DrumGAN

Adversarial synthesis of drum sounds - DCGAN with dilated convolutions

Victor Falini

Spring 2021

15 ECTS
STAN40: M.Sc in Statistics
Department of Statistics - Lund University
Supervised by Peter Gustafsson

Abstract

This paper faces the problem of audio synthesis with Generative Adversarial Networks (GAN), with an attempt to create novel, original high-quality samples of drums that could be used within the realm of music production. My results show that it is possible to create high-quality drum samples with architecture such as DCGAN (Deep Convolutional GAN) and that using causal and dilated convolutions is a viable approach, while not making clear if this approach is significantly better than the one of using standard convolutions. It also shows that 1-d transpose convolutions can be substituted with nearest neighbour upsampling followed by regular 1-d convolutions for GANs that generate one-dimensional data. In the final discussion the idea of initializing generator weights in a strategic way in order to increase GAN training stability is introduced.

Index

1 Introduction

Being able to synthesize new, original sounds is something with a large amount of practical applications within the sound design field and can be used both for movies, music production and even possibly for video games. An example of this is Deep Adaptive Music (DAM) which has as a goal of producing music in real-time to fit and dynamically adapt to the current emotional state present in a video game (Velardo 2019). Another example is Jukebox by Dhariwal et al. (2020) that can generate full length songs if just given a primer with amazing results. Altough the quality of the audio is not sufficient for Jukebox to be used for creating new records, it could be very useful for musicians looking for inspiration or creative ideas.

Recently developed state of the art generative models such as WaveNet by Oord et al. (2016) and the previously mentioned Jukebox are both autoregressive generative models. They therefore require long times to generate audio, since every output has to be reinserted into the network each time to make the next prediction in the time series. For an audio file with CD quality, this means that each second of generated audio is the result of 44100 predictions that cannot be done in parallel. Another kind of popular type of generative network known as Generative Adversarial Network (GAN) has the advantage of being able to produce the entire waveform directly, meaning that audio generation can be performed in real time. While GANs are very popular and successful within the field of image generation, there are few publications that tackle the problem of generating audio with GANs.
In this paper I focus on the production of drum samples with a GAN. My work is much inspired by the WaveGAN model developed by Donahue, McAuley, and Puckette (2019) which was claimed by its authors to be, at the time of its publication, the "[...]first application of GANs to unsupervised audio generation."(ibid, p.9). My approach is different and focuses more on creating high-quality audio samples that can be used in practical applications with a more rapid training phase when compared to WaveGAN. My results show that, with certain small adjustments, audio artefacts present in WaveGAN model can be avoided and it is possible to create drum audio samples that are useable in practice, within the field of music production. The technique of dilated 1d-convolutions is also used for the GAN models in this paper.

Many audio generation models use spectral image representations of the audio, obtained through a Short-time Fourier transform, that are used to train a generative model that produces new spectrograms, which are then transformed back into audio. The main reasons for the popularity of this spectrogram-based approach are that image generation is a much more

developed field than the one of one-dimensional timseries generation and that this transformation can save processing power since the image representation of the audio is often "smaller" than the original data.
I decided to directly use the raw wave data for my model, since the audio samples used for this type of data (drum sounds) are small enough for processing power problems to be no real issue. The original, unmodified data contains all information and creating a spectral image will always be an approximation. Donahue, McAuley, and Puckette (2019) also show that the spectral approach is inferior to using the waveform directly.

2 Method

To generate this novel audio I will be using Artificial Neural Networks and more specifically I will be using a Deep Convolutional Generative Adversarial Network. The purpose of this paper is not to explain all these methods in detail but for anyone wanting to learn more about these interesting methods I recommend reading *Deep Learning* by Goodfellow, Bengio, and Courville (2016), *Generative Adversarial Nets* by Goodfellow, Pouget-Abadie, et al. (2014) and other literature referenced in the method part below. All of the ideas presented below regarding neural networks and convolutional neural networks are based on knowledge taken from Deep Learning by Goodfellow, Bengio, and Courville (2016).

2.1 Artificial Neural Networks

Artificial Neural Networks are a type of machine learning method that is inspired by biological neural networks.
A example of a basic neural network structure is illustrated in Figure 1 below.

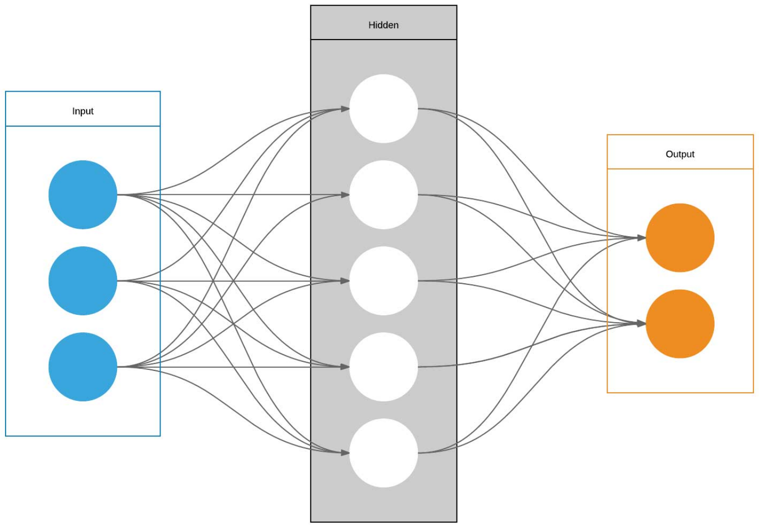

Figure 1: Illustration of a basic neural network with one hidden layer
This image has been created by LearnDataSci (www.learndatasci.com) and is licensed under CC BY-SA 4.0

We have a input layer and an output layer. We want the network to learn the parameters of a function that will be able to map each input to a certain desired output. The layers inbetween must therefore learn this mapping. Each circle in this network is called a neuron. Each neuron takes as input all of the values of the previous layer neurons that are feeding into it and applies the following linear function where x_i is the value contained in the i:th input neuron and w_i is the weight between the i:th input neuron and the neuron in question while b is the bias term.

$$\sum_{i=1}^{m} w_i \cdot x_i + b \tag{1}$$

It is difficult to approximate any function simply using linear tranformations of this kind and therefore a non-linear "activation function" is usually applied to the result of the previous equation. Any function could be chosen but choosing a continuous and differentiable function is preferable since this is the prerequisite for the network learning. The most common activation functions for hidden layers are currently ReLU (Rectified Linear Unit) and Leaky ReLU, shown in the equations below and illustrated in figure 2.

$$f(x) = max(0, x) \tag{2}$$

$$f(x) = max(\alpha x, x) \qquad (0 \leq \alpha < 1)$$

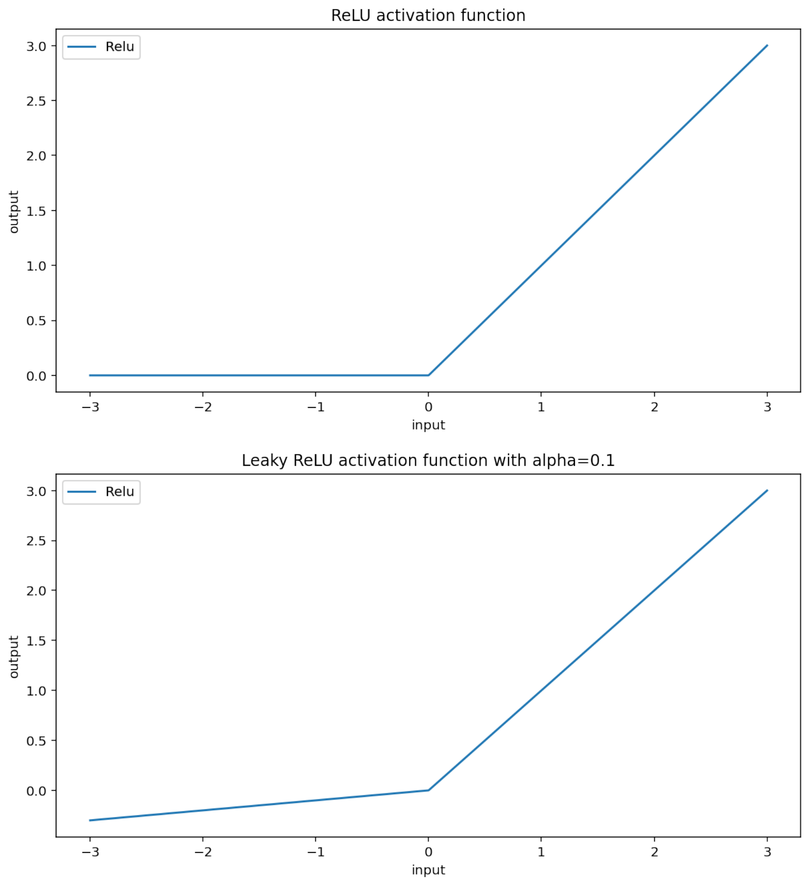

Figure 2: ReLU activation function and Leaky ReLU activation function.

Combining (1) and (2) gives us the following output that we would have from a hidden layer in a neural network with ReLU activation function.

$$max(0, \sum_{i=1}^{m} w_i \cdot x_i + b) \tag{3}$$

According to the universal approximation theorem (Hornik, Stinchcombe, and White 1989) a neural network of arbitrary length or arbitrary depth with the right kind of activation function (such as ReLU, TanH or Sigmoid) can approximate any continuous Borel measurable function. This means that the only problem left is to find these optimal parameters (or weights) that will lead our network to learn an approximation of the desired function. The current way to doing this is the following:

1. Initialize the network parameters with values sampled from a distribution of choice.

2. Decide on a loss function to use. (An example for the regression case is to take the MSE)

3. Run a input through the network and calculate the loss for this input. If the loss function is MSE then this would be:

   $$Loss = MSE = (\hat{y} - y)^2$$

   where y is the real value and $\hat{y}$ is the value predicted by the Neural Network.

4. Calculate the gradient which is the multi-variable derivative of the loss function in respect to the network parameters. This is known as backpropagation. This gradient gives us a vector that indicates where the loss increases fastest. Since we want to decrease our loss we will move in the opposite direction to adjust our parameters (weights and biases). In simple terms we need to calculate the adjustments to the parameters (at the same time) that we need to make in order to minimize the loss as much as possible.

5. Update the weights and biases through the following expression, where W is the weight vector, $\Delta C(W)$ is the gradient vector and α is the learning rate:

   $$W - (\Delta C(W)) * \alpha.$$

   The learning rate α is a value most often between 0.0 and 1.0 ensuring that we don't move too far in the gradient direction at once, which can lead the model converging to a suboptimal solution.

6. Repeat 3-5 until this has been done for all data. This is usually done in randomly selected batches in order to reduce training bias. It can then be repeated for several epochs, where one epoch is a whole training round where the model is trained on the entire dataset. In order to know when to stop training, we can evaluate the loss on test data that the model has not been trained on. If this loss stops decreasing or starts increasing, training should be stopped. This is only possible in cases where an objective evaluation of model performance is possible, so in the GAN case it cannot be done. I have instead evaluated the model myself at a certain number of epochs during the training phase and have stopped training when I deemed it to having stopped improving.

2.2 CNN

A Convolutional Neural Network or CNN is a class of neural network that is most commonly used for classification of images. The CNN is once again inspired by how biological brains are connected in the visual cortex in order to process visual information that is entering the brain. The CNN still consists of an input and output layer and the hidden layers inbetween. The hidden layers in this case differ by not being fully connected: local connectivity is used instead meaning that each neuron is connected to a local subset of the input image.

The filter values are learned automatically in the same way that regular weights and biases are learnt for a hidden layer.

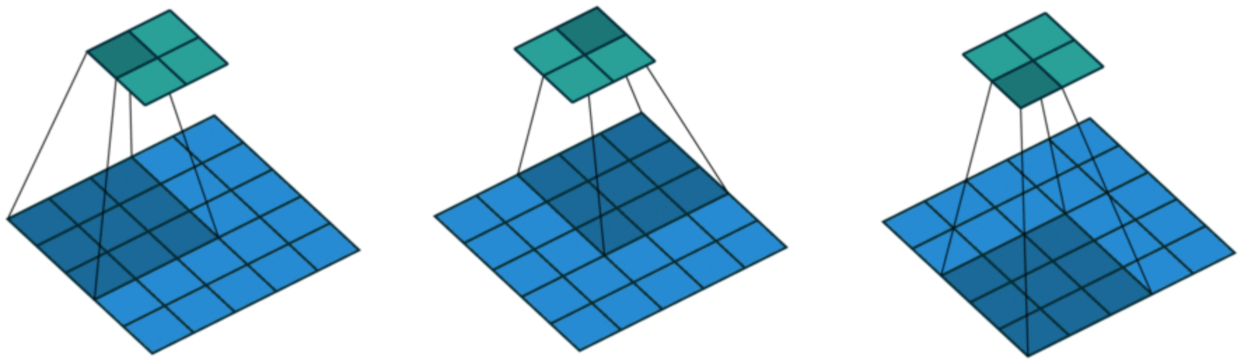

Figure 3: Illustration of a single 3x3 filter in a Two-Dimensional Convolutional Neural Network. The blue squares are the input image while the cyan one is the output.Stride is equal to two.
Dumoulin and Visin 2016

2.2.1 Dilated Convolutions

I have used dilated convolutions (illustrated in Figure 5), that simply mean that the connections are offset with a certain timestep in order to expand the receptive field. Expanding the receptive field is important for capturing long-term temporal dependencies in the data, but can be very costly in processing power, so using dilations is a great solution. As put by Yu and Koltun (2016) *dilated convolutions support exponential expansion of the receptive field without loss of resolution or coverage.* In some of my experiments I have also used dilated causal convolutions, a technique proposed by Oord et al. (2016). The causal convolutions have no clear theoretical reason to be applied in GANs of this type so this should not be seen as a focus. Causal convolutions are used in autoregressive models in order to ensure that only past timesteps can influence the future value. The idea is is illustrated in Figure 6.

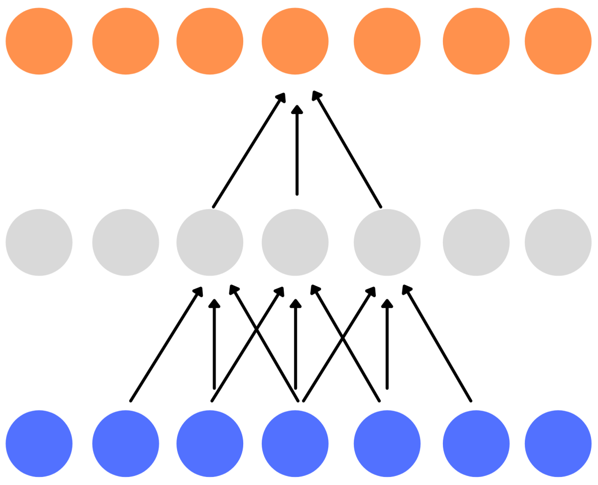

Figure 4: Standard one-dimensional convolution.The blue circles represent a small part of the entire input while the orange circles represent the outputs.

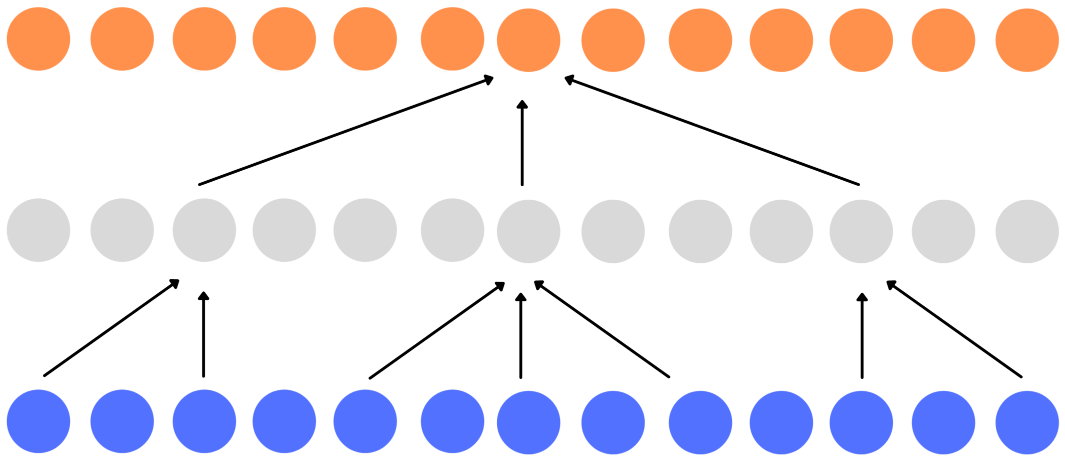

Figure 5: Convolutions but with dilation in order to increase the receptive field while saving processing power, thus increasing speed. Dilations are increased exponentially: dilation is 2 in the fist layer and 4 in the second.

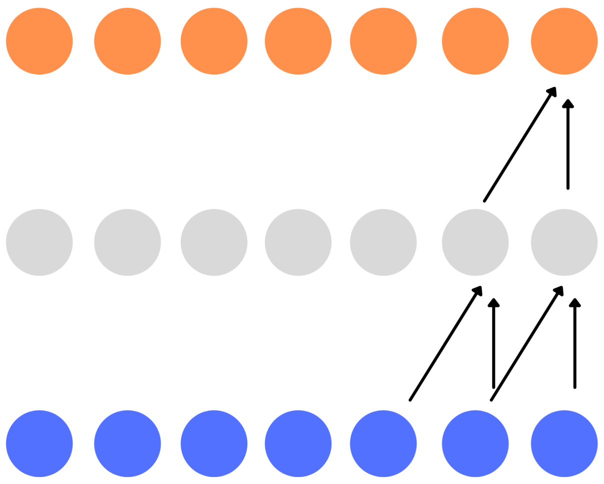

Figure 6: Causal convolution where future timesteps only depend on previous inputs

2.3 Generative Adversarial Network

Generative Adversarial Networks or GANs are a type of Generative Neural Network proposed by Goodfellow, Pouget-Abadie, et al. (2014). Since then, together with Variational Autoencoders, they have become one of the most popular generative models (Karpathy et al. 2016).

In essence, all generative models are simply an attempt to learn a real statistical distribution from which our data originate from, which can be denoted by P_x. We use our method to find an approximation to the real distribution, denoted as P_θ, where $\boldsymbol{\theta}$ are the parameters of this learnt distribution (Irpan 2017). Then, in order to generate new data, we simply take samples from this distribution. A GAN is a neural network, itself composed by two neural networks. One of the networks is the Generator and the other is the Discriminator. It can be seen as a two player game where the generator is trying to fool the discriminator into thinking that the generated data is from P_x while the discriminator is trying to find the best way to differentiate between the data coming from these two distinct distributions.

A common metaphor used is the one of the art forger (generator) trying to create art in the style of a certain artist in order to trick the art critic (discriminator). As the critic gets better as spotting the fakes, the forger also gets better since it learns from its mistakes and its successes until, ideally, it will create art that the discriminator cannot distinguish from the originals. In practice, traditional DCGAN networks have many problems converging to this ideal state, and instead tend to be known for problems such as Mode Collapse, Vanishing Gradients and a subsequent failure to converge (Arjovsky, Chintala, and Bottou 2017). These problems and techniques to adress them will be discussed later in the paper.

The structure of a GAN is illustrated in Figure 7 below.

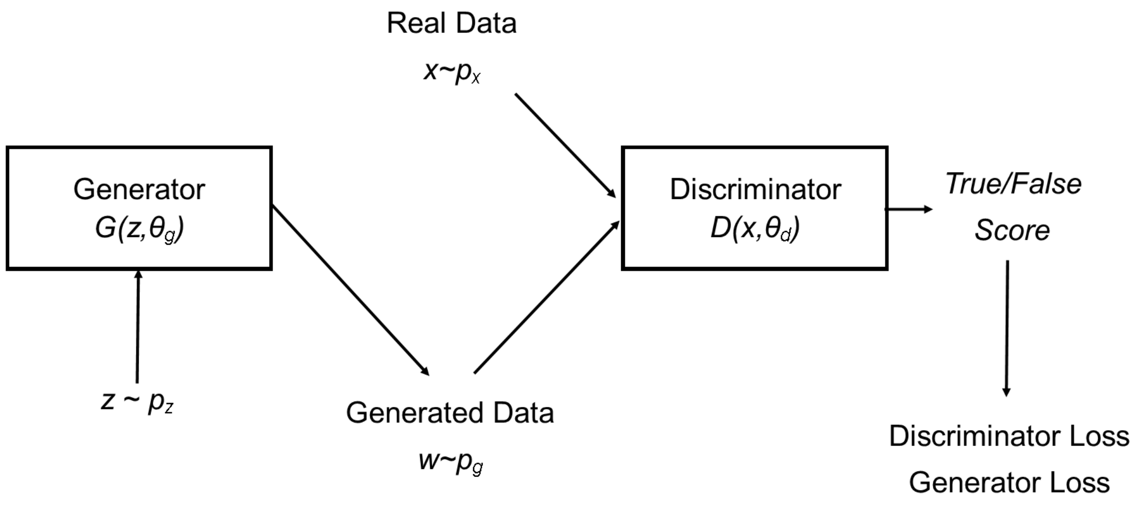

Figure 7: Illustration of the structure of a GAN.

2.3.1 Discriminator

The discriminator is the network that teaches the generator, through its calculation of the loss and by becoming better at classifying inputs, about the real distribution of the data. Depending on the type of architecture, the discriminator can give a score that is bounded between 0 and 1 indicating the probability of the input being real or it can instead give a score that has no lower and upper bound. I decided to use a score bounded between 0 and 1.

2.3.2 Generator

The generator, as shown in Figure 7, takes a random input z and upscales and tranforms this with convolutions in order to reach the desired size for the audio. The parameters in the convolutional layers will then be used to try to approximate the p_x during training so that each z will be a seed that produces a different sounding audio file.

The original WaveGAN paper (Donahue, McAuley, and Puckette 2019) uses transpose convolutions in order to upsample the generated white noise (z) into data with same length as the original data. These transpose convolutions are known in images to produce checkerboard artifacts, something that is not present in the original data and that therefore could be exploited by the classifer in order to spot the generated images easily. These checkerboard artifacts will also be present in the audio data. They solve this problem with a technique called Phase Shuffle. I won't do this since, while it solves the problem with the discriminator spotting the generated images, it still leads to the generator creating these artifacts. I solve this in the way proposed by Odena, Dumoulin, and Olah (2016) instead: upscale the images and then use convolutions. Their solution is for 2d images but based on my results it works perfectly even in the case of 1-d time series data. This gives the same output size as with the transpose convolutions but does not generate artifacts.

2.3.3 GAN Training

Both the generator and the discriminator learn the parameters needed to model the probability distribution of the data in the way described previously in section 2.1, but since the loss of the generator depends on the discriminator output, the backpropagation must start from the discriminator and then continue backpropagating in the generator in order to calculate the gradient, but only the generator parameters are updated in this stage (GoogleDevelopers 2019). The discriminator is updated at a different stage with its own loss that is only backpropagated through itself. The generator learns through unsupervised learning since the generator can never directly observe the real data.

2.3.4 Loss Functions

Loss functions used depend on the architecture. Deep Convolutional GANs or DCGANs are known for faster training but are prone to more of the problems discussed in the next section. Wasserstein Generative Adversarial Networks or WGANs with Gradient Penalty, are the architecture with most stable training at the moment. I have myself used a simpler DCGAN since writing the code for the Wasserstein GAN was outside of the scope of this paper. DCGANs are also known for producing higher quality samples and for faster training(Arjovsky, Chintala, and Bottou 2017), when problems do not arise. This was also positive for me due to the lack of processing power and time.

In our case the chosen loss function used is binary cross-entropy. The equation for calculating the loss for one single example can be seen below where y is the label of the image that is being classified (generated=0 or real=1) and p is the output of the discriminator and is also the probability , evaluated by the discriminator, of the input being real.

$$L(p, y) = -(y \log(p) + (1-y)\log(1-p)) \tag{4}$$

Both the discriminator and the generator share this loss, only that the generator will solely recieve loss for its own generated examples while the discriminator also has to calculate and use loss for all examples, both real or generated, that it is presented.

2.3.5 Known Problems

The most well known problems when training GANs are Mode Collapse ,Vanishing Gradients and Failure to Converge. All the information below is from GoogleDevelopers (2019).

- Danger of Mode collapse or just memorization. There is little risk of the generator copying the original data, since it is never shown directly to the generator, but it is very common that mode collapse will happen, leading the generator to find a data point (or a small collection of these) that will always fool the generator, leading all generated data to sound incredibly similar. This happens when the discriminator gets stuck in a local minima while training.

- Vanishing Gradients : It is possible that the generator will stop learning in the early stages of training when the generator is still outputting random noise; this is **due to the generated data distribution and the real data distribution initially having little or no overlap making the job for the discriminator very easy and leading to gradients that are close to zero** (Weng 2019). It has been suggested that this can be solved with Wasserstein loss, which is discussed later in this paper.

- Failure to converge : Attempts to solve this problem are using normalization techniques such as adding noise to the inputs of the discriminator and by penalizing discriminator weights.

3 Data Processing

3.1 Different Sample Rates

When faced with the problem of storing an analog signal such as sound into a digital memory such as the one of a computer, we are forced to create a discrete representation of the continuous sound. In order to do this, we take a certain amount of samples per second and record the amplitude at these sample points. If enough samples per second are taken, the human ear will not be able to hear the difference between the digital approximation and the original signal. I therefore choose not to resample the audio to 16khz since this would lead to all sound above the Nyquist frequency (In this case 8kHz) to be affected by a distortion known as aliasing (Zawistowski and Shah 2005). Since the human range of hearing is from 20Hz to 20kHz (Purves et al. 2001), I choose a sample rate of 44.1kHz, which will ensure that the data is free of distortion for the entire range of human hearing. Some of the files in the dataset had different sample rates, but due to this being a very small percentage (less than

1%), I decided to simply remove them. This is a difference in my approach to the problem when compared to WaveGAN, where they take data with 16kHz as sample rate. Due to the anti-aliasing filter removing the distortion, all information above 8kHz is lost and means that most sounds will sound less "bright", having lost a lot of high frequency energy. The sharp nature of high frequencies is often a desirable characteristic in snare drum sounds and especially in hihats, two of the drums sounds my model will be creating.

3.2 Stereo to Mono

In the original dataset, most of the audio files are in stereo format, meaning that there are two different waves, one for the left channel and one for the right channel. This is done so that they can differ between each other and create the illusion of a multi-directional sound when reproduced by a system with two speakers. This is complicated to process for me and would basically more than double the time needed to train and create the files so my first step is to take the average of both channels in order to have one single, simple wave signal. In essence, I go from a stereo signal to a mono signal.

3.3 Data normalization

For all of the original data, the amplitude (the strength of signal at each sample point) when recorded, was represented in different formats as shown in the table below. I decided to convert all of my data to assume 32-bit floating-point values between -1 and +1 so as to have everything in the same scale.

WAV format	Min	Max
32-bit floating-point	-1.0	+1.0
32-bit integer PCM	-2147483648	+2147483647
24-bit integer PCM	-2147483648	+2147483392
16-bit integer PCM	-32768	+32767
8-bit integer PCM	0	255

3.4 Length of signal

In order for me to be able to input my original data into the discriminator network, it was necessary that both the original data and the output from the generative networks had the same length - sample size. A shorter length would reduce training time but would also mean losing a large part of the training data. Another problem I had was that some of the files had a large gap of silence at the beginning and the end so I decided to eliminate the beginning silence and the end silence of each file, which also helps in aligning the files so that the peak amplitudes for each sound file align better with each other. An example of the effects of this data processing are represented in Figure 8 below.

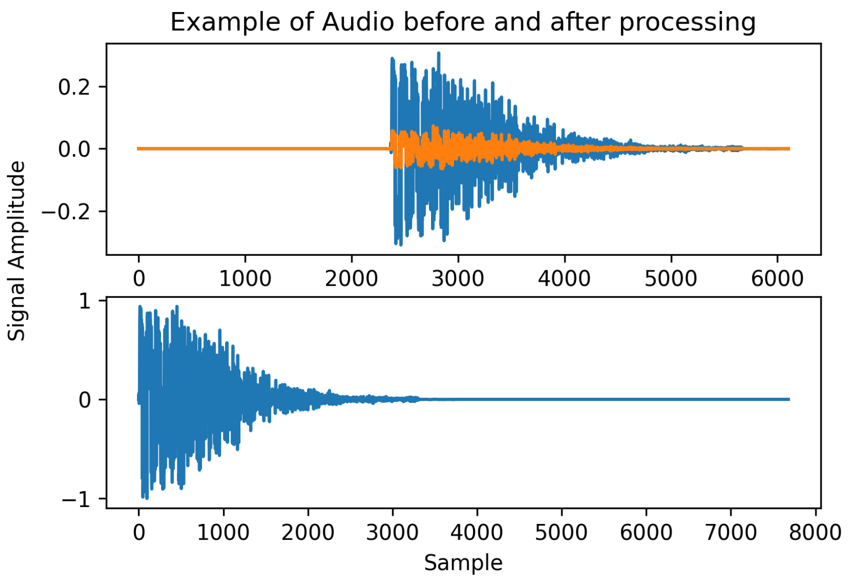

Figure 8: The effects of conversion from stereo to mono going from two audio channels, one for left(blue) and one for right(orange), to a single mono channel (shown in blue) can be clearly seen. The plot also shows the effect of data normalization with the scale changing while the waveform remains the same, which should make learning easier for the model. It also shows the effect of removing the silence from the beginning and end of the file in order to create files of equal length and that all begin at the same point.

4 Architecture and Training

The architecture is similar to the one for WaveGAN. The specific neural network structure of the final model can be seen in the appendix. I decide to compare some different models by using a DCGAN with and without causal colvolutions and also with and without dilations. Batch normalization, a technique that normalizes the values of the inputs to a neural layer based on batches of training data, is used for each upscale/convolution block in the generator. Batch normalization transforms the input of a layer of neurons so that the resulting normalized activation has zero mean and unit variance (equation 5,6,7). This activation is then scaled and shifted by two parameters as shown in (equation 8), the two of which are also learnt by the network during the training process. In these equations x_i is the original input to the layer and $\hat{y}_i^{(k)}$ is the final, batch normalized input. Batch normalization is used to avoid the problem with internal covariate shift (Ioffe and Szegedy 2015) and in order to avoid vanishing gradient problem by smoothing the optimization landcape (Santurkar et al. 2018).

$$\mu_b = \frac{1}{m}\sum_{i}^{m} x_i \qquad \text{(5) compute mean of input } x_i \text{ for batch of size m}$$

$$\sigma_b^2 = \frac{1}{m}\sum_{i}^{m} (x_i - \mu_b)^2 \qquad \text{(6) compute variance of input } x_i \text{ for batch of size m}$$

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_b^{(k)}}{\sqrt{\sigma_b^{(k)^2} + \epsilon}} \qquad \text{(7) normalize } x_i \text{ so that it is } \mathcal{N}(0,1) \text{ in respect to batches of data}$$

$$\hat{y}_i^{(k)} = \gamma^{(k)}\hat{x}_i^{(k)} + \beta^{(k)} \qquad \text{(8) linear transform with learned parameters } \gamma \text{ and } \beta$$

Dropout is used instead of batch normalization after each convolution block in the discriminator (see appendix). Dropout is used to avoid overfitting, and works by randomly dropping neurons out of the net during each training phase (Srivastava et al. 2014). Every neuron has a probability p of being dropped out at each training phase. This probability p is set to 0.3 in our discriminator, meaning that each neuron has a 30% chance of being disconnected. Dropout is illustrated in figure 9.

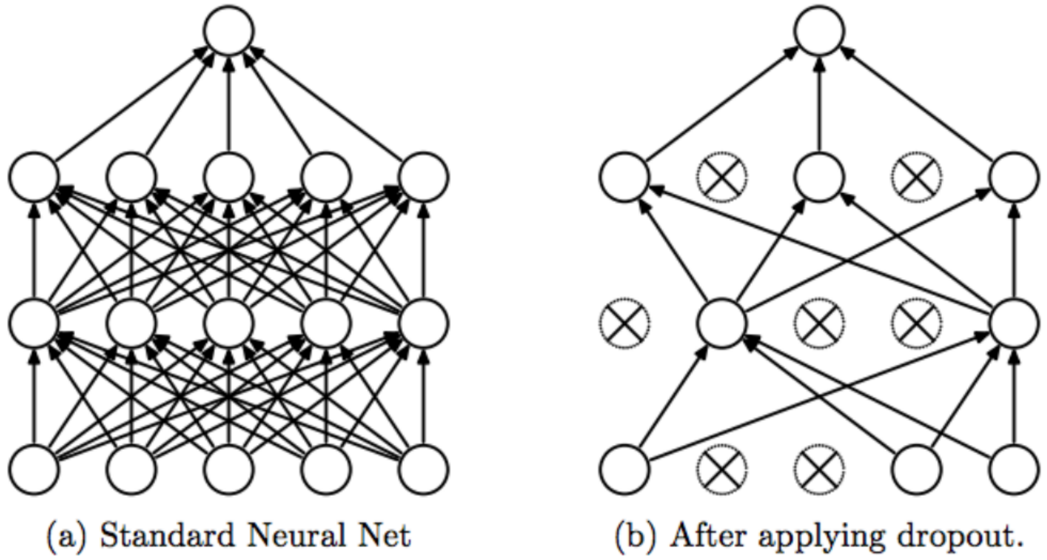

Figure 9: Graphical representation of a network before and after applying dropout. Figure from Srivastava et al. 2014

ReLU is used as an activation function for the generator and LeakyReLU is used for the discriminator. Stride is 4 for all 1D convolutional layers and Batch Normalization is always performed prior to LeakyReLU layers due to recommendations about this from previous research (Chen et al. 2019). All LeakyReLU layers have α set to 0.2.

I train the GANs on the drum data, with each drum type having around 800-1000 training examples. Each network is trained on one drum type at a time, since I want the generator to learn a representation of that specific data type. Training all the different drums at once will both require more time and also lead to the creation of hybrids between different sounds that may be interesting but that may also lack in quality. This is something that could be done in a future study.

I have also performed the same test to the one done by Donahue, McAuley, and Puckette (2019) , using the google Speech Commands Dataset in order to see if my model can correctly synthesize human speech. Due to limitations in time I haven't used the entire dataset , only using recordings of the word yes. This Yes-GAN model had to be adapted, with a different depth in order to generate and accept input of length 16000 instead of 7500. The purpose of this test is making the results of the model easier to judge, since humans are very good at judging the quality of generated speech.

5 Results

The generated sounds are difficult to evaluate objectively. It is my subjective opinion that many of the drum sounds are of excellent quality, altough having a weakness with mode collapse in a certain amount of the runs. In order to verify the viability of using these sounds I arranged a couple of short tracks using some of the produced drum sounds. These are available, along with several examples of generated drum sounds, by clicking *here*. The different DCGAN models were difficult to compare since each run can give completely different results due to the problems with GAN and especially DCGAN training stability. This made it very difficult to evaluate the effect of changes in model and the conclusion is simply that it is difficult to establish whether the addition of causal and-or dilated convolutions in the generator and-or the discriminator have a positive effect. What we are able to state is that a GAN model with both causal and-or dilated convolutions is able to perform very well in generating this kind of data if training doesn't get stuck.

Another important result is that the type of data that needs to be generated, has a large influence on the results. While it seems more difficult to generate speech (although i didn't have enough time for training as Donahue, McAuley, and Puckette (2019), which converged within four days whilst i stopped training after 12 hours), short drum stabs seem to be easy for the model to produce, with almost every training instance giving good results and requiring very little training time to achieve this. One of the reasons for this difference in training time may be due to the difference in dimensions, comparatively $\mathbb{R}^{7500}$ and $\mathbb{R}^{16000}$. But this would not explain why training on certain drum sounds always gives training stability while others do not. This may be due to the fact that certain sounds, such as a snare drum, are mostly composed by broadband noise (ModeAudio 2014). Since the model without training creates noise, due to the weights being initialized at random, this might mean that the starting distribution, when the weights are initialized, is much closer to the real distribution, making training easier and more stable even when we don't use Wasserstein GANs. In fact, in all of the different models, both the generation of snares and hihat sound always went well and gave great results, while producing other kinds of sounds like kick drums has not given equally consistent results.

A bizarre result which occured was when the generator created snare sounds when it was trained on only hihat data. This may either be due to the fact that they share the same data generating distribution, or a more likely explanation would simply be because some of the data been incorrectly labeled. A superficial search within the data to find an example of incorrectly labeled data was performed but no such example was found.

In order to make sure that the algorithm wasn't simply learning or memorizing certain data points, I identified the most similar sounds from the dataset using the lowest MSE as criterium. The results cannot be presented here due to it being audio, but performing this test with a small number of randomly selected generated files showed that the generated sounds are clearly quite different from the most similar data in the training dataset which seems to show that the GAN is able to generate novel audio, although the result may also simply indicate that MSE is a bad evaluation metric for similarity in audio.

Result of human voice synthesis by my Yes-GAN is that the algorithm is not equally quick in converging and giving convincing results. The results are worse, but still comparable to, those achieved by Donahue et al. (2019) when the algorithm is trained for around 800 epochs with a batch size of 5. Some of my results are similar, compared to what has been observed previously in GAN generation of animal images, where GANs create animals with several faces (Goodfellow 2017). In my case, the algorithm has often generated audio where yes is said more than once.

6 Discussion and Conclusions

My results show that it is possible to create high-quality drum samples with architectures such as DCGAN and that using causal and dilated convolutions is a viable approach, but not making it clear if it is significantly better. It also shows that 1-d transpose convolutions can be substituted with nearest neighbour upsampling followed by regular 1-d convolutions in GANs that generate one-dimensional data.

The obtained results of certain sounds being easier to generate, possibly due to them sharing similarities with the output of the untrained generator (in this case hihat sounds being similar to white noise), means that it might be interesting to investigate whether initializing generator weights in a strategic way might make training easier for the generator and therefore stabilize the training process.

References

Arjovsky, Martin, Soumith Chintala, and Léon Bottou (2017). *Wasserstein GAN.* `arXiv: 1701.07875 (stat.ML)`.

Chen, Guangyong, Pengfei Chen, Yujun Shi, Chang-Yu Hsieh, Benben Liao, and Shengyu Zhang (2019). *Rethinking the Usage of Batch Normalization and Dropout in the Training of Deep Neural Networks.* `arXiv: 1905.05928 (cs.LG)`.

Dhariwal, Prafulla, Heewoo Jun, Christine Payne, Jong Wook Kim, Alec Radford, and Ilya Sutskever (2020). *Jukebox: A Generative Model for Music.* `arXiv: 2005.00341 (eess.AS)`.

Donahue, Chris, Julian McAuley, and Miller Puckette (2019). *Adversarial Audio Synthesis.* `arXiv: 1802.04208 (cs.SD)`.

Dumoulin, Vincent and Francesco Visin (Mar. 2016). “A guide to convolution arithmetic for deep learning”. In: *ArXiv e-prints.* eprint: `1603.07285`.

Goodfellow, Ian (2017). *NIPS 2016 Tutorial: Generative Adversarial Networks.* `arXiv: 1701.00160 (cs.LG)`.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning.* `http://www.deeplearningbook.org`. MIT Press.

Goodfellow, Ian J., Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio (2014). *Generative Adversarial Networks.* `arXiv: 1406.2661 (stat.ML)`.

GoogleDevelopers (Oct. 2019). *Generative Adversarial Networks.* URL: `https://developers.google.com/machine-learning/gan/`.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White (1989). “Multilayer feedforward networks are universal approximators”. In: *Neural Networks* 2.5, pp. 359–366. ISSN: 0893-6080. DOI: `https://doi.org/10.1016/0893-6080(89)90020-8`.

Ioffe, Sergey and Christian Szegedy (2015). *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. `arXiv: 1502.03167 (cs.LG)`.

Irpan, Alex (Feb. 2017). *Read-through: Wasserstein GAN*. URL: `https://www.alexirpan.com/2017/02/22/wasserstein-gan.html`.

Karpathy, Andrej, Pieter Abbeel, Greg Brockman, Peter Chen, Vicki Cheung, Rocky Duan, Ian Goodfellow, Durk Kingma, Jonathan Ho, Rein Houthooft, Tim Salimans, John Schulman, Ilya Sutskever, and Wojciech Zaremba (June 2016). *Generative Models*. URL: `https://openai.com/blog/generative-models/`.

ModeAudio (Sept. 2014). *Massive Drum Design, Part 2: Snares: ModeAudio Magazine*. URL: `https://modeaudio.com/magazine/massive-drum-design-part-2-snares`.

Odena, Augustus, Vincent Dumoulin, and Chris Olah (2016). “Deconvolution and Checkerboard Artifacts”. In: *Distill*. DOI: `10.23915/distill.00003`. URL: `http://distill.pub/2016/deconv-checkerboard`.

Oord, Aaron van den, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu (2016). *WaveNet: A Generative Model for Raw Audio*. `arXiv: 1609.03499 (cs.SD)`.

Purves, Dale, George J Augustine, David Fitzpatrick, Lawrence C Katz, Anthony-Samuel LaMantia, James O McNamara, and S Mark Williams (2001). *Neuroscience. 2nd edition*. Sinauer Associates.

Santurkar, Shibani, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry (2018). “How Does Batch Normalization Help Optimization?” In: *Advances in Neural Information Processing Systems*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Vol. 31. Curran Associates, Inc. URL: `https://proceedings.neurips.cc/paper/2018/file/905056c1ac1dad141560467e0a99e1cf-Paper.pdf`.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov (2014). “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15.56, pp. 1929–1958. URL: `http://jmlr.org/papers/v15/srivastava14a.html`.

Velardo, Valerio (Apr. 2019). *Generating music in realtime with Artificial Intelligence: What if music could change automatically with the emotional state in a video game? Valerio Velardo*. URL: `https://idalab.de/seminar/generating-music-in-realtime-with-artificial-intelligence-what-if-music-could-change-automatically-with-the-emotional-state-in-a-video-game/`.

Weng, Lilian (2019). *From GAN to WGAN*. `arXiv: 1904.08994 (cs.LG)`.

Yu, Fisher and Vladlen Koltun (2016). *Multi-Scale Context Aggregation by Dilated Convolutions*. `arXiv: 1511.07122 (cs.CV)`.

Zawistowski, Thomas and Paras Shah (Mar. 2005). URL: `http://www2.egr.uh.edu/~glover/applets/Sampling/Sampling.html`.

7 Appendix

Table 1: Final Generator Network Architecture

Layer (type)	Kernel Size	Output Shape
z ∼Normal(0,1)		(n, 120)
Dense	120*256	(n, 30720)
Reshape + Batch Normalization + ReLU		(n, 120, 256)
Upsampling1D (16x)		(n, 120*16, 256)
Conv1D + Batch Normalization + ReLU	(25,128)	(n, 474, 128)
Upsampling1D		(n, 3792, 128)
Conv1D + Batch Normalization + ReLU	(25,64)	(n, 942, 64)
Upsampling1D		(n, 7536, 64)
Conv1D + Batch Normalization + ReLU	(25,32)	(n, 1878, 32)
Upsampling1D		(n, 15024, 32)
Conv1D + Batch Normalization + ReLU	(25,16)	(n, 3750, 16)
Upsampling1D		(n, 30000,16)
Conv1D + TanH	(25,1)	(n,7500,1)

Table 2: Final Discriminator Network Architecture

Layer (type)	Kernel Size	Output Shape
Input x or G(z)		(n, 7500)
Conv1D + LeakyReLU + Dropout	(25,128)	(n, 1869, 128)
Conv1D + LeakyReLU + Dropout	(25,64)	(n,1821,64)
Conv1D + LeakyReLU + Dropout	(25,64)	(None, 1725, 64)
Conv1D + LeakyReLU + Dropout	(25,64)	1533
Flatten		(None, 98112)
Dense + Sigmoid	(98112,1)	1

Adam optimizer, the as of date most popular optimization algorithm for gradient descent, is used with learning rate being set to 1e-4, while all other optimizer parameters are set to the default value in keras.