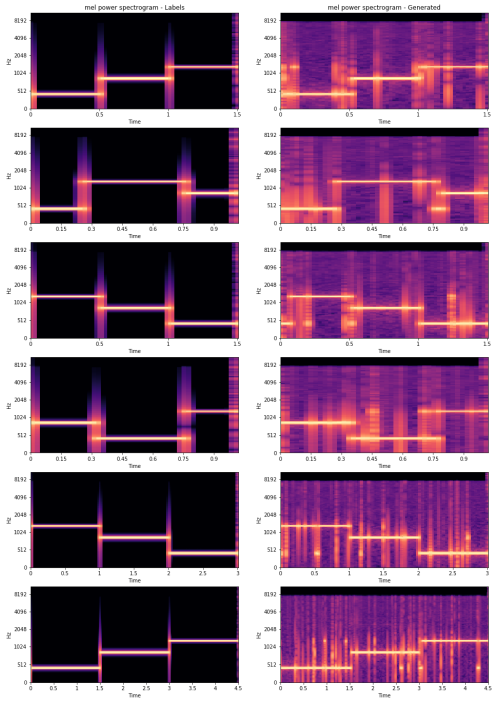
Your browser does not support the audio element.

In [55]:

```
# mel-scaled power (energy-squared) spectrogram
Strain0 = librosa.feature.melspectrogram(train0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Strain0 = librosa.power_to_db(Strain0, ref=np.max)
Slabeltrain0 = librosa.feature.melspectrogram(lc_train0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabeltrain0 = librosa.power_to_db(Slabeltrain0, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen0 = librosa.feature.melspectrogram(gen0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen0 = librosa.power_to_db(Sgen0, ref=np.max)
Slabelgen0 = librosa.feature.melspectrogram(lc_gen0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen0 = librosa.power_to_db(Slabelgen0, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen1 = librosa.feature.melspectrogram(gen1, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen1 = librosa.power_to_db(Sgen1, ref=np.max)
Slabelgen1 = librosa.feature.melspectrogram(lc_gen1, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen1 = librosa.power_to_db(Slabelgen1, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen2 = librosa.feature.melspectrogram(gen2, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen2 = librosa.power_to_db(Sgen2, ref=np.max)
Slabelgen2 = librosa.feature.melspectrogram(lc_gen2, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen2 = librosa.power_to_db(Slabelgen2, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen3 = librosa.feature.melspectrogram(gen3, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen3 = librosa.power_to_db(Sgen3, ref=np.max)
Slabelgen3 = librosa.feature.melspectrogram(lc_gen3, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen3 = librosa.power_to_db(Slabelgen3, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen4 = librosa.feature.melspectrogram(gen4, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen4 = librosa.power_to_db(Sgen4, ref=np.max)
Slabelgen4 = librosa.feature.melspectrogram(lc_gen4, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen4 = librosa.power_to_db(Slabelgen4, ref=np.max)

f, axs = plt.subplots(6,2,figsize=(16,24))
plt.subplot(6, 2, 1)
plt.title('mel power spectrogram - Labels')
librosa.display.specshow(log_Slabeltrain0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 2)
librosa.display.specshow(log_Strain0, sr=sr, x_axis='time', y_axis='mel')
plt.title('mel power spectrogram - Generated')
plt.subplot(6, 2, 3)
librosa.display.specshow(log_Slabelgen0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 4)
librosa.display.specshow(log_Sgen0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 5)
librosa.display.specshow(log_Slabelgen1, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 6)
librosa.display.specshow(log_Sgen1, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 7)
librosa.display.specshow(log_Slabelgen2, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 8)
librosa.display.specshow(log_Sgen2, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 9)
librosa.display.specshow(log_Slabelgen3, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 10)
librosa.display.specshow(log_Sgen3, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 11)
librosa.display.specshow(log_Slabelgen4, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 12)
librosa.display.specshow(log_Sgen4, sr=sr, x_axis='time', y_axis='mel')
```

Out[55]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1c1f549d68>
```



- **LocalTrainBigDataset2**

When the network is trained with 500 epoch, the resulting signal is too noisy. However, the frequencies are well detected on time.

In [56]:

```
train0, sr = librosa.load('generatedSignals/general/localTrainBigDataset2/randomBigAmp_16_499_train0.wav')
ipd.Audio(train0, rate=sr)
```

Out[56]:

Your browser does not support the audio element.

In [57]:

```
gen0, sr = librosa.load('generatedSignals/general/localTrainBigDataset2/randomBigAmp_16_499_gen0.wav')
ipd.Audio(gen0, rate=sr)
```

Out[57]:

Your browser does not support the audio element.

In [58]:

```
gen1, sr = librosa.load('generatedSignals/general/localTrainBigDataset2/randomBigAmp_16_499_gen1.wav')
ipd.Audio(gen1, rate=sr)
```

Out[58]:

Your browser does not support the audio element.

In [59]:

```
gen2, sr = librosa.load('generatedSignals/general/localTrainBigDataset2/randomBigAmp_16_499_gen2.wav')
ipd.Audio(gen2, rate=sr)
```

Out[59]:

Your browser does not support the audio element.

In [60]:

```
gen3, sr = librosa.load('generatedSignals/general/localTrainBigDataset2/randomBigAmp_16_499_gen3.wav')
ipd.Audio(gen3, rate=sr)
```

Out[60]:

Your browser does not support the audio element.

In [61]:

```
gen4, sr = librosa.load('generatedSignals/general/localTrainBigDataset2/randomBigAmp_16_499_gen4.wav')
ipd.Audio(gen4, rate=sr)
```

Out[61]:

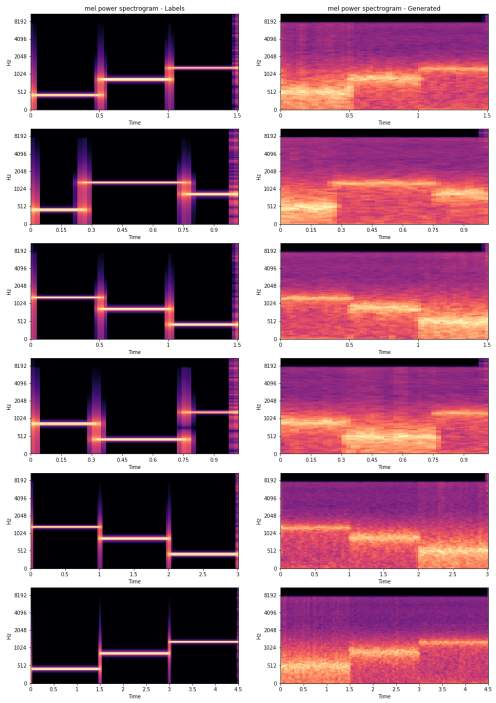
Your browser does not support the audio element.

In [62]:

```
# mel-scaled power (energy-squared) spectrogram
Strain0 = librosa.feature.melspectrogram(train0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Strain0 = librosa.power_to_db(Strain0, ref=np.max)
Slabeltrain0 = librosa.feature.melspectrogram(lc_train0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabeltrain0 = librosa.power_to_db(Slabeltrain0, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen0 = librosa.feature.melspectrogram(gen0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen0 = librosa.power_to_db(Sgen0, ref=np.max)
Slabelgen0 = librosa.feature.melspectrogram(lc_gen0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen0 = librosa.power_to_db(Slabelgen0, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen1 = librosa.feature.melspectrogram(gen1, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen1 = librosa.power_to_db(Sgen1, ref=np.max)
Slabelgen1 = librosa.feature.melspectrogram(lc_gen1, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen1 = librosa.power_to_db(Slabelgen1, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen2 = librosa.feature.melspectrogram(gen2, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen2 = librosa.power_to_db(Sgen2, ref=np.max)
Slabelgen2 = librosa.feature.melspectrogram(lc_gen2, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen2 = librosa.power_to_db(Slabelgen2, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen3 = librosa.feature.melspectrogram(gen3, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen3 = librosa.power_to_db(Sgen3, ref=np.max)
Slabelgen3 = librosa.feature.melspectrogram(lc_gen3, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen3 = librosa.power_to_db(Slabelgen3, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen4 = librosa.feature.melspectrogram(gen4, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen4 = librosa.power_to_db(Sgen4, ref=np.max)
Slabelgen4 = librosa.feature.melspectrogram(lc_gen4, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen4 = librosa.power_to_db(Slabelgen4, ref=np.max)

f, axs = plt.subplots(6,2,figsize=(16,24))
plt.subplot(6, 2, 1)
plt.title('mel power spectrogram - Labels')
librosa.display.specshow(log_Slabeltrain0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 2)
librosa.display.specshow(log_Strain0, sr=sr, x_axis='time', y_axis='mel')
plt.title('mel power spectrogram - Generated')
plt.subplot(6, 2, 3)
librosa.display.specshow(log_Slabelgen0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 4)
librosa.display.specshow(log_Sgen0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 5)
librosa.display.specshow(log_Slabelgen1, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 6)
librosa.display.specshow(log_Sgen1, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 7)
librosa.display.specshow(log_Slabelgen2, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 8)
librosa.display.specshow(log_Sgen2, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 9)
librosa.display.specshow(log_Slabelgen3, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 10)
librosa.display.specshow(log_Sgen3, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 11)
librosa.display.specshow(log_Slabelgen4, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 12)
librosa.display.specshow(log_Sgen4, sr=sr, x_axis='time', y_axis='mel')
```

Out[62]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1c231c4b38>
```



Increasing the number of iterations produces sounds less noisy, but the conditionality is not manteined.

In [63]:

```
train0, sr = librosa.load('generatedSignals/general/localTrainBigDataset2/randomBigAmp_16_5998_train0.wav')
ipd.Audio(train0, rate=sr)
```

Out[63]:

Your browser does not support the audio element.

In [64]:

```
gen0, sr = librosa.load('generatedSignals/general/localTrainBigDataset2/randomBigAmp_16_5998_gen0.wav')
ipd.Audio(gen0, rate=sr)
```

Out[64]:

Your browser does not support the audio element.

In [65]:

```
gen1, sr = librosa.load('generatedSignals/general/localTrainBigDataset2/randomBigAmp_16_5998_gen1.wav')
ipd.Audio(gen1, rate=sr)
```

Out[65]:

Your browser does not support the audio element.

In [66]:

```
gen2, sr = librosa.load('generatedSignals/general/localTrainBigDataset2/randomBigAmp_16_5998_gen2.wav')
ipd.Audio(gen2, rate=sr)
```

Out[66]:

Your browser does not support the audio element.

In [67]:

```
gen3, sr = librosa.load('generatedSignals/general/localTrainBigDataset2/randomBigAmp_16_5998_gen3.wav')
ipd.Audio(gen3, rate=sr)
```

Out[67]:

Your browser does not support the audio element.

In [68]:

```
gen4, sr = librosa.load('generatedSignals/general/localTrainBigDataset2/randomBigAmp_16_2999_gen4.wav')
ipd.Audio(gen4, rate=sr)
```

Out[68]:

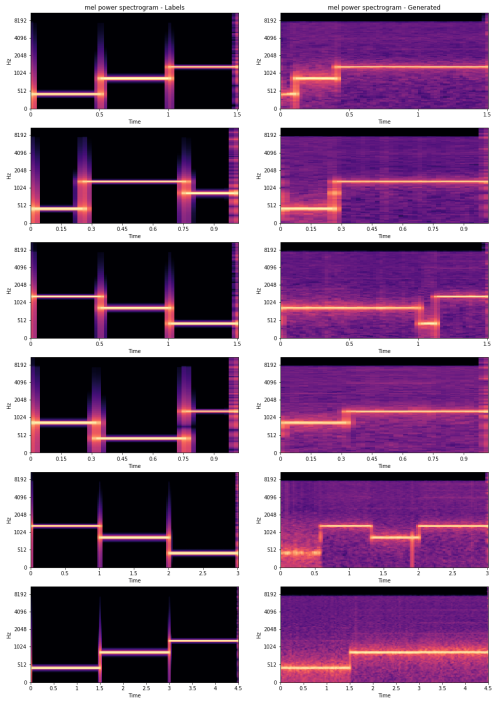
Your browser does not support the audio element.

In [69]:

```
# mel-scaled power (energy-squared) spectrogram
Strain0 = librosa.feature.melspectrogram(train0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Strain0 = librosa.power_to_db(Strain0, ref=np.max)
Slabeltrain0 = librosa.feature.melspectrogram(lc_train0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabeltrain0 = librosa.power_to_db(Slabeltrain0, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen0 = librosa.feature.melspectrogram(gen0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen0 = librosa.power_to_db(Sgen0, ref=np.max)
Slabelgen0 = librosa.feature.melspectrogram(lc_gen0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen0 = librosa.power_to_db(Slabelgen0, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen1 = librosa.feature.melspectrogram(gen1, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen1 = librosa.power_to_db(Sgen1, ref=np.max)
Slabelgen1 = librosa.feature.melspectrogram(lc_gen1, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen1 = librosa.power_to_db(Slabelgen1, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen2 = librosa.feature.melspectrogram(gen2, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen2 = librosa.power_to_db(Sgen2, ref=np.max)
Slabelgen2 = librosa.feature.melspectrogram(lc_gen2, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen2 = librosa.power_to_db(Slabelgen2, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen3 = librosa.feature.melspectrogram(gen3, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen3 = librosa.power_to_db(Sgen3, ref=np.max)
Slabelgen3 = librosa.feature.melspectrogram(lc_gen3, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen3 = librosa.power_to_db(Slabelgen3, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen4 = librosa.feature.melspectrogram(gen4, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen4 = librosa.power_to_db(Sgen4, ref=np.max)
Slabelgen4 = librosa.feature.melspectrogram(lc_gen4, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen4 = librosa.power_to_db(Slabelgen4, ref=np.max)

f, axs = plt.subplots(6,2,figsize=(16,24))
plt.subplot(6, 2, 1)
plt.title('mel power spectrogram - Labels')
librosa.display.specshow(log_Slabeltrain0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 2)
librosa.display.specshow(log_Strain0, sr=sr, x_axis='time', y_axis='mel')
plt.title('mel power spectrogram - Generated')
plt.subplot(6, 2, 3)
librosa.display.specshow(log_Slabelgen0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 4)
librosa.display.specshow(log_Sgen0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 5)
librosa.display.specshow(log_Slabelgen1, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 6)
librosa.display.specshow(log_Sgen1, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 7)
librosa.display.specshow(log_Slabelgen2, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 8)
librosa.display.specshow(log_Sgen2, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 9)
librosa.display.specshow(log_Slabelgen3, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 10)
librosa.display.specshow(log_Sgen3, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 11)
librosa.display.specshow(log_Slabelgen4, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 12)
librosa.display.specshow(log_Sgen4, sr=sr, x_axis='time', y_axis='mel')
```

Out[69]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1c23b9bf98>
```



- **localTrainBigDatasetShapeAmp:**

In [70]:

```
train0, sr = librosa.load('generatedSignals/general/localTrainBigDatasetShapeAmp/shapeAmp_16_2999_train0.wav')
ipd.Audio(train0, rate=sr)
```

Out[70]:

Your browser does not support the audio element.

In [71]:

```
gen0, sr = librosa.load('generatedSignals/general/localTrainBigDatasetShapeAmp/shapeAmp_16_2999_gen0.wav')
ipd.Audio(gen0, rate=sr)
```

Out[71]:

Your browser does not support the audio element.

In [72]:

```
gen1, sr = librosa.load('generatedSignals/general/localTrainBigDatasetShapeAmp/shapeAmp_16_2999_gen1.wav')
ipd.Audio(gen1, rate=sr)
```

Out[72]:

Your browser does not support the audio element.

In [73]:

```
gen2, sr = librosa.load('generatedSignals/general/localTrainBigDatasetShapeAmp/shapeAmp_16_2999_gen2.wav')
ipd.Audio(gen2, rate=sr)
```

Out[73]:

Your browser does not support the audio element.

In [74]:

```
gen3, sr = librosa.load('generatedSignals/general/localTrainBigDatasetShapeAmp/shapeAmp_16_2999_gen3.wav')
ipd.Audio(gen3, rate=sr)
```

Out[74]:

Your browser does not support the audio element.

In [75]:

```
gen4, sr = librosa.load('generatedSignals/general/localTrainBigDatasetShapeAmp/shapeAmp_16_2999_gen4.wav')
ipd.Audio(gen4, rate=sr)
```

Out[75]:

Your browser does not support the audio element.

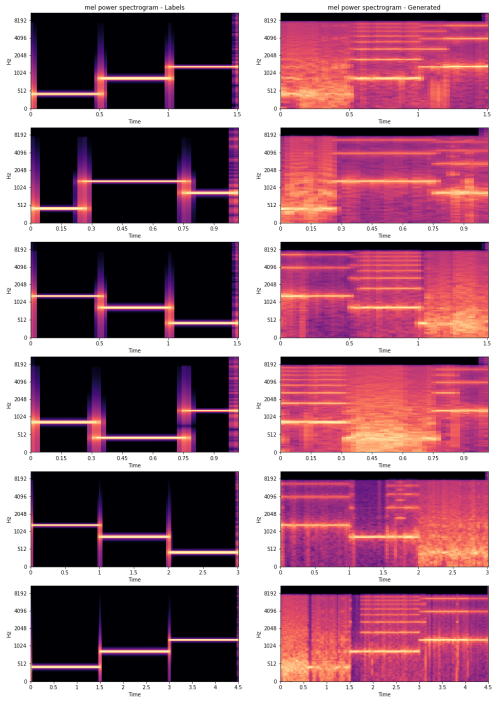
In [76]:

```
# mel-scaled power (energy-squared) spectrogram
Strain0 = librosa.feature.melspectrogram(train0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Strain0 = librosa.power_to_db(Strain0, ref=np.max)
Slabeltrain0 = librosa.feature.melspectrogram(lc_train0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabeltrain0 = librosa.power_to_db(Slabeltrain0, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen0 = librosa.feature.melspectrogram(gen0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen0 = librosa.power_to_db(Sgen0, ref=np.max)
Slabelgen0 = librosa.feature.melspectrogram(lc_gen0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen0 = librosa.power_to_db(Slabelgen0, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen1 = librosa.feature.melspectrogram(gen1, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen1 = librosa.power_to_db(Sgen1, ref=np.max)
Slabelgen1 = librosa.feature.melspectrogram(lc_gen1, sr=sr, n_mels=128)
```

```
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen1 = librosa.power_to_db(Slabelgen1, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen2 = librosa.feature.melspectrogram(gen2, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen2 = librosa.power_to_db(Sgen2, ref=np.max)
Slabelgen2 = librosa.feature.melspectrogram(lc_gen2, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen2 = librosa.power_to_db(Slabelgen2, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen3 = librosa.feature.melspectrogram(gen3, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen3 = librosa.power_to_db(Sgen3, ref=np.max)
Slabelgen3 = librosa.feature.melspectrogram(lc_gen3, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen3 = librosa.power_to_db(Slabelgen3, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen4 = librosa.feature.melspectrogram(gen4, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen4 = librosa.power_to_db(Sgen4, ref=np.max)
Slabelgen4 = librosa.feature.melspectrogram(lc_gen4, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen4 = librosa.power_to_db(Slabelgen4, ref=np.max)

f, axs = plt.subplots(6,2,figsize=(16,24))
plt.subplot(6, 2, 1)
plt.title('mel power spectrogram - Labels')
librosa.display.specshow(log_Slabeltrain0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 2)
librosa.display.specshow(log_Strain0, sr=sr, x_axis='time', y_axis='mel')
plt.title('mel power spectrogram - Generated')
plt.subplot(6, 2, 3)
librosa.display.specshow(log_Slabelgen0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 4)
librosa.display.specshow(log_Sgen0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 5)
librosa.display.specshow(log_Slabelgen1, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 6)
librosa.display.specshow(log_Sgen1, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 7)
librosa.display.specshow(log_Slabelgen2, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 8)
librosa.display.specshow(log_Sgen2, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 9)
librosa.display.specshow(log_Slabelgen3, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 10)
librosa.display.specshow(log_Sgen3, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 11)
librosa.display.specshow(log_Slabelgen4, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 12)
librosa.display.specshow(log_Sgen4, sr=sr, x_axis='time', y_axis='mel')
```

Out[76]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1c25f2bb00>
```



- **panFluteBigDataset**

Labels

In [77]:

```
lc_train0, sr = librosa.load('corpus/Analysis/lc_train0_flute.wav')
ipd.Audio(lc_train0, rate=sr)
```

Out[77]:

Your browser does not support the audio element.

In [78]:

```
lc_gen0, sr = librosa.load('corpus/Analysis/lc_gen0_flute.wav')
ipd.Audio(lc_gen0, rate=sr)
```

Out[78]:

Your browser does not support the audio element.

In [79]:

```
lc_gen1, sr = librosa.load('corpus/Analysis/lc_gen1_flute.wav')
ipd.Audio(lc_gen1, rate=sr)
```

Out[79]:

Your browser does not support the audio element.

In [80]:

```
lc_gen2, sr = librosa.load('corpus/Analysis/lc_gen2_flute.wav')
ipd.Audio(lc_gen2, rate=sr)
```

Out[80]:

Your browser does not support the audio element.

In [81]:

```
lc_gen3, sr = librosa.load('corpus/Analysis/lc_gen3_flute.wav')
ipd.Audio(lc_gen3, rate=sr)
```

Out[81]:

Your browser does not support the audio element.

In [82]:

```
lc_gen4, sr = librosa.load('corpus/Analysis/lc_gen4_flute.wav')
ipd.Audio(lc_gen4, rate=sr)
```

Out[82]:

Your browser does not support the audio element.

In [83]:

```
lc_scale, sr = librosa.load('corpus/Analysis/7freq_56000.wav')
ipd.Audio(lc_scale, rate=sr)
```

Out[83]:

Your browser does not support the audio element.

Generated

In [84]:

```
train0, sr = librosa.load('generatedSignals/general/panFluteBigDataset/panFluteBig_16_2999_train0_2.wav')
ipd.Audio(train0, rate=sr)
```

Out[84]:

Your browser does not support the audio element.

In [85]:

```
gen0, sr = librosa.load('generatedSignals/general/panFluteBigDataset/panFluteBig_16_2999_gen0_2.wav')
ipd.Audio(gen0, rate=sr)
```

Out[85]:

Your browser does not support the audio element.

In [86]:

```
gen1, sr = librosa.load('generatedSignals/general/panFluteBigDataset/panFluteBig_16_2999_gen1_2.wav')
ipd.Audio(gen1, rate=sr)
```

Out[86]:

Your browser does not support the audio element.

In [87]:

```
gen2, sr = librosa.load('generatedSignals/general/panFluteBigDataset/panFluteBig_16_2999_gen2_2.wav')
ipd.Audio(gen2, rate=sr)
```

Out[87]:

Your browser does not support the audio element.

In [88]:

```
gen3, sr = librosa.load('generatedSignals/general/panFluteBigDataset/panFluteBig_16_2999_gen3_2.wav')
ipd.Audio(gen3, rate=sr)
```

Out[88]:

Your browser does not support the audio element.

In [89]:

```
gen4, sr = librosa.load('generatedSignals/general/panFluteBigDataset/panFluteBig_16_2999_gen4_2.wav')
ipd.Audio(gen4, rate=sr)
```

Out[89]:

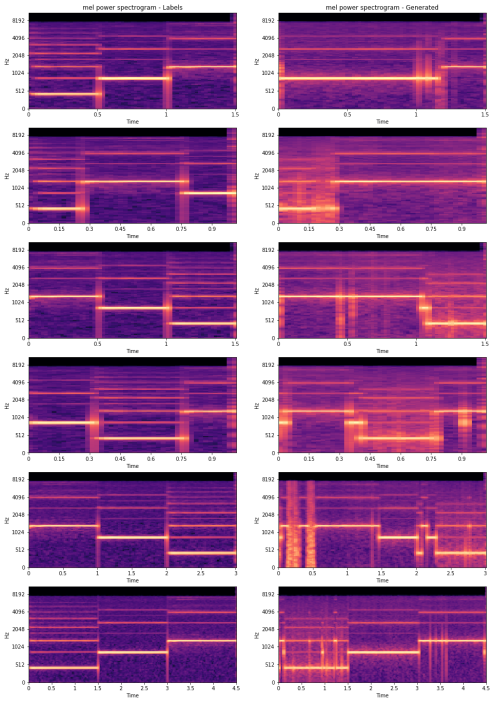
Your browser does not support the audio element.

In [90]:

```
# mel-scaled power (energy-squared) spectrogram
Strain0 = librosa.feature.melspectrogram(train0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Strain0 = librosa.power_to_db(Strain0, ref=np.max)
Slabeltrain0 = librosa.feature.melspectrogram(lc_train0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabeltrain0 = librosa.power_to_db(Slabeltrain0, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen0 = librosa.feature.melspectrogram(gen0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen0 = librosa.power_to_db(Sgen0, ref=np.max)
Slabelgen0 = librosa.feature.melspectrogram(lc_gen0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen0 = librosa.power_to_db(Slabelgen0, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen1 = librosa.feature.melspectrogram(gen1, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen1 = librosa.power_to_db(Sgen1, ref=np.max)
Slabelgen1 = librosa.feature.melspectrogram(lc_gen1, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen1 = librosa.power_to_db(Slabelgen1, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen2 = librosa.feature.melspectrogram(gen2, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen2 = librosa.power_to_db(Sgen2, ref=np.max)
Slabelgen2 = librosa.feature.melspectrogram(lc_gen2, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen2 = librosa.power_to_db(Slabelgen2, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen3 = librosa.feature.melspectrogram(gen3, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen3 = librosa.power_to_db(Sgen3, ref=np.max)
Slabelgen3 = librosa.feature.melspectrogram(lc_gen3, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen3 = librosa.power_to_db(Slabelgen3, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen4 = librosa.feature.melspectrogram(gen4, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen4 = librosa.power_to_db(Sgen4, ref=np.max)
Slabelgen4 = librosa.feature.melspectrogram(lc_gen4, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen4 = librosa.power_to_db(Slabelgen4, ref=np.max)

f, axs = plt.subplots(6,2,figsize=(16,24))
plt.subplot(6, 2, 1)
plt.title('mel power spectrogram - Labels')
librosa.display.specshow(log_Slabeltrain0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 2)
librosa.display.specshow(log_Strain0, sr=sr, x_axis='time', y_axis='mel')
plt.title('mel power spectrogram - Generated')
plt.subplot(6, 2, 3)
librosa.display.specshow(log_Slabelgen0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 4)
librosa.display.specshow(log_Sgen0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 5)
librosa.display.specshow(log_Slabelgen1, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 6)
librosa.display.specshow(log_Sgen1, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 7)
librosa.display.specshow(log_Slabelgen2, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 8)
librosa.display.specshow(log_Sgen2, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 9)
librosa.display.specshow(log_Slabelgen3, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 10)
librosa.display.specshow(log_Sgen3, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 11)
librosa.display.specshow(log_Slabelgen4, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 12)
librosa.display.specshow(log_Sgen4, sr=sr, x_axis='time', y_axis='mel')
```

Out[90]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1c1e67cdd8>
```



- **panFluteBigDataset7freq**

In [91]:

```
train0, sr = librosa.load('generatedSignals/general/panFluteBigDataset7freq/panFluteBig7freq_16_2999_train0_3.wav')
ipd.Audio(train0, rate=sr)
```

Out[91]:

Your browser does not support the audio element.

In [92]:

```
gen0, sr = librosa.load('generatedSignals/general/panFluteBigDataset7freq/panFluteBig7freq_16_2999_gen0_3.wav')
ipd.Audio(gen0, rate=sr)
```

Out[92]:

Your browser does not support the audio element.

In [93]:

```
gen1, sr = librosa.load('generatedSignals/general/panFluteBigDataset7freq/panFluteBig7freq_16_2999_gen1_3.wav')
ipd.Audio(gen1, rate=sr)
```

Out[93]:

Your browser does not support the audio element.

In [94]:

```
gen2, sr = librosa.load('generatedSignals/general/panFluteBigDataset7freq/panFluteBig7freq_16_2999_gen2_3.wav')
ipd.Audio(gen2, rate=sr)
```

Out[94]:

Your browser does not support the audio element.

In [95]:

```
gen3, sr = librosa.load('generatedSignals/general/panFluteBigDataset7freq/panFluteBig7freq_16_2999_gen3_3.wav')
ipd.Audio(gen3, rate=sr)
```

Out[95]:

Your browser does not support the audio element.

In [96]:

```
gen4, sr = librosa.load('generatedSignals/general/panFluteBigDataset7freq/panFluteBig7freq_16_2999_gen4_4.wav')
ipd.Audio(gen4, rate=sr)
```

Out[96]:

Your browser does not support the audio element.

In [97]:

```
scale, sr = librosa.load('generatedSignals/general/panFluteBigDataset7freq/panFluteBig7freq_16_2999_scale4.wav')
ipd.Audio(scale, rate=sr)
```

Out[97]:

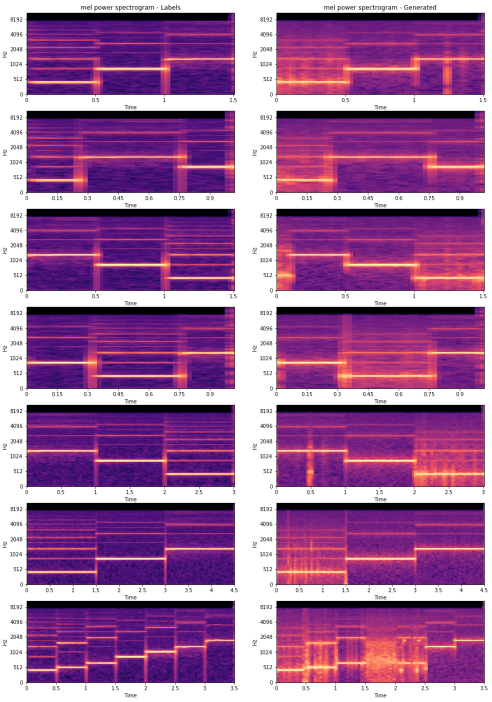
Your browser does not support the audio element.

In [98]:

```
# mel-scaled power (energy-squared) spectrogram
Strain0 = librosa.feature.melspectrogram(train0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Strain0 = librosa.power_to_db(Strain0, ref=np.max)
Slabeltrain0 = librosa.feature.melspectrogram(lc_train0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabeltrain0 = librosa.power_to_db(Slabeltrain0, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen0 = librosa.feature.melspectrogram(gen0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen0 = librosa.power_to_db(Sgen0, ref=np.max)
Slabelgen0 = librosa.feature.melspectrogram(lc_gen0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen0 = librosa.power_to_db(Slabelgen0, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen1 = librosa.feature.melspectrogram(gen1, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen1 = librosa.power_to_db(Sgen1, ref=np.max)
Slabelgen1 = librosa.feature.melspectrogram(lc_gen1, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen1 = librosa.power_to_db(Slabelgen1, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen2 = librosa.feature.melspectrogram(gen2, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen2 = librosa.power_to_db(Sgen2, ref=np.max)
Slabelgen2 = librosa.feature.melspectrogram(lc_gen2, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen2 = librosa.power_to_db(Slabelgen2, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen3 = librosa.feature.melspectrogram(gen3, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen3 = librosa.power_to_db(Sgen3, ref=np.max)
Slabelgen3 = librosa.feature.melspectrogram(lc_gen3, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen3 = librosa.power_to_db(Slabelgen3, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen4 = librosa.feature.melspectrogram(gen4, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen4 = librosa.power_to_db(Sgen4, ref=np.max)
Slabelgen4 = librosa.feature.melspectrogram(lc_gen4, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen4 = librosa.power_to_db(Slabelgen4, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sscale = librosa.feature.melspectrogram(scale, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sscale = librosa.power_to_db(Sscale, ref=np.max)
Slabelscale = librosa.feature.melspectrogram(lc_scale, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelscale = librosa.power_to_db(Slabelscale, ref=np.max)

f, axs = plt.subplots(7,2,figsize=(16,24))
plt.subplot(7, 2, 1)
plt.title('mel power spectrogram - Labels')
librosa.display.specshow(log_Slabeltrain0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(7, 2, 2)
librosa.display.specshow(log_Strain0, sr=sr, x_axis='time', y_axis='mel')
plt.title('mel power spectrogram - Generated')
plt.subplot(7, 2, 3)
librosa.display.specshow(log_Slabelgen0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(7, 2, 4)
librosa.display.specshow(log_Sgen0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(7, 2, 5)
librosa.display.specshow(log_Slabelgen1, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(7, 2, 6)
librosa.display.specshow(log_Sgen1, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(7, 2, 7)
librosa.display.specshow(log_Slabelgen2, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(7, 2, 8)
librosa.display.specshow(log_Sgen2, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(7, 2, 9)
librosa.display.specshow(log_Slabelgen3, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(7, 2, 10)
librosa.display.specshow(log_Sgen3, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(7, 2, 11)
librosa.display.specshow(log_Slabelgen4, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(7, 2, 12)
librosa.display.specshow(log_Sgen4, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(7, 2, 13)
librosa.display.specshow(log_Slabelscale, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(7, 2, 14)
librosa.display.specshow(log_Sscale, sr=sr, x_axis='time', y_axis='mel')
```

Out[98]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1c28d59470>
```



- **pianoBigDataset**

Piano Labels

In [99]:

```
lc_train0, sr = librosa.load('corpus/Analysis/lc_train0_piano.wav')
ipd.Audio(lc_train0, rate=sr)
```

Out[99]:

Your browser does not support the audio element.

In [100]:

```
lc_gen0, sr = librosa.load('corpus/Analysis/lc_gen0_piano.wav')
ipd.Audio(lc_gen0, rate=sr)
```

Out[100]:

Your browser does not support the audio element.

In [101]:

```
lc_gen1, sr = librosa.load('corpus/Analysis/lc_gen1_piano.wav')
ipd.Audio(lc_gen1, rate=sr)
```

Out[101]:

Your browser does not support the audio element.

In [102]:

```
lc_gen2, sr = librosa.load('corpus/Analysis/lc_gen2_piano.wav')
ipd.Audio(lc_gen2, rate=sr)
```

Out[102]:

Your browser does not support the audio element.

In [103]:

```
lc_gen3, sr = librosa.load('corpus/Analysis/lc_gen3_piano.wav')
ipd.Audio(lc_gen3, rate=sr)
```

Out[103]:

Your browser does not support the audio element.

In [104]:

```
lc_gen4, sr = librosa.load('corpus/Analysis/lc_gen4_piano.wav')
ipd.Audio(lc_gen4, rate=sr)
```

Out[104]:

Your browser does not support the audio element.

Generated Signals

In [105]:

```
train0, sr = librosa.load('generatedSignals/general/pianoBigDataset/piano_62_2998_train_0_2.wav')
ipd.Audio(train0, rate=sr)
```

Out[105]:

Your browser does not support the audio element.

In [106]:

```
gen0, sr = librosa.load('generatedSignals/general/pianoBigDataset/piano_62_2998_gen0_2.wav')
ipd.Audio(gen0, rate=sr)
```

Out[106]:

Your browser does not support the audio element.

In [107]:

```
gen1, sr = librosa.load('generatedSignals/general/pianoBigDataset/piano_62_2998_gen1_2.wav')
ipd.Audio(gen1, rate=sr)
```

Out[107]:

Your browser does not support the audio element.

In [108]:

```
gen2, sr = librosa.load('generatedSignals/general/pianoBigDataset/piano_62_2998_gen2_2.wav')
ipd.Audio(gen2, rate=sr)
```

Out[108]:

Your browser does not support the audio element.

In [109]:

```
gen3, sr = librosa.load('generatedSignals/general/pianoBigDataset/piano_62_2998_gen3_2.wav')
ipd.Audio(gen3, rate=sr)
```

Out[109]:

Your browser does not support the audio element.

In [110]:

```
gen4, sr = librosa.load('generatedSignals/general/pianoBigDataset/piano_62_2998_gen4_2.wav')
ipd.Audio(gen4, rate=sr)
```

Out[110]:

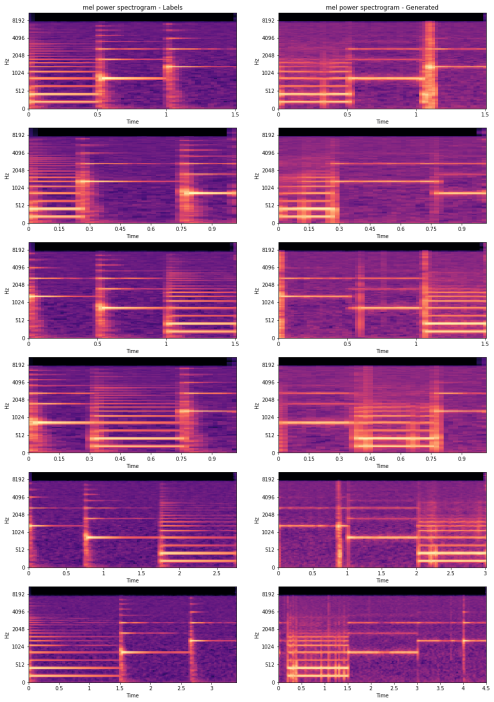
Your browser does not support the audio element.

In [111]:

```
# mel-scaled power (energy-squared) spectrogram
Strain0 = librosa.feature.melspectrogram(train0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Strain0 = librosa.power_to_db(Strain0, ref=np.max)
Slabeltrain0 = librosa.feature.melspectrogram(lc_train0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabeltrain0 = librosa.power_to_db(Slabeltrain0, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen0 = librosa.feature.melspectrogram(gen0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen0 = librosa.power_to_db(Sgen0, ref=np.max)
Slabelgen0 = librosa.feature.melspectrogram(lc_gen0, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen0 = librosa.power_to_db(Slabelgen0, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen1 = librosa.feature.melspectrogram(gen1, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen1 = librosa.power_to_db(Sgen1, ref=np.max)
Slabelgen1 = librosa.feature.melspectrogram(lc_gen1, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen1 = librosa.power_to_db(Slabelgen1, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen2 = librosa.feature.melspectrogram(gen2, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen2 = librosa.power_to_db(Sgen2, ref=np.max)
Slabelgen2 = librosa.feature.melspectrogram(lc_gen2, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen2 = librosa.power_to_db(Slabelgen2, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen3 = librosa.feature.melspectrogram(gen3, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen3 = librosa.power_to_db(Sgen3, ref=np.max)
Slabelgen3 = librosa.feature.melspectrogram(lc_gen3, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen3 = librosa.power_to_db(Slabelgen3, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen4 = librosa.feature.melspectrogram(gen4, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen4 = librosa.power_to_db(Sgen4, ref=np.max)
Slabelgen4 = librosa.feature.melspectrogram(lc_gen4, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen4 = librosa.power_to_db(Slabelgen4, ref=np.max)

f, axs = plt.subplots(6,2,figsize=(16,24))
plt.subplot(6, 2, 1)
plt.title('mel power spectrogram - Labels')
librosa.display.specshow(log_Slabeltrain0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 2)
librosa.display.specshow(log_Strain0, sr=sr, x_axis='time', y_axis='mel')
plt.title('mel power spectrogram - Generated')
plt.subplot(6, 2, 3)
librosa.display.specshow(log_Slabelgen0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 4)
librosa.display.specshow(log_Sgen0, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 5)
librosa.display.specshow(log_Slabelgen1, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 6)
librosa.display.specshow(log_Sgen1, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 7)
librosa.display.specshow(log_Slabelgen2, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 8)
librosa.display.specshow(log_Sgen2, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 9)
librosa.display.specshow(log_Slabelgen3, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 10)
librosa.display.specshow(log_Sgen3, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 11)
librosa.display.specshow(log_Slabelgen4, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(6, 2, 12)
librosa.display.specshow(log_Sgen4, sr=sr, x_axis='time', y_axis='mel')
```

Out[111]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1c295d17f0>
```



- **pianoPanFlute**

In [112]:

```
train0, sr = librosa.load('generatedSignals/general/pianoPanFlute/pianoFlute_62_2999_train_0.wav')
ipd.Audio(train0, rate=sr)
```

Out[112]:

Your browser does not support the audio element.

In [113]:

```
gen0, sr = librosa.load('generatedSignals/general/pianoPanFlute/pianoFlute_62_2999_gen0.wav')
ipd.Audio(gen0, rate=sr)
```

Out[113]:

Your browser does not support the audio element.

In [114]:

```
gen1, sr = librosa.load('generatedSignals/general/pianoPanFlute/pianoFlute_62_2999_gen1.wav')
ipd.Audio(gen1, rate=sr)
```

Out[114]:

Your browser does not support the audio element.

In [115]:

```
gen0, sr = librosa.load('generatedSignals/general/pianoPanFlute/pianoFlute_62_2999_gen2.wav')
ipd.Audio(gen2, rate=sr)
```

Out[115]:

Your browser does not support the audio element.

In [116]:

```
gen3, sr = librosa.load('generatedSignals/general/pianoPanFlute/pianoFlute_62_2999_gen3.wav')
ipd.Audio(gen3, rate=sr)
```

Out[116]:

Your browser does not support the audio element.

In [117]:

```
gen4, sr = librosa.load('generatedSignals/general/pianoPanFlute/pianoFlute_62_2999_gen4.wav')
ipd.Audio(gen4, rate=sr)
```

Out[117]:

Your browser does not support the audio element.

Local Conditioning using MFCC

- **LocalTrainBigDataset2**

Labels

In [118]:

```
lc_gen3, sr = librosa.load('corpus/Analysis/lc_gen3_48000.wav')
ipd.Audio(lc_gen3, rate=sr)
```

Out[118]:

Your browser does not support the audio element.

In [119]:

```
lc_gen4, sr = librosa.load('corpus/Analysis/lc_gen4_72000.wav')
ipd.Audio(lc_gen4, rate=sr)
```

Out[119]:

Your browser does not support the audio element.

Generated Signals

In [120]:

```
gen3, sr = librosa.load('generatedSignals/mfcc/localTrainBigDataset2/mfccBigAmp_16_2999_gen3.wav')
ipd.Audio(gen3, rate=sr)
```

Out[120]:

Your browser does not support the audio element.

In [121]:

```
gen4, sr = librosa.load('generatedSignals/mfcc/localTrainBigDataset2/mfccBigAmp_16_2999_gen4.wav')
ipd.Audio(gen4, rate=sr)
```

Out[121]:

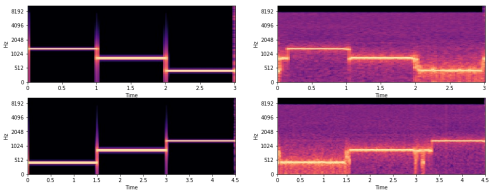
Your browser does not support the audio element.

In [122]:

```
# mel-scaled power (energy-squared) spectrogram
Sgen3 = librosa.feature.melspectrogram(gen3, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen3 = librosa.power_to_db(Sgen3, ref=np.max)
Slabelgen3 = librosa.feature.melspectrogram(lc_gen3, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen3 = librosa.power_to_db(Slabelgen3, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen4 = librosa.feature.melspectrogram(gen4, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen4 = librosa.power_to_db(Sgen4, ref=np.max)
Slabelgen4 = librosa.feature.melspectrogram(lc_gen4, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen4 = librosa.power_to_db(Slabelgen4, ref=np.max)

f, axs = plt.subplots(2,2,figsize=(16,6))
plt.subplot(2, 2, 1)
librosa.display.specshow(log_Slabelgen3, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(2, 2, 2)
librosa.display.specshow(log_Sgen3, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(2, 2, 3)
librosa.display.specshow(log_Slabelgen4, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(2, 2, 4)
librosa.display.specshow(log_Sgen4, sr=sr, x_axis='time', y_axis='mel')
```

Out[122]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1c255d10b8>
```



- **pianoBigDataset**

Labels

In [123]:

```
lc_gen3, sr = librosa.load('corpus/Analysis/lc_gen3_piano.wav')
ipd.Audio(lc_gen3, rate=sr)
```

Out[123]:

Your browser does not support the audio element.

In [124]:

```
lc_gen4, sr = librosa.load('corpus/Analysis/lc_gen4_piano.wav')
ipd.Audio(lc_gen4, rate=sr)
```

Out[124]:

Your browser does not support the audio element.

Generated

In [125]:

```
gen3, sr = librosa.load('generatedSignals/mfcc/pianoBigDataset/mfccPianoBig_62_256_9999_gen3.wav')
ipd.Audio(gen3, rate=sr)
```

Out[125]:

Your browser does not support the audio element.

In [126]:

```
gen4, sr = librosa.load('generatedSignals/mfcc/pianoBigDataset/mfccPianoBig_62_256_9999_gen4.wav')
ipd.Audio(gen4, rate=sr)
```

Out[126]:

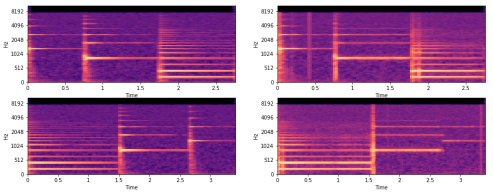
Your browser does not support the audio element.

In [127]:

```
# mel-scaled power (energy-squared) spectrogram
Sgen3 = librosa.feature.melspectrogram(gen3, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen3 = librosa.power_to_db(Sgen3, ref=np.max)
Slabelgen3 = librosa.feature.melspectrogram(lc_gen3, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen3 = librosa.power_to_db(Slabelgen3, ref=np.max)

# mel-scaled power (energy-squared) spectrogram
Sgen4 = librosa.feature.melspectrogram(gen4, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Sgen4 = librosa.power_to_db(Sgen4, ref=np.max)
Slabelgen4 = librosa.feature.melspectrogram(lc_gen4, sr=sr, n_mels=128)
# Convert to log scale (dB). We'll use the peak power (max) as reference.
log_Slabelgen4 = librosa.power_to_db(Slabelgen4, ref=np.max)

f, axs = plt.subplots(2,2,figsize=(16,6))
plt.subplot(2, 2, 1)
librosa.display.specshow(log_Slabelgen3, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(2, 2, 2)
librosa.display.specshow(log_Sgen3, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(2, 2, 3)
librosa.display.specshow(log_Slabelgen4, sr=sr, x_axis='time', y_axis='mel')
plt.subplot(2, 2, 4)
librosa.display.specshow(log_Sgen4, sr=sr, x_axis='time', y_axis='mel')
```

Out[127]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1c27e4c4a8>
```



Pitch Transformation

I wanted to see if modifying the target in the training wavenet was able to learn a transformation. For example feeding the network with a 440 Hz signal and a target consisting of a 880 Hz, I wanted to see if wavenet could learn the transformations. Although the loss function goes below 0.02 with 100 epoch, looks like in wavenet the input have to match the output. More information about this here

This implementation could be seen below the branch pitchTransformation link

In this implementation, when I pass as a target the same signal as the input, I achieve a good result, however, when the target is a signal with a different frequency, wavenet is not able to learn that.

Be Careful, LOUD NOISE!

In [128]:

```
x, sr = librosa.load('generatedSignals/pitch1.wav')
ipd.Audio(x, rate=sr)
```

Out[128]:

Your browser does not support the audio element.

Real sounds

Bunch of experiments with two different real instruments.

Drum Samples Dataset

With this reduced wavenet of a receptive field of 47 bad results are achieved working with real sounds. In order to solve that, the receptive field has been increased to ... and the original dataset has been reduced to 29 items split in 5 categories (Kick Drum, Snare, Tom, Hit Hat and Cymbal). Later the neural network is trained with 5000 epoch.

Some results are presented below:

In [129]:
```
x, sr = librosa.load('generatedSignals/cymbal_0_4999.wav')
ipd.Audio(x, rate=sr)
```

Out[129]:

Your browser does not support the audio element.

In [130]:
```
x, sr = librosa.load('generatedSignals/kick_1_4999.wav')
ipd.Audio(x, rate=sr)
```

Out[130]:

Your browser does not support the audio element.

In [131]:
```
x, sr = librosa.load('generatedSignals/snare_2_4999.wav')
ipd.Audio(x, rate=sr)
```

Out[131]:

Your browser does not support the audio element.

In [132]:
```
x, sr = librosa.load('generatedSignals/hithat_3_4999.wav')
ipd.Audio(x, rate=sr)
```

Out[132]:

Your browser does not support the audio element.

In [133]:
```
x, sr = librosa.load('generatedSignals/tom_4_4999.wav')
ipd.Audio(x, rate=sr)
```

Out[133]:

Your browser does not support the audio element.

Even with increasing the receptive field and the number of iterations is still impossible to recreate a good drum sound. Because of that, we decided to train the neural network now with only one signal, and try to generate that signal. When that would be achieved, we will apply again global conditioning

Cymbal

```
python train.py --data_dir=corpus/drumsamplesReduced2/ --num_steps=5000 --silence_threshold=0.0000001
```

```
python generate.py --wav_out_path=cymbal_nogc_4999.wav --samples=32000 logdir/train/2018-04-15T09-32-07/model.ckpt-4999
```

In [134]:
```
x, sr = librosa.load('generatedSignals/cymbal_nogc_4999.wav')
ipd.Audio(x, rate=sr)
```

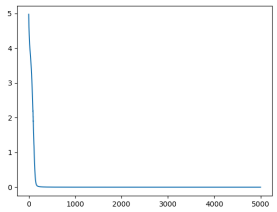
Out[134]:

Your browser does not support the audio element.

Even though the loss function became 0 with a few iterations, the sound quality is not good.

In [135]:
```
from IPython.display import Image
Image("corpus/drumsamplesReduced2/loss.png")
```

Out[135]:




```
python train.py --data_dir=corpus/kickDrum/ --num_steps=5000 --silence_threshold=0.0000001
```

```
python generate.py --wav_out_path=generatedSignals/kickDrum_4999.wav --samples 16000 ./logdir/train/2018-04-15T13-32-01/model.ckpt-4999
```

Kick Drum

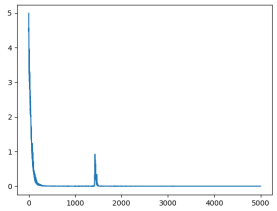
In [136]:
```
x, sr = librosa.load('generatedSignals/kickDrum_4999.wav')
ipd.Audio(x, rate=sr)
```

Out[136]:

Your browser does not support the audio element.

In [137]:
```
from IPython.display import Image
Image("corpus/kickDrum/loss.png")
```

Out[137]:



Snare

```
python train.py --data_dir=corpus/snare/ --num_steps=5000 --silence_threshold=0.01
```

```
python generate.py --wav_out_path=snare_256_4999.wav --samples 16000 ./logdir/train/2018-04-15T14-18-48/model.ckpt-4999
```

```
quantization_channels = 256
```

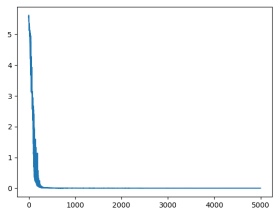
In [138]:
```
x, sr = librosa.load('generatedSignals/snare_256_4999.wav')
ipd.Audio(x, rate=sr)
```

Out[138]:

Your browser does not support the audio element.

In [139]:
```
from IPython.display import Image
Image("corpus/snare/loss.png")
```

Out[139]:



Hit Hat

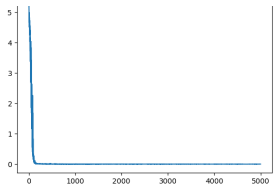
In [140]:
```
x, sr = librosa.load('generatedSignals/hithat_256_4999.wav')
ipd.Audio(x, rate=sr)
```

Out[140]:

Your browser does not support the audio element.

In [141]:
```
from IPython.display import Image
Image("corpus/hithat/loss.png")
```

Out[141]:



Tom

In [142]:

```
x, sr = librosa.load('generatedSignals/tom_256_4998.wav')
ipd.Audio(x, rate=sr)
```

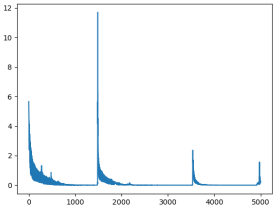
Out[142]:

Your browser does not support the audio element.

In [143]:

```
from IPython.display import Image
Image("corpus/tom/loss.png")
```

Out[143]:



Acoustic Scenes Dataset

Here I selected 144 recordings from water (beach and lakes) from the TUT Acoustic Scenes 2017 Dataset [link] and I trained wavenet with a receptive field of 5117 and 10000 epoch.

In [144]:

```
x, sr = librosa.load('generatedSignals/water_9999.wav')
ipd.Audio(x, rate=sr)
```

Out[144]:

Your browser does not support the audio element.