BeatGAN: Developing a Generative Adversarial Network for Rhythm Generation

Author: Thom Hoogerheijde (t.k.a.hoogerheijde@students.uu.nl)
Supervisor: Dr. Tomas Klos (t.b.klos@uu.nl)
Second Reader: Emre Erdoğan (e.erdogan1@uu.nl)

Bachelor Artificial Intelligence, Utrecht University
7.5 ECTS

July 2, 2021

Abstract

We introduce BeatGAN, a generative model for rhythmic patterns. A GAN is a generative model, used in image generation. BeatGAN was trained on an image dataset consisting of a visual representation of the Groove [12] data set, which contains drum recordings. We are interested in applying image generation techniques to music generation, which leads to the following research question: How suitable are generative adversarial networks for generating rhythmic patterns represented in a visual format? We created a custom data set on which we trained BeatGAN. BeatGAN uses convolutional neural networks to generate images, making it a DCGAN [10]. Adjustments to the model and the data set were made during training, with the aim to achieve better GAN performance. Results show that the visual output of BeatGAN is not comparable to the samples in the data set. Converting the visual representation of the rhythmic patterns to audio, we conclude that the audio is also not comparable to the samples in the data set. We identify multiple problems with our method and the training of BeatGAN, and provide suggestions for future research to improve BeatGAN. The results of our research provide evidence that a visual representation of music in combination with a DCGAN is not suitable to generate novel rhythmic patterns.

Contents

1 Introduction

In nature, we can identify many examples of periodicity. The movement of the planets within our solar system can be described as periodic. Their orbits affect our perception of the rising and setting of the sun, which in turn is a periodic recurrence. We as humans seem to want to identify these periodic patterns [1]. In the art of music, this periodicity is essential. We identify periodicity as a pulse or a beat. In music, we amplify the importance of some of the beats, and deamplify the importance of others. This accentuation of different beats causes rhythm [2].
We have seen that within the field of artificial intelligence, much research has been done relating to image generation [3][4][5]. Deep generative modeling techniques have produced images of people that do not exist in the world. There have also been examples of deep learning models that are able to generate art. Within the field of music generation, much research has been done to generate songs that were not created by humans [6][7][8]. Image generation and music generation are sub-fields of generative modeling. The goal of generative modeling is to generate novel pieces of data, that are comparable to already existing data. Generative modeling techniques often provide an architecture which can be adjusted such that they are able to process different types of data. In this research, we will be adapting a generative modeling technique to generate music, and more specifically, rhythm.
A promising technique within the field of generative modeling is the generative adversarial network (GAN), first introduced by Goodfellow et al. [9]. A GAN is an architecture in which two neural networks are trained in parallel, using the performance of each other to get better at their respective tasks. One of the neural networks is a generative network, which can be used to produce new samples. The GAN architecture will be explained in more detail in chapter 2. GANs have been shown to produce convincing results when applied to image generation. However, within the field of music generation, research has been somewhat lacking [7]. Mogren performed research with a GAN architecture, using classical piano music as a data set to train the model [11]. In this study, Mogren aimed to generate melodies and harmony similar to the classical piano music the model was shown. There was no emphasis placed on rhythmic elements. The output data of the model reflects the aim of the study. A lack of rhythm can be identified in the output.
The relevance of this research is the adaptation of the GAN architecture to rhythm generation, a topic that has not seen as much research as melody and harmony within the field of music generation. GANs have produced convincing results in image generation. We will be creating an image representation of rhythm, and we will convert samples of a rhythm data set to this image representation. We will use these image representations as a data set to train the networks in the GAN architecture. We then aim to generate new music by converting generated images from the GAN back to a musical representation of the generated rhythms.
We are curious to see if GANs can be applied to a visual representation of

rhythm. Are GANs able to discover the relations in rhythm that are important in our experience of rhythm? Can the rhythmic relations be learned when utilizing a visual representation of rhythm? We have formulated our main research question as follows: How suitable are generative adversarial networks for generating rhythmic patterns represented in a visual format?
There are some sub-questions we will encounter. Because we are using a technique from the field of image generation, we need an image representation of rhythm. The first sub-question is: how can rhythm be represented in a visual format? Second, what types of neural networks should be selected within the GAN structure? Third, how does the output of the GAN compare to man-made rhythms?
The application of a GAN specifically to generate rhythmic patterns could give insight into computers' ability to interpret rhythm, when represented as a visual format. Rhythm is characterized by the selective accentuation of certain beats. We aim to give insight into the usability of GANs for identifying and reproducing these characterizations.
Finally, we will now give an overview of the chapters in this research. In chapter 2, we will first describe what neural networks are, and how they can be used inside the GAN architecture to generate images. We will then give a high-level description of the implementation of our GAN in chapter 3. Chapter 4 will describe the process of selecting the data set, and processing the data set to create a visual representation of music which we used in the GAN architecture. Chapter 5 consists of the results produced by our model. Chapter 6 contains an analysis of the results, a review of our method and suggestions for further research. Chapter 7 concludes our research by answering the research questions.

2 Generative Adversarial Network

The GAN architecture is a generative modelling technique. GANs are able to generate new data, whenever it is shown many examples of what the programmer is trying to generate. We will introduce the GAN architecture by first giving a general overview of neural networks. The GAN requires the use of two neural networks that are trained in parallel. One neural network is an image classifier and one neural network is an image generator. The high-level flow of a GAN is as follows: the image generator generates images and the image classifier is trying to classify generated and real images as generated and as real. The goal of the GAN is to create an image generator that generates images that are indistinguishable from real images, according to the image classifier. We will also describe the specific type of neural network used in our model, the convolutional neural network (CNN). Finally, we give an overview of the GAN architecture and describe how the CNN fits in the GAN architecture.

2.1 Neural Networks

One part of the GAN architecture is the use of an image classifier. In this section, we describe what a neural network is, and how it can be implemented as an image classifier.
A neural network is a mathematical function, taking numerical inputs and producing numerical output. Neural networks consists of layers. Every layer consists of a number of neurons, which store numbers. A neural network has an input layer, an output layer and intermediate layers which are called hidden layers, as shown in figure 1.
The input of a neural network consists of an array of numbers equal in size to the number of input neurons. When classifying images, the number of neurons in the input layer is equal to the number of pixels in the image. The neurons of the input layer are connected to the neurons of the next layer, whose neurons are in turn connected to the following layer's neurons. The connection between neurons is called a weight, and is represented as a variable numerical value. We often initialize all the weights as random numbers at first. If we feed the network numerical input, we can calculate the value of all neurons in the next layer. For every neuron n_x in layer x, its value is defined as the sum of all weighted values of n_x's input neurons N_{x-1}. We also add a randomly initialized numerical variable to the value of n_x, called a bias. Bias is used to make a model more flexible when handling data it has not seen before. The number of layers and neurons is specified by the programmer. The final layer of the network is the output layer. In an image classifier, the number of neurons in the output layer is often equal to the number of classes of images we would like the network to classify. Every class is represented by an output node in the output layer. We scale the output layer neuron values between 0 and 1. Their values can then represent probabilities that some input belongs to a certain class.
In an image classifier, the output node with the highest probability is taken as the network's prediction on what class the image belongs to. It is not likely that an untrained network is able to assign the correct class to some data input, when all weights and biases are initialized randomly. To achieve better performance, we show the network many images, along with the class the image belongs to. We compare the network's output to the real label of the sample and adjust the weights and biases, aiming to improve the model's performance, such that the network is able to make better predictions. This process is called training. We use a data set to train a neural network. Our data set is labeled, which means that the data set contains some label value for every sample. This label tells us what class the sample belongs to.
When comparing the real labels of samples to the network's predictions of the same samples, we can measure the network's performance. Measuring performance is done using a loss function. We use the network's predictions on samples and the real labels of the same samples as input for the loss function. If the network performs well, this loss function will output a small value. If the network performs poorly, it will output a large value. The loss function gives us a measure of how poorly the network performs when classifying data. In neural

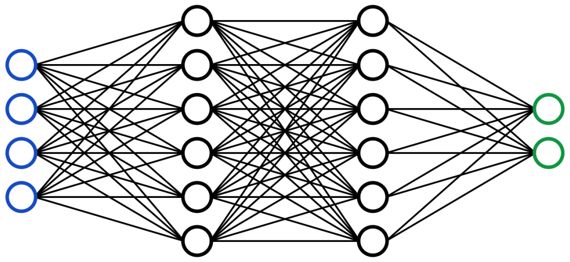

Figure 1: Example of a neural network with one input layer (4 nodes), two hidden layers (6 nodes) and one output layer (2 nodes). Connections between neurons are represented as lines. In neural network implementations, these lines represent a certain weight value.

network training, we are trying to minimize this loss function, which can be done by utilizing an algorithm called gradient descent.
The output of the loss function is a numerical value. Since the weights and biases are used to produce the network's predictions, we can tweak the weights and biases to search for the gradient along which the output of the loss function decreases the most. The gradient is computed using an algorithm called backpropagation. We descend the gradient, thereby adjusting the weights and biases of the network. We repeat gradient descent until a minimum is reached for the loss function. If the loss does not decrease any more, we have found a minimum. At this point, the model's performance is optimized for all training samples in the data set. Note that this minimum is not necessarily a global minimum. Overall model performance of the network will then be analyzed by calculating the model's performance on unseen data. A neural network is trying to approximate a data distribution. A data set almost never contains all images that exist belonging to a certain class. For example, a data set containing images of cats can always be expanded. A data set is therefore a subset of the full data distribution. However, as long as the data set has enough variation, we can approximate the full data distribution. A well trained network on a data set that is representative of the full data distribution is likely to perform well on data from the full data distribution, even though the network had never seen that data before.

2.2 Convolutional Neural Networks

Convolutional neural networks (CNN) are a subset of neural networks. The type of image classifier we used for our GAN is a CNN. Groupings of pixels often contain important information about the image, which is taken into account when classifying images with CNNs. The description of a neural network and its training procedure as described in the previous section still applies to a CNN. However, a CNN has some additional mechanisms, which improve perfor-

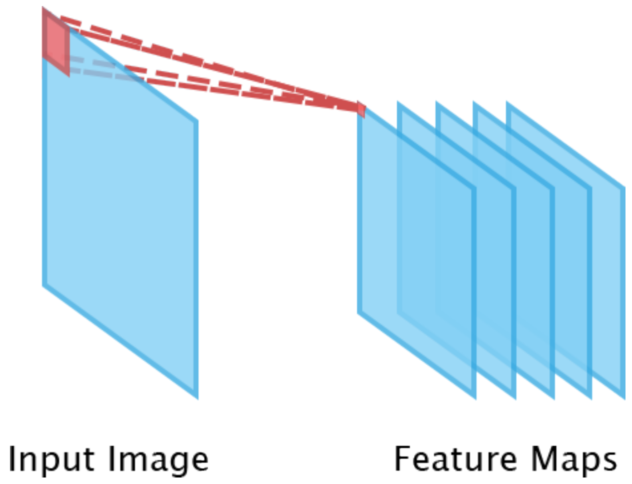

Figure 2: High-level example of a convolution from an input image to a feature map. A window of pixels is selected, which is mapped to a single pixel in the feature map, using a filter.

mance when compared to a densely connected network, when used in the GAN architecture [19].

In a CNN, the weights are not represented as single values per neuron connection. Neuron connections are weighted as a collective, instead of weighting them individually. The use of collective weighting causes CNNs to be useful in image classification, where pixel context is important [22].

The connections between each layer are defined by a convolution operation. To apply the convolution we randomly initialize a filter, which is a matrix of values. This filter is used to perform the convolution. Applying a convolution results in a set of nodes called a feature map. The feature map is a downsampled representation of the image, a result of the convolution. A high-level overview of a convolution in a CNN is shown in figure 2. Figure 3 provides a detailed explanation of a two-dimensional convolution, where the two dimensions represent the width and height of an image.

In CNNs, we initialize multiple different filters per layer. When applying the convolution with a single filter, we get a feature map, which is two dimensional. Applying the convolution for all filters, we get a three-dimensional convolutional layer, where the z-axis corresponds to the different feature maps created by the different filters. At first, these filters are initialized randomly. But during training, the filters get adjusted, such that better feature maps are learned. To connect a convolutional layer to a new convolutional layer, we apply a convolution to the three-dimensional layer, with a three-dimensional filter instead of a two-dimensional filter. This results in a single two-dimensional feature map. Using multiple filters results in a new convolutional layer in three-dimensions. The convolutional layers are the hidden layers of our CNN.

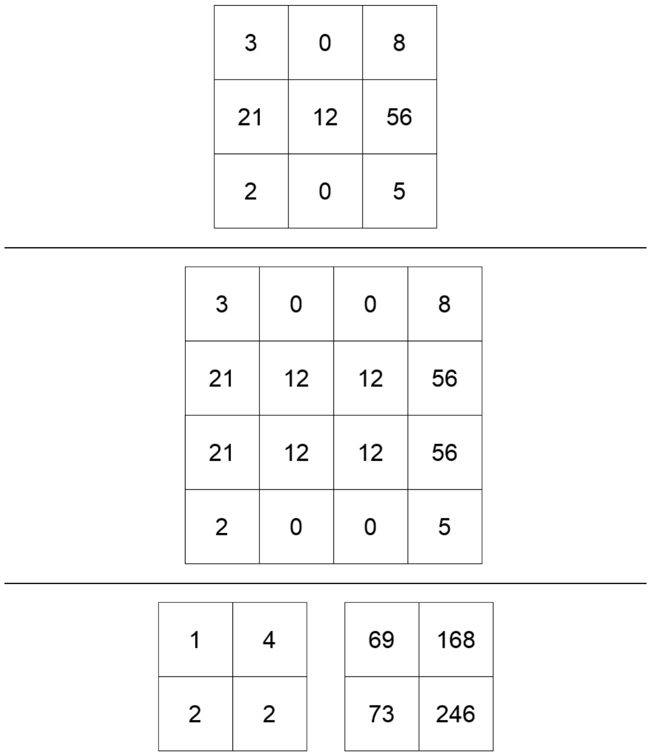

Figure 3: Moving from top to bottom, this figure describes a two-dimensional convolution in a CNN. First row represents a 3×3 pixel image. The second row corresponds to all sub-matrices of the image selected when using a filter size of 2×2. The third row contains a randomly initialized filter (left), and the resulting feature map (right). Feature map calculation using the filter and the top-left sub-matrix in the middle row: $(1 \times 3) + (4 \times 0) + (2 \times 21) + (2 \times 12) = 69$.

The last convolutional layer gets connected to an output layer. Since the convolutional layer is three-dimensional, we first flatten the layer. Flattening a three-dimensional layer creates a one-dimensional array, resulting in a regular neural network layer consisting of an array of all nodes from the three-dimensional layer. Every single node in the flattened layer is connected to every single node in the output layer. The value of a node in the output layer corresponds to a probability that an input image belongs to the class represented by the node.
The CNN is given no information on what filters need to be learned. This lies at the core of the CNN architecture. The network is itself is responsible for learning useful features from processing the data set using gradient descent.

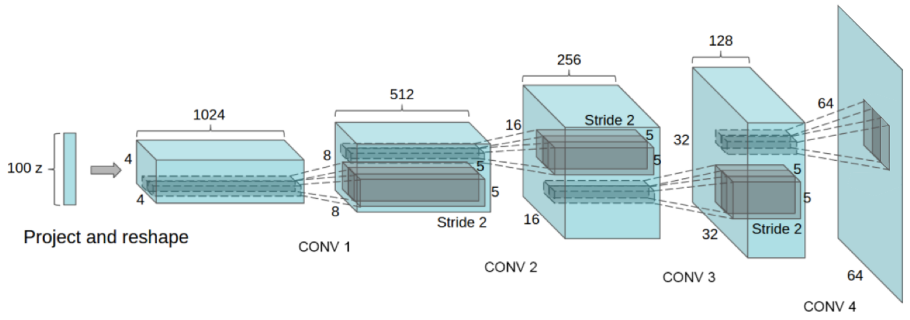

Figure 4: Example of a transposed convolutional neural network. Network generates random noise images as input and applies transposed convolutions to upsample the random noise. Filters are learned which generate feature maps. Multiple feature maps form a three-dimensional transposed convolutional layer. The network consists of 4 transposed convolutional layers in total. Output layer is a two-dimensional layer, consisting of a single feature map. Original image is taken from Radford et al. [10], adjusted to a two-dimensional output.

2.2.1 Transposed convolution for Generative Networks

CNNs can not only be implemented as classifiers, they can also be used as generative networks. Instead of using the convolution, which downsamples an image, we use the transposed convolution, which can upsample an image. The CNN architecture still applies, so the weights to be learned are still filters, matrices of trainable values.
Figure 4 provides an example of a transposed convolutional neural network. The transposed convolution requires specifying a stride and a filter size. The stride is a tuple of two numbers, when creating an image generator. The stride tells us how many pixel steps we need to take in the feature map, to match one pixel step in the original image. The filter size can also enlarge the resulting feature map, this happens when two filters overlap when the filters are either too big, or the stride is too small. A transposed convolution in two dimensions is shown in figure 5. The transposed convolution is applied for all filters in the transposed convolutional layer, resulting in many different feature maps. Similar to the standard convolutional layer, using more than one filter results in a three-dimensional layer. Filters used in three-dimensional transposed convolutional layers are also three-dimensional. The result of the transposed convolution for a single filter in three-dimensions is a two-dimensional feature map. It is common practice to use many filters in the early layers of the network, and slowly decrease the number of filters in the later layers, until a final layer can be added with a single filter, such that a single feature map is produced, which is the image output of the generative network.

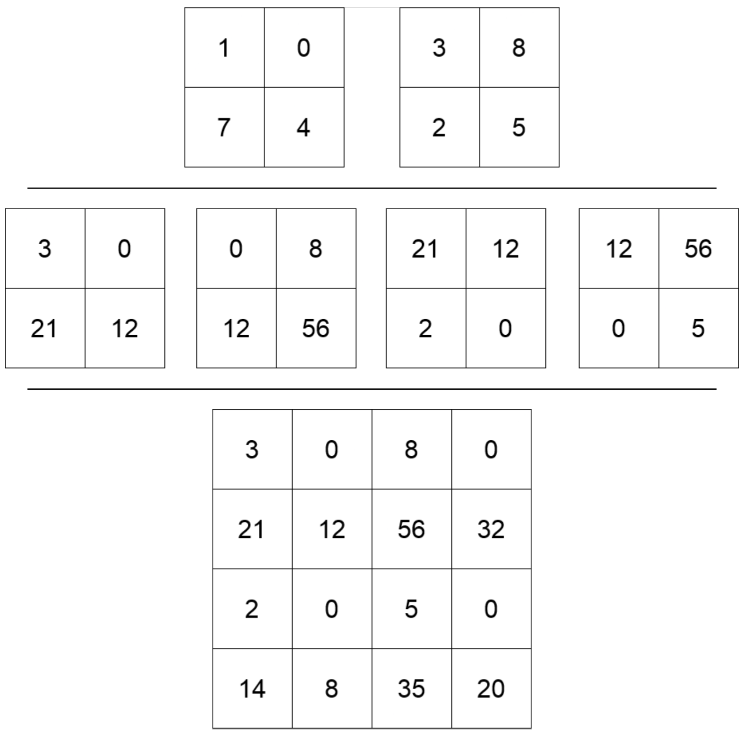

Figure 5: Moving from top to bottom, this figure describes a two-dimensional transposed convolution in a CNN. The first rows represent a pixel image (left), and a filter (right). Every value in the filter is used as a scalar for the pixel image. Results after scalar multiplication are shown in the middle row. Stride is set to (2,2), resulting in the feature map in the bottom row.

2.3 GAN Overview

In the paper Generative Adversarial Nets [9], the authors propose a framework for training two neural networks in parallel. One neural network is an image generator, the second neural network is an image classifier. In this research, we use a convolutional neural network for the classifier, and a transposed convolutional neural network for the image generator. These days, many of the GANs are similar to the deep convolutional type (DCGAN) [19][10]. The term deep refers to the fact that the CNNs used in the GAN contain more than one hidden layer. We follow this approach with our implementation.
We will introduce the GAN architecture with an analogy [9]. Suppose that two parties exist, a group of criminals, and a group of policemen. The criminals are trying to produce fake currency. The policemen are trying to distinguish fake currency from real currency, to prevent the criminals from using fake currency. The criminals and the policemen are competing with each other, which causes

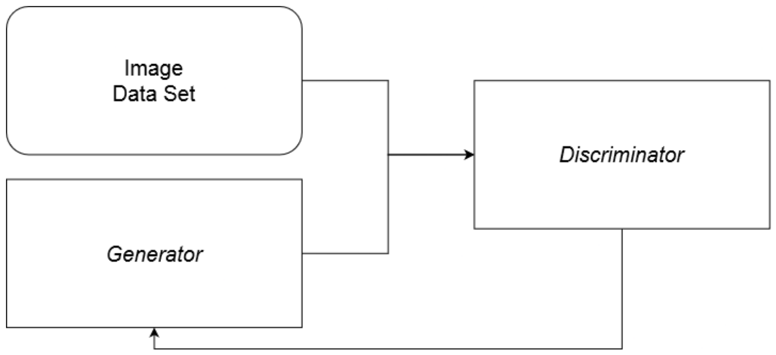

Figure 6: High-level overview of the GAN structure, describing the program flow between the data set, discriminator and generator

both groups to improve their methods to gain the upper hand. In the GAN architecture, the criminals are analogous to the image generator, and the policemen are analogous to the image classifier.
Following GAN terminology [9], we define the generative neural network as the generator (G), and the image classifier as the discriminator (D). Figure 6 provides a high-level overview of the architecture. G is responsible for generating images from random noise. D is responsible for assigning real labels to the images from the data set and the generated samples from G. D tries to classify images belonging to the data set as real, and images generated by G as fake. The output of D is the prediction $D(z)$. $D(z)$ is interpreted as the probability that the sample z is a sample from the data set. $D(z)$ is compared to the true label, after which D adjusts its weights using some gradient, such that it improves in classifying samples. G moves using a gradient such that it improves in generating samples that are more likely to be wrongly classified by D. The gradient for G is calculated using the classifications D made on G's generated samples. The training process of D and G can be defined as a minmax game, where D maximizes, and G minimizes the probability that D assigns the correct label to the samples it processes.
A GAN converges whenever $D(z) = 0.5$ [9]. At convergence, D cannot distinguish real samples from fake samples, and therefore classifies all samples as true and fake equally as often. Goodfellow et al. have proven that theoretically, GANs always converge to $D(z) = 0.5$ [9]. After convergence, we can conclude that G is successful in generating images that fool D.

3 GAN Implementation

This chapter first describes the implementations of D and G in more detail. We then describe the high-level training loop used to train D and G. Finally, we describe some of the heuristics we used when training D and G. Training

GANs is a difficult task [29][30][31][32], as it requires training and balancing two neural networks in parallel. Trying to obtain meaningful results, we often adjusted our model architectures and tweaked the step size used in gradient descent. Because of this trial-and-error approach, this chapter will only describe the main elements and heuristics that were used in the models. For the specific model implementations, we refer to the GitHub repository containing all code.[1]

3.1 Generator

G is a CNN with transposed convolutional layers. Following Goodfellow et al. [9], we first generate an input vector of 100 random values from a normal distribution. The input vector is connected to a dense neural network layer. The dense neural network is reshaped to a three-dimensional transposed convolutional layer. The layers following this first transposed convolutional layer are also transposed convolutional layers, whose dimensions depend on the preceding layer's dimensions and stride specifications. The output layer of G is a transposed convolutional layer with a single filter, producing a feature map whose dimensions are equal to the desired image dimensions. Design of the network is similar to the example in figure 4.
G uses the transposed convolutional layers to upsample random noise to an image of the desired dimensions. In the early layers of the network, many filters are used, resulting in many feature maps. After some set number of transposed convolutional layers, the desired image size can be reached. At that point, *G* has generated an image from random noise. We then apply the tanh activation function to normalize the image's pixel values between -1 and 1, following Goodfellow [19]. We used LeakyReLU as the activation function for the hidden layers of *G*. An activation function adjusts the output space of a node. LeakyReLU weighs negative values lesser than positive values.

3.2 Discriminator

D is a CNN with convolutional layers. All input is first rescaled between -1 and 1. The network consists of several convolutional layers after the input layer. The last convolutional layer is densely connected to the output layer. The output layer consists of a single node, which corresponds to the probability that an input sample is real. We used LeakyReLU as the activation function for the hidden layers of *D*.

3.3 Training Loop

In chapter 2, we described how neural networks are able to learn. Whenever a network has produced an output, we can compare its output to the real output we would want the network to give. We calculate the network's performance using a loss function, which outputs a value resembling how poorly the network

[1] `https://github.com/thomhoog/BA_Thesis-BeatGAN`

performs. The higher the loss function output, the poorer the network's performance. We can calculate a gradient using the weights and biases of the network, along which the loss function decreases the most. Calculating the gradient is done efficiently using backpropagation. We adjust the weights and biases such that we descend the gradient. During the training of the GAN, we repeat the process of adjusting the weights and biases multiple times.
Calculating the gradient and adjusting the weights and biases is a costly operation with respect to time. Applying standard gradient descent means that we process all examples in the data set before we make a decision on how to adjust the weights and biases, such that the loss decreases the most. Alternatives exist that are more time efficient, such as mini-batch gradient descent, also called stochastic gradient descent (SGD). In SGD, we shuffle the data and divide the data set in equally sized mini-batches. We then perform standard gradient descent on the mini-batches. Processing one mini-batch is called an iteration. When all mini-batches have been processed, we say that one epoch has passed. We can repeat training the networks for as many epochs as we specify. In this research, we use SGD.[2]
The output value of a neuron is first passed through an activation function. We can normalize the output values from the activation functions, such that the mean of the neuron's output distribution is equal to 0 and the standard deviation is equal to 1. Normalizing the outputs of neurons within a mini-batch is called mini-batch normalization.
Mini-batch normalization is used frequently in GANs and often improves GAN stability, even though the reasons for model improvement are somewhat unclear [23][24]. We used mini-batch normalization during training.
In the implementation of our GAN, one iteration of the training loop starts with calculating loss for D and G. Calculating the loss requires D to output predictions for all samples in the mini-batch, and all samples generated by G. The number of samples G generates is equal to the mini-batch size. D outputs predictions on all samples in the mini-batch and all samples generated by G. With D's output and the real labels for all samples, we are able to calculate the cross-entropy loss:

$$-\frac{1}{batch\_size}\sum_{i=0}^{batch\_size} P(z_i)\cdot log(D(z_i)) + (1-P(z_i))\cdot log(1-D(z_i))$$

where number of samples $= \{z_1, ..., z_i\}$ and $P(z)$ represents the real probability that a sample is real. Real samples are labelled as 1, generated samples are labelled as 0. Cross-entropy loss is calculated for both the real and the generated samples. The two calculated cross-entropy losses are then combined to find the total loss for D. The loss for G is obtained by calculating the cross-entropy loss between D's predictions on the generated samples and a set of labels all set to

[2]Goodfellow [19] suggested using the Adam optimizer, which is an extension of some variants of SGD. We initially followed this recommendation, but during training we were curious if using standard SGD would improve performance. We did not find any noticeable improvements when using SGD over Adam, or vice versa.

1. We compare to a set of 1's, because we want G to adjust its weights such that D is more likely to classify a generated sample as a 1, which causes G to improve at fooling D.
After calculating the losses for D and G, we can calculate two gradients using the Keras API optimizers. We ascend the gradient for G, and descend the gradient for D. Moving along the gradient completes one epoch.
Whenever the loss value converged, we stopped training the models. The output of G at the last completed epoch was taken as the output of the model.

3.4 Training The Model

During training, we generate 16 images from G after every epoch of training. These are saved for review. Loss per epoch was printed, to monitor for improvements in performance of both networks. If the loss did not decrease anymore, the network has found a minimum for the loss function. The loss for D and G is printed after every epoch. Both parts of D's loss function are printed separately.

3.5 Zero Padding

A side effect of the convolution is the reduction of image dimensions. Depending on the window size and image size, images lose some detail when applying a convolution. The loss of detail is caused by the fact that the window is compressing areas of images into a single value. Another side effect of the convolution is that pixels at the edges have less impact on the feature map values, since they are processed less often in the convolutions.
A solution to size reduction and information loss is zero padding [21]. Before the image is processed by D, we pad the border of the image with zeros. Zero-padding makes sure that the output feature map dimensions are the same as the input feature map dimensions. Zero-padding was used heuristically during training.

3.6 Dropout

We want to avoid that a model overfits the data. Dropout is a technique we can use to address the problem of overfitting the dataset. With a set probability, dropout is applied to every neuron of the layer. The probability is used to decide if a neuron is ignored, or if it may propagate its output. Some neurons are therefore purposely ignored, which significantly reduces overfitting of the data set [25]. Dropout was used heuristically during training.

4 Data Set

In this chapter, we describe the process for selecting and processing the data set such that it can be used in the GAN architecture. In the first section, we provide some music theory, that lays the foundation on which we built our image representation of the data set. The second section describes the selection of the data set. The third section will describe the general method for processing the data set. The fourth section will describe the conversion process from MIDI to CSV. The fifth section will describe the conversion process from CSV to PNG. The final section provides a summary of the total data set, after having applied all processing steps.

4.1 Introduction

The data set we used to train the GAN consists of MIDI files. MIDI files require some additional processing before they can be used in our GAN. We are able to exploit the structure music theory provides, such that we can convert MIDI to an image.

Drum Hits

Notes are the fundamental building block of music. In this research, we are trying to model rhythms produced by a drum kit. We define playing one element of the drum kit as a drum hit. Drum hits can be represented as notes in music theory. In practice, notes and drum hits have duration and pitch. In this research, we discard the pitches of specific drum hits, given that the MIDI data set does not contain pitch information. We also discard drum hit duration. The argumentation for discarding duration is given in section 4.3.

Groupings of Drum Hits

Rhythm is defined by context. Playing a single drum hit in isolation can be musical, but it is not music. Music occurs whenever we play multiple drum hits over time. A result of playing a steady drum hit pattern over time is the beat within the music.
In music theory, the beat of a song is typically defined in quarter notes. The time between beats is equal to the length of a quarter note. Note that this definition does not provide any information on how long a quarter note is in practice.
The real time between beats is provided by the beats per minute (BPM). BPM is a value which is set by the composer. BPM relates the length of a beat, and thus the length of a quarter note to time. Adjusting the BPM, adjusts the real length of a quarter note. The length of quarter notes is therefore variable, but the relation of a quarter note to BPM is always the same.

Measures

A grouping of some specific set of notes is called a measure. A related term to a measure is the time signature. A time signature defines how many notes are played in a measure [33]. The most common time signature in western music is common time, or $\frac{4}{4}$. The top of the fraction specifies that there are 4 notes in a measure. The bottom of the fraction specifies that these 4 notes are quarter

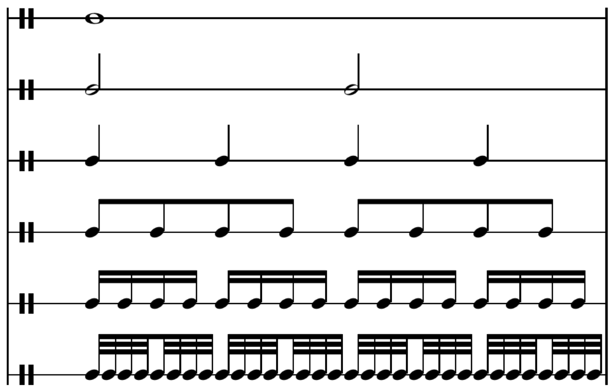

Figure 7: Comparison of measure subdivisions. The width of every row of notes is equal to a single measure, spanning the same amount of time. From top to bottom: whole note, 2 half notes, 4 quarter notes, 8 eighth notes, 16 sixteenth notes, 32 thirty-second notes.

notes. As a result, the measure cannot be longer than the total real time of 4 quarter notes. Common time therefore defines the length of a measure as 4 quarter notes per measure.

Subdividing Measures

As the name implies, a quarter note is a quarter of the length of a whole note. A whole note in common time spans the whole measure. Dividing the measure in quarter notes is called a subdivision. We are not restricted to quarter note subdivisions. We can divide in two, giving us half note subdivisions. We can also divide in sixteenth notes, which means that we play sixteen notes per measure, during the same time it takes us to play one whole note. Comparisons between subdivisions are given in figure 7. In this research, we will use the sixteen note subdivision as the smallest subdivision. Larger subdivisions do not allow for enough expressiveness in the rhythms. If we use smaller subdivisions, the amount of data to be saved doubles, increasing the file size. Sixteenth notes strike a good balance between expressiveness and file size.

4.2 Data Selection

This section describes the selection of the data set used as input for the model. The source of the data is a processed subset of the Groove data set, made public by Magenta [12]. This data set contains recordings of 10 drummers, who played drum beats on an electronic drum kit, which converted the performances to MIDI. Drummers performed with a metronome, which causes the data to be aligned with a particular BPM, set by the drummers themselves. Performances were mainly played in common time. We restrict the data set to only contain

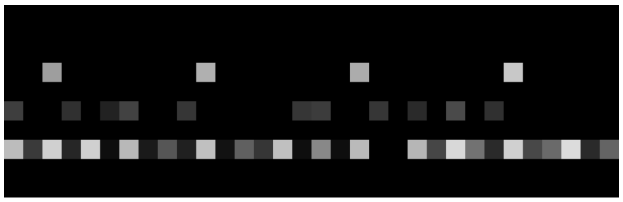

Figure 8: Example of a sample in the data set, pixels on the x-axis represent a sixteenth note. Different drum hits are represented on the y-axis. A non-black pixel represents an instrument being played at a sixteenth note.

music played in common time. Different time signatures impact the length and feel of a measure. Using a single time signature allows us to make sure that every drum beat is of equal length. Genres varied, with rock being the most common. We exclude Latin drum beats, being outliers in general rhythmic sensation. Drum beats that were shorter than two measures were also discarded, since these are shorter than the desired length. A complete list of selected genres is provided in section 4.6.

4.3 Processing Data - General Method

This section describes the general method we used to process the data set such that we can use it for generating music. We are trying to use image generation methods to generate music. We convert MIDI files to an image, which is then used as input for the GAN. We first converted all MIDI files to a CSV file, which allows for easier access to the information in the MIDI files. The CSV files consist of multiple rows with information. All rows describe events at some time. The events we extracted are the events that specify that a note is being played. The CSV files also contain information on when a note ends. Drum hits have a fixed duration in the MIDI format. Because of the fixed duration, information about the end of the note is not relevant. This information was therefore discarded. The CSV files contain headers with meta-information, relating to the playback speed of the MIDI file. The notes with their respective times, and the headers were saved and used to convert the CSV file to an image. This resulted in a data set of images, which we used to train the GAN. Figure 8 shows an example of a sample in the data set. Audio examples of the samples in the data set can be found in the GitHub repository.

4.4 Processing Data - MIDI to CSV

The MIDI files were converted to the CSV format using an open source script [15]. Using Python, the conversion was easily performed by calling the script on every MIDI file in the data set, which resulted in a CSV representation of all MIDI files in the data set.

4.5 Processing Data - CSV to PNG

This chapter describes the process of converting the CSV music representation to the image music representation. We also highlight some encountered problems during the conversion process.
We introduce three dimensions of music, which we will use in our image representation of the music. First, we have pitch, in our case, the different drum hits. Second, we have velocity, representing how loud the drum hit is being played. Third, we have time. Combining the three dimensions, we get a definition of music: different drum hits being played with a certain velocity, at a certain point in time. We pick an image format which is able to represent the three dimensions: greyscale images. The three dimensions in greyscale images are the x-axis, y-axis and the pixel values. We mapped time to the x-axis, the different drum hits to the y-axis and velocity to the pixel values. This specification relates pitch, velocity and time to images. The extracted information from the CSV files consists of the drum hits being played, at a time, with a velocity. With the defined dimensions, we are able to map the extracted information in the CSV file to the image.
The first encountered problem during the mapping process is the fact that time is continuous. A digital image is a two-dimensional array of values, and is not continuous. We are trying to map a continuous scale to a non-continuous format. Utilizing the structure of music allows us to relate time to the x-axis. Like Gillick et al. [12], we specify that one pixel represents a sixteenth note. The first pixel of a row in the image represents the first sixteenth note in a piece of music. The second pixel represents the second sixteenth note. The third pixel represents the third sixteenth note, et cetera.
The first line of the CSV file contains a header, with a field describing the number of MIDI clock pulses per quarter note ($PPQN$). This is the link between the time representation of a MIDI file and the time representation in music theory. We define the MIDI clock time for quarter note x as follows:

$$t_x = PPQN \times (x - 1)$$

If $t_1 = 0$, it follows that $t_2 = PPQN$. In the time it takes to play one quarter note, we can play four sixteenth notes. The MIDI time per sixteenth note is therefore equal to $\frac{PPQN}{4}$. This is the core of the mapping of MIDI time to the pixels of an image. If we set $PPQN$ to 480, it follows that all MIDI events that fall between the range of 0-119 MIDI pulses, should be played on the first sixteenth note. The process of mapping values within a certain range to a certain sixteenth note is called quantization [34].
Mapping different drum hits to the y-axis is done by selecting a row of pixels, which corresponds to a specific drum sound. In the MIDI format, the velocities range from 0 to 127. To map the velocities, we normalize them to the range 0-255, corresponding to the pixel range of a greyscale image.

Image Dimensions

To decrease the time spent on the training procedure, it is important that the images are not too large. The aim was to minimize the size of the images, such that training the model can be done as fast as possible. We initially chose the image dimensions as 32 pixels wide and 10 pixels high. The 10 pixels are able to represent every possible MIDI drum sound used in the data set. Every row of pixels in the image corresponds to a specific drum sound. In total, 22 drum sounds were available during recording. Gillick et al. [12] and Roberts et al. [13] simplified the 22 drum sounds to 9 classes. We added one extra class, resulting in one extra pixel. The class we selected to add was the side stick of the snare, which plays the role of a snare, but has a different sound. Adding an extra class aids in creating more stable transposed convolutional neural networks. A transposed convolution with strides (2,2), doubles the image size, which is useful since we are upscaling a random noise image in the GAN. We cannot multiply a number of pixels by 2 to achieve a pixel height of 9 pixels. This is possible with 10 pixels.
In the images, 32 pixels represent 32 sixteenth notes, or two measures of music in common time.
Whenever we convert a MIDI track to an image, we first convert the entire track to a single image. We then slice the image to sub-images of the desired dimensions.
During the implementation and experimentation phase of this research, we suspected that the 10×32 image dimensions might be a cause of instability in the training phase. We were therefore interested in the model's behavior when we changed the aspect ratio and size of the images. We created two more data sets, consisting of the same data, but with different image dimensions. We suspected that using square images would aid in training the models. Samples in the first additional data set have image dimensions of 32×32. We obtained these dimensions by padding the 10×32 data set with 11 rows of black pixels on the top and bottom of the sample. Samples in the second additional data set have image dimensions of 16×16. We obtained this data set using the same methods we used to obtain the 10×32 data set. We argued that using multiples of 2 would

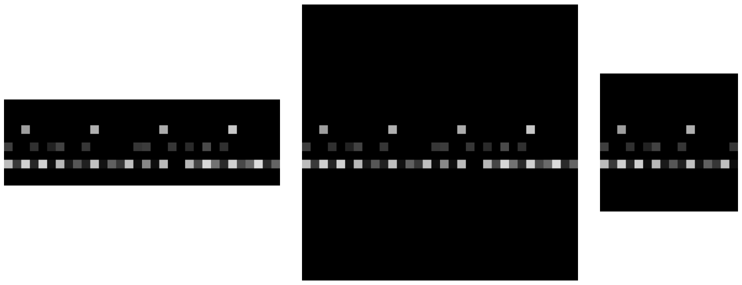

Figure 9: Comparison between samples with different image dimensions. Image dimensions from left to right: 10×32, 32×32 and 16×16.

allow for a more stable transposed CNN. Note that the length of a sample in the 16×16 data set spans one measure, instead of two measures for the other two data sets. This resulted in a larger data set. Examples of the additional data sets are shown in figure 9.
Implementing and training the GAN was split into three parts, because of the three different data sets. We denote the model that uses the 10×32 data set as BeatGAN 1. The model that uses the 32×32 data set is denoted as BeatGAN 2. The model that uses the 16×16 data set denotes BeatGAN 3. The training procedure was the same for all versions of BeatGAN, except that the models and learning parameters were adjusted, and a different version of the data set was used.

Beat Detection and Offset Correction

The recordings in the Groove data set are not perfect. Examples can be found where the drumming does not start when the track begins. Converting such a track to an image and slicing that image to smaller images of the desired dimensions without addressing these problems would result in drum beats that are offset in time by some slight amount, when compared to other samples in the data set. We need to make sure that every sample in the data set starts with a beat. We achieve this by making sure that a beat lies in the first column of the pixel images. In music one can identify a beat of a track, which can be seen as a grid with equal step size, laying on top of the track, this is called a beatgrid. Examples of how the beatgrid relates to a sample is shown in figure 10.
The first step in our offset correction process is detecting the beat of the track. The method we used here is inspired by known beat detection algorithms [16][17]. For every column in the image, the pixel values are summed, this gives us a velocity distribution of the MIDI track over time. To aid in the detection of peaks, bass drum velocity is emphasized when played under certain conditions. In pop music, the bass drum is often played on the first and third beat, in combination with a snare drum on the second and fourth beat [18]. The exact locations of the bass drum and snare drum are known in the array. Bass drum velocity is emphasized whenever it is followed by a snare drum on the next expected beat. Averaging over all velocities gives us a threshold, which is calculated as follows:

$$threshold = 0.95 \times avg(velocities)$$

For every column in the sample image, the sum of velocities is compared to the threshold, if it is larger than the threshold, the column is defined as a peak. The peaks are used to identify high energy points in the track, which can be indicators for places where a beat can lie.
Beats occur at every quarter note. For every peak, we created a beatgrid using quarter notes as step size, and count how many peaks lie on that grid. We selected the peak with the maximum count as the most likely beat to lie on the beatgrid. From this selected peak, we expanded the grid to the end and to the

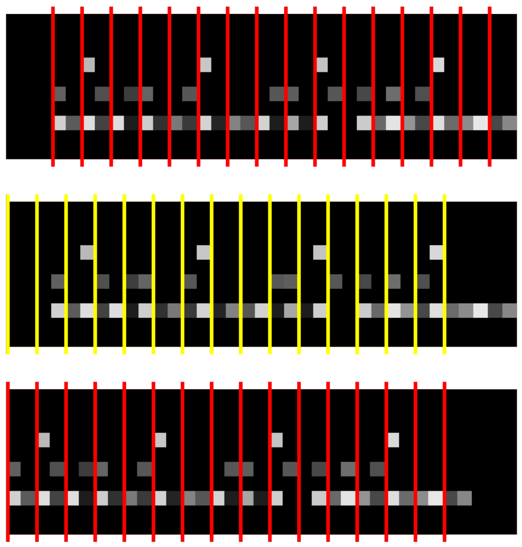

Figure 10: Visualisation of the offset correction process. Red bars in the top image represent the beatgrid without offset correction. The yellow bars in the middle image represent the desired beatgrid location. The bottom image represents the corrected beatgrid, note that the pixels have shifted to the left, indicating offset correction.

start of the track in steps of quarter notes, resulting in a beatgrid for the track. The beatgrid array is of equal length as the full track.

Downbeat Selection

The downbeat is the first beat of the measure. It is the beat which often has the strongest accent [18]. We shifted the track such that the downbeat lies on the first column of the pixel images, which was not always achieved using just the offset correction. Using the beatgrid array we calculated in the previous section, we checked every beat i in the grid, starting from the beginning of the grid. If it contained a snare drum, we returned $i - 4$, giving us a location of an expected bass drum. Otherwise, we checked if i contained a bass drum, which often lies on the first beat. If that was the case, we returned i.
It is possible that the index is negative, when a snare drum was selected. If that was the case, we extended the image such that the negative index becomes the first column of the image. A bass drum is then placed at that point.

Final Image Processing

After the process of downbeat selection, we sliced the image according to the desired dimensions of the data set, starting from index i defined above. These sub-images were saved, as samples for our data set. Any leftover data is discarded.

4.6 Final Data Set

For the 10×32 and 32×32 dimensions, the data set contains 4906 samples. The 16×16 data set contains 9952 samples. The additional samples obtained when preparing the 16×16 data set is a result of applying our MIDI processing method for smaller image dimensions. The data set consists of the following genres, ordered from most common to least common: rock, funk, jazz, afrobeat, afrocuban, hiphop, neworleans, soul, dance, reggae, pop, highlife, country, punk and blues. Table 1 gives an overview of the General MIDI (GM) sound mappings to the recorded pitch and array position. Drum performances were recorded on a drum kit with more elements than GM provides in their specification [14]. Sounds recorded at pitch 26 and pitch 22 do not exist in the GM specification and were mapped to an open hi-hat and closed hi-hat respectively.

Recorded sound	**Pitch**	**Array position**	**Class (pitch)**
tom 2	45	0	low-mid tom (47)
tom 2 (rim)	47	0	
tom 1	48	1	high tom (50)
tom 1 (rim)	50	1	
tom 3 (head)	43	2	high floor tom (43)
tom 3 (rim)	58	2	
snare (head)	38	3	snare (38)
snare (rim)	40	3	
snare x-stick	37	4	side stick (37)
kick	36	5	kick (36)
open hi-hat (bow)	46	6	open hi-hat (46)
open hi-hat (edge)	26	6	
closed hi-hat (bow)	42	7	closed hi-hat (42)
closed hi-hat (edge)	22	7	
pedal hi-hat	44	7	
ride (bow)	51	8	ride cymbal(51)
ride (edge)	59	8	
ride (bell)	53	8	
crash 1 (bow)	49	9	crash cymbal (49)
crash 1 (edge)	55	9	
crash 2 (bow)	57	9	
crash 2 (edge)	52	9	

Table 1: Overview of GM sound mappings to recorded pitches, array position and drum hit class.

5 Results

We will now give a brief overview of how the data set is used in the GAN. We used the final data sets described in the previous chapter to train the GAN. The data set is used in combination with the output from the generator as input for the discriminator. The discriminator assigns a probability that the samples it processes are real. We compare D's predictions to the real labels of the real and generated data. Calculating the cross-entropy loss, we are able to use backpropagation to calculate gradients for D and G. Using the gradients we adjust the weights and biases of D and G, such that D improves at classifying samples and G improves at generating samples that fool D. Whenever the loss converged to a certain value, we stopped the training procedure. When the loss converged, we saw that change in the output of G stagnated. At every epoch, we ask G to generate 16 images which are saved for review.
The image output of G was first converted to a CSV file, in the format specified by the MIDICSV script [15]. We then used the MIDICSV script to convert the CSV files to MIDI. Images were imported as arrays, and the rows of the images

were mapped to a drum hits specified in table 1.
Image results vary in size, because of the use of three different data sets. As a reminder, image dimensions consist of: (10×32) for BeatGAN 1, (32×32) for BeatGAN 2 and (16×16) for BeatGAN 3. We adjusted the parameters of BeatGAN 2, twice, resulting in two extra versions of BeatGAN 2: BeatGAN 2.1 and BeatGAN 2.2.
To better analyze the rhythmic sensations, we looped the generated MIDI pattern 4 times for the images of length 32, and 8 times for the patterns of length 16. Images were upscaled, results do not represent the actual image size. For every implementation of BeatGAN we first show all generated samples over all epochs. We then show a single generated sample per epoch.

5.1 Visual Output

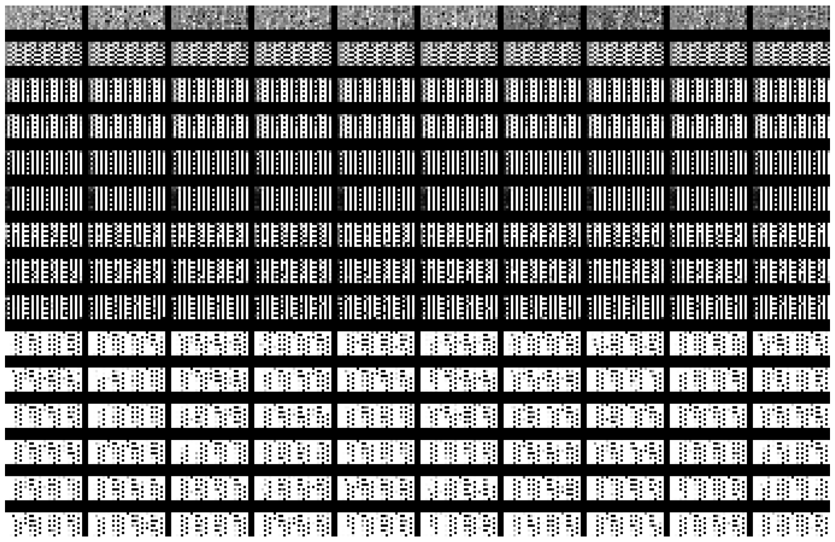

Figure 11: Total overview of output of G in BeatGAN 1 (10×32 pixels). Epochs are separated by horizontal black bars. Every row contains 16 generated samples from G. Epochs from top to bottom: 0, 1, 2, 3, 4, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50. Loss for G converged to 0.338, loss for D converged to 0.694.

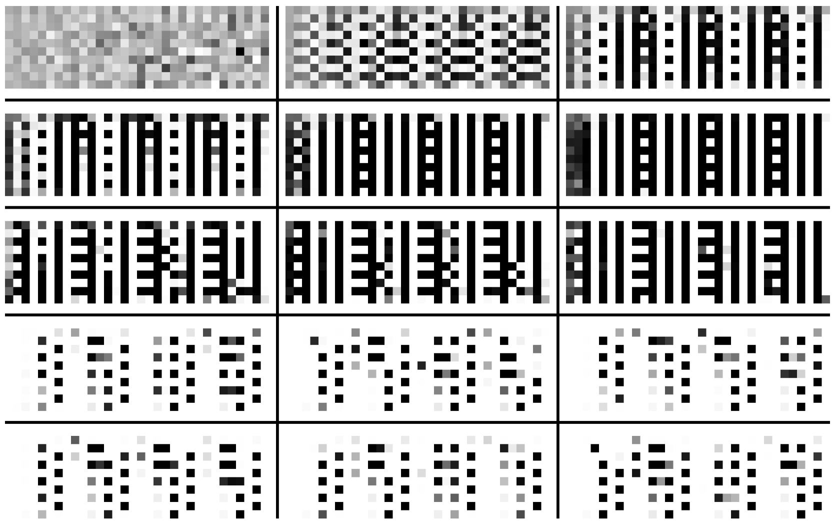

Figure 12: Figure is read from left to right and top to bottom. The figure shows the first column of figure 11.

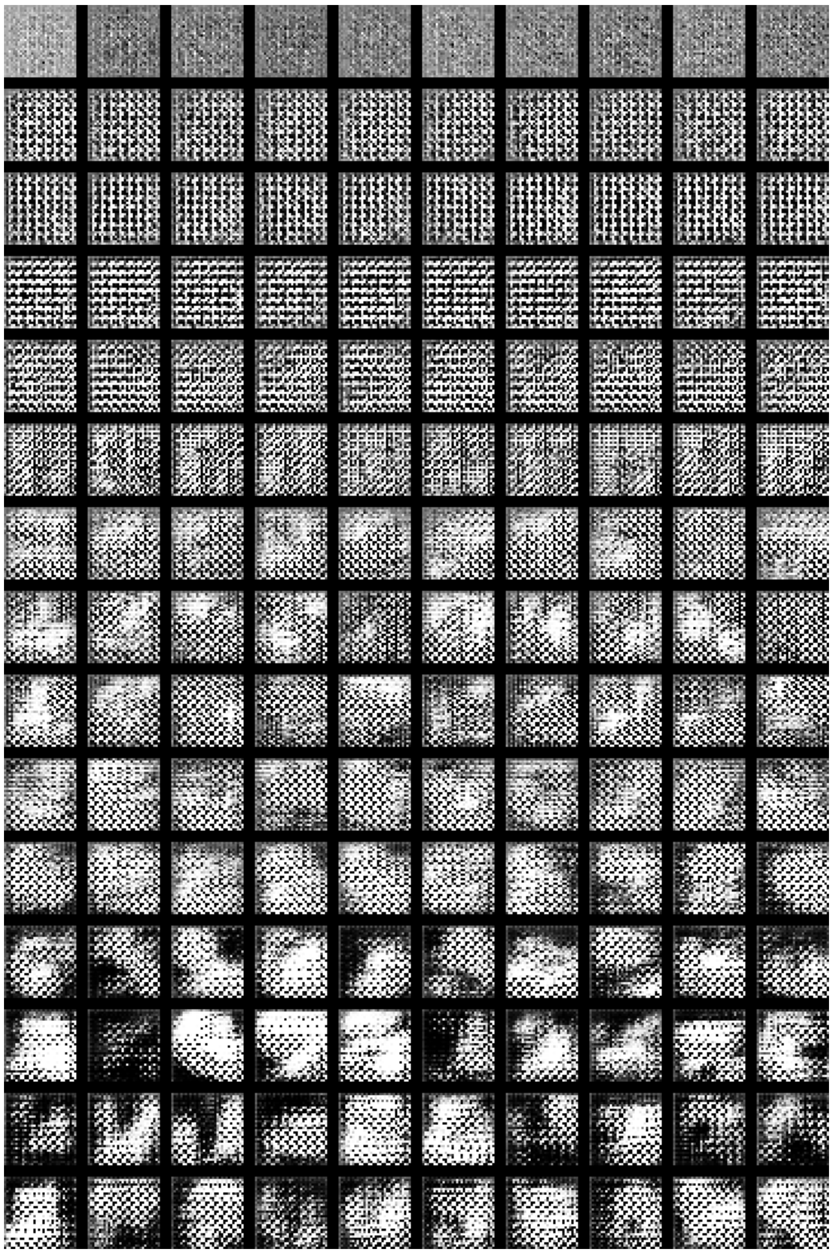

Figure 13: Total overview of output of G in BeatGAN 2.0 (32×32 pixels). Epochs are separated by horizontal black bars. Every row contains 16 generated samples from G. Epochs from top to bottom: 0, 5, 10, 20, 40, 60, 80, 100, 120, 140, 160, 180, 200, 220, 240. Loss for G converged to 3.001, loss for D converged to 6.029.

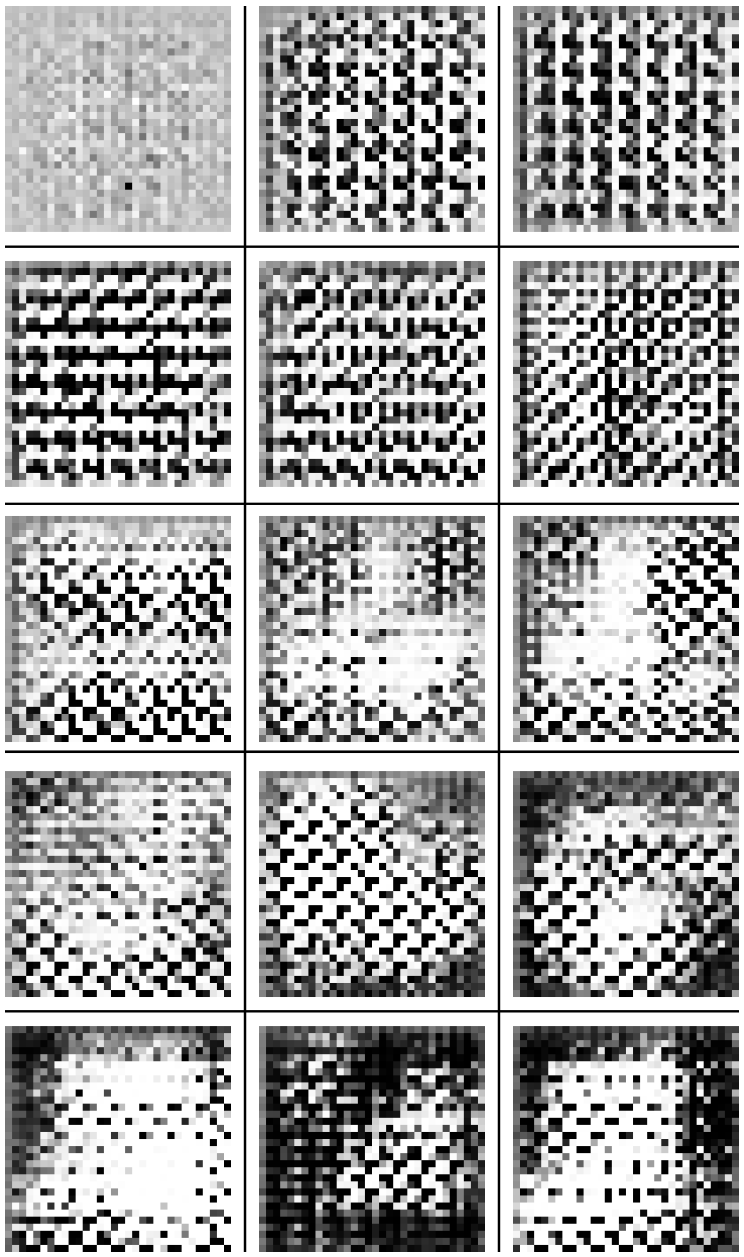

Figure 14: Figure is read from left to right and top to bottom. The figure shows the first column of figure 13.

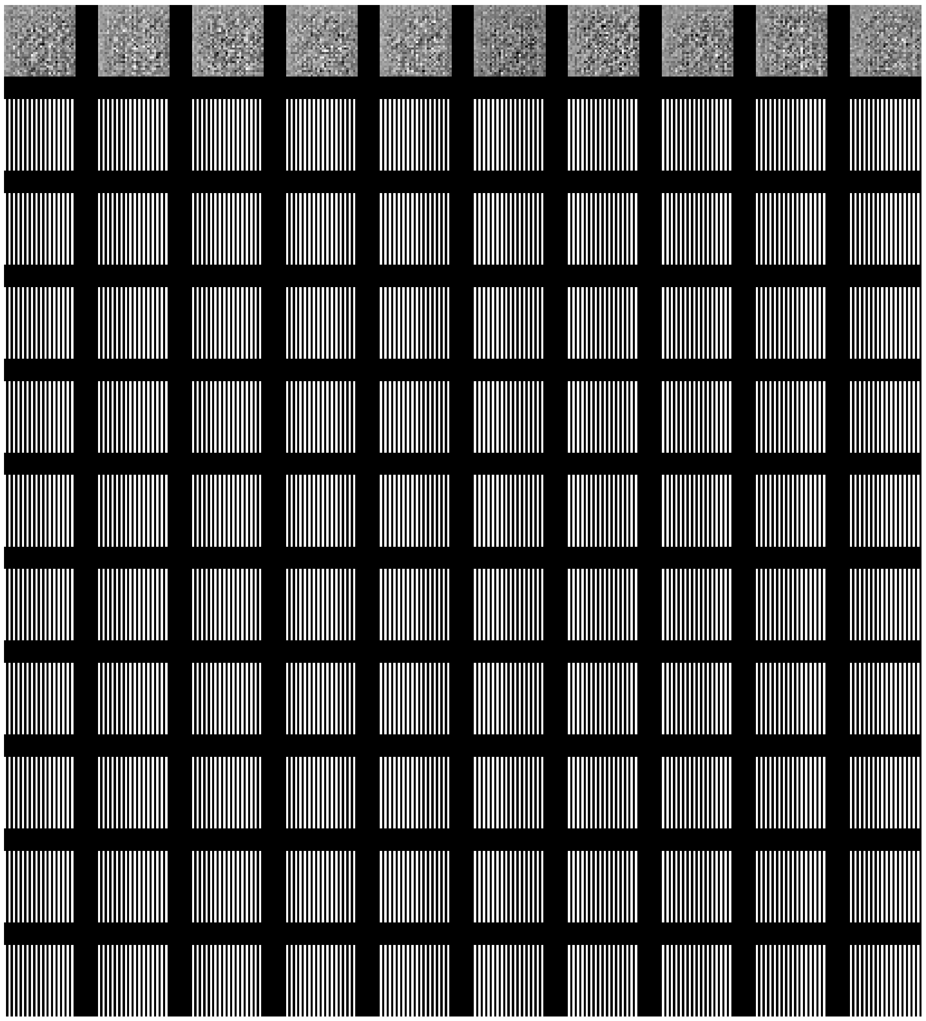
Figure 15: Total overview of output of G in BeatGAN 2.1 (32×32 pixels). Epochs are separated by horizontal black bars. Every row contains 16 generated samples from G. Epochs from top to bottom: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Losses for both D and G did not converge.

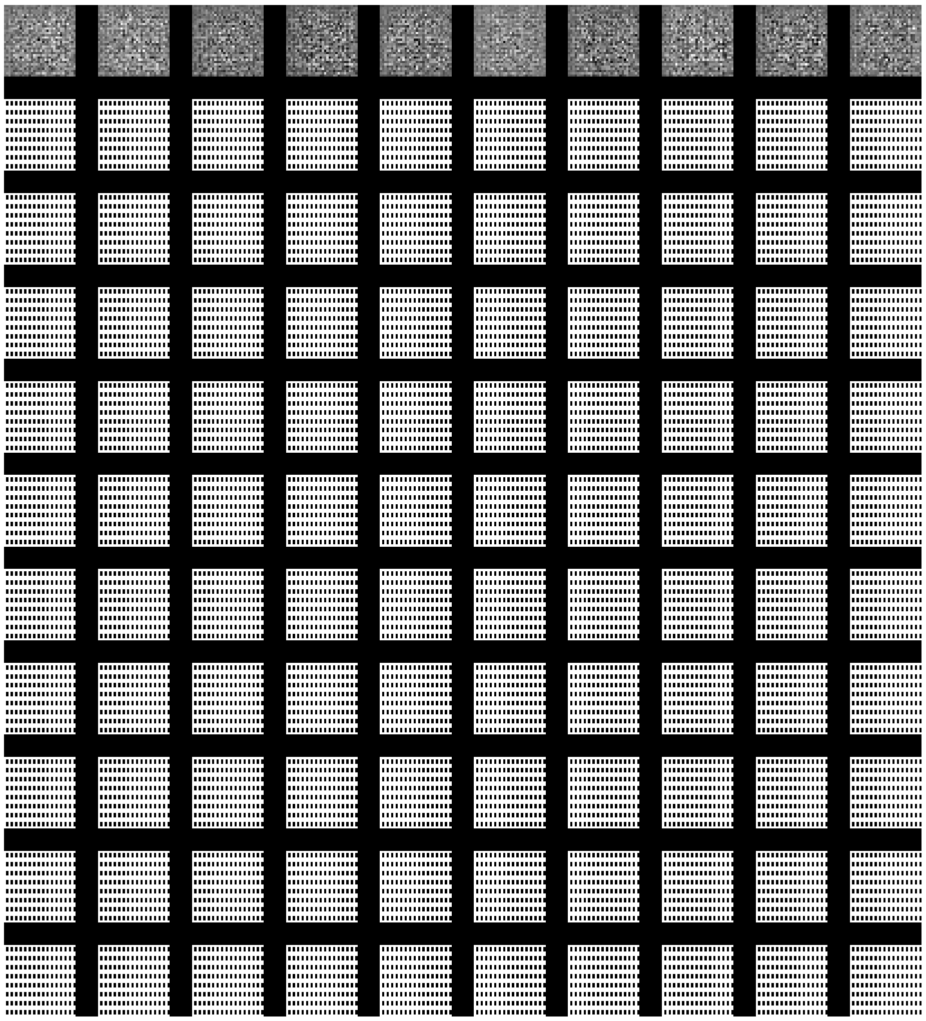

Figure 16: Total overview of output of G in BeatGAN 2.2 (32×32 pixels). Epochs are separated by horizontal black bars. Every row contains 16 generated samples from G. Epochs from top to bottom: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Losses for both D and G did not converge.

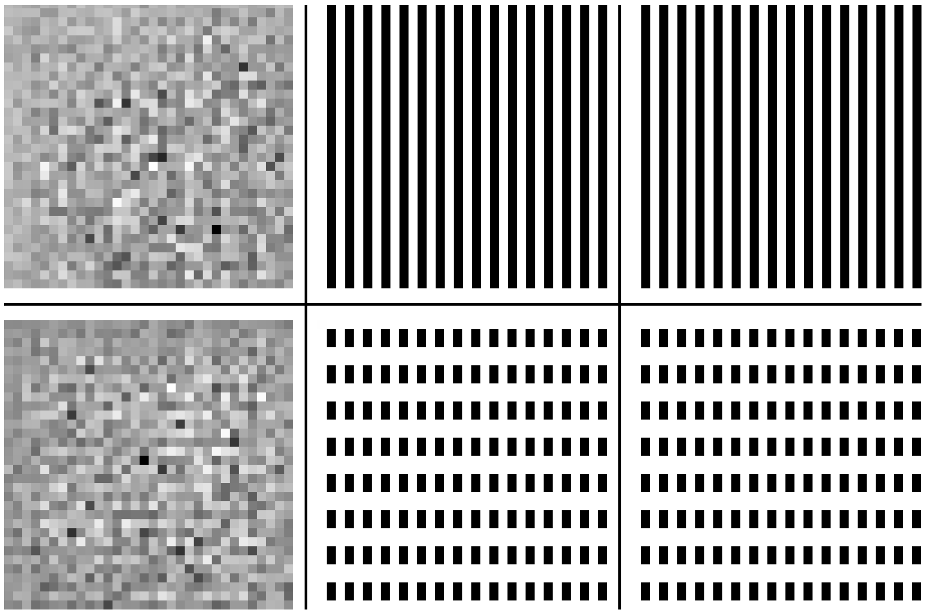

Figure 17: Figure is read from left to right. The figure shows the output of G for BeatGAN 2.1 (first row) and BeatGAN 2.2 (second row), at epochs 0, 1 and 10.

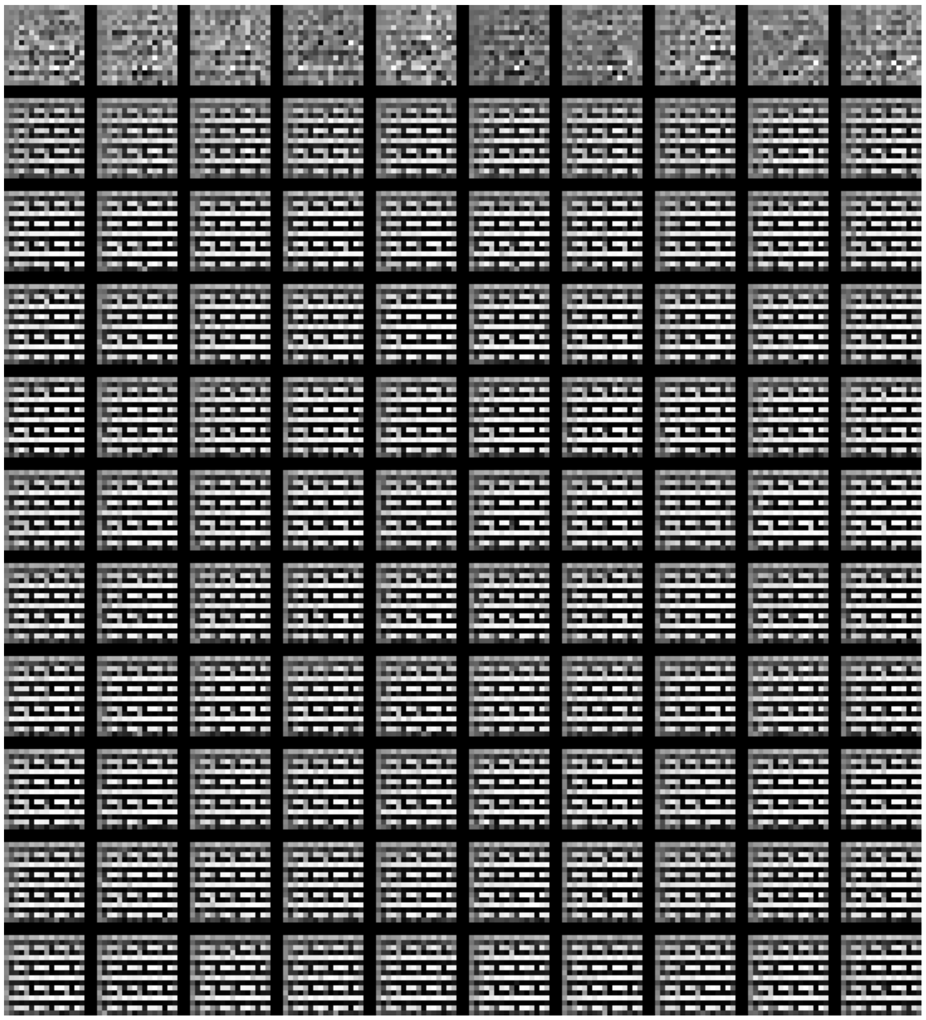

Figure 18: Total overview of output of G in BeatGAN 3 (16×16 pixels). Epochs are separated by horizontal black bars. Every row contains 16 generated samples from G. Epochs from top to bottom: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Loss for G converged to 0.693, loss for D converged to 1.385.

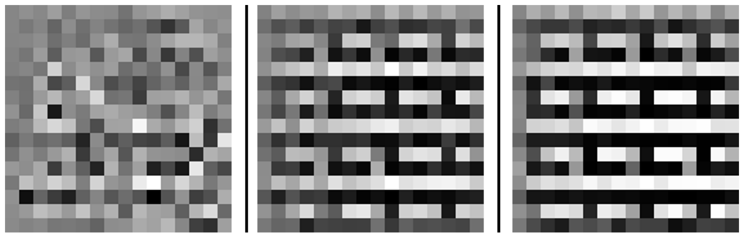

Figure 19: Figure is read from left to right. The figure shows the output of G for BeatGAN 3 at epochs 0, 1 and 10.

5.2 Audio Output

The middle 10 pixel rows were used to convert the images to MIDI. Images of 32×32 pixels were sliced to 10×32 pixels. Images of 16×16 pixels were sliced to 10×16 pixels. MIDI output of the final epochs of every BeatGAN version can be found on the GitHub repository.

6 Discussion

In this chapter we analyze our results, highlight the main problems with our GAN and review our method. We will suggest avenues for further research in the final section.

6.1 Analysis of Results

Comparing the image output of all models at the final epoch to the image samples in the data set, we can state that the generated images do not resemble images in the data set. For all versions of BeatGAN 2, and BeatGAN 3, we can see that the G did not generate output just in the middle 10 pixels of the image, but generally used the whole pixel space to generate patterns. In the image output of BeatGAN 2.0, we see that grey/black pixels start to emerge in the later epochs. We do not know what features the CNNs are learning. For more in-depth analysis we could look at the generated feature maps. Feature map visualisation of CNNs is possible [22], but lies outside the scope of this research.
After converting the images to MIDI, we are able to identify a steady pulse in all outputs. However, we do not conclude that this pulse is caused by the GAN interpreting rhythm. Because every column in the images corresponds to a sixteenth note, we conclude that this steady pulse is a result from our particular choice of data distribution. We converted a random noise sample from epoch 0 of BeatGAN 2 to MIDI, which produces similar results to all generated output of G. We can conclude that the pulse we identified is caused by the data representation. In the output of BeatGAN 1, we are able to identify that the second and fourth beats are accentuated with a hi-hat. The output

of BeatGAN 2 also includes a form of fade-in and fade-out, which is caused by the gray/black pixels on the left and right of the generated image. We find the MIDI results to be inconclusive, as it is hard to determine if some of the faint features present in the generated MIDI files were learned features, or produced by random chance.

6.2 Mode Collapse and Vanishing Gradient

A GAN has a lot of moving parts. The implementation of the parts is left up to the programmer, which allows for many degrees of freedom. The task of the programmer is to find a good balance between D and G. Any change in the model's implementation affects this balance. Identifying training problems in GANs is difficult because of the many variables involved, which makes balancing D and G a difficult task. Finding the perfect balance is analogous to finding a needle in a haystack. However, we are convinced that one can decrease the volume of the haystack, by choosing the right variables. We suspect that training GANs is a trainable skill, that heavily relies on intuition and background knowledge. GANs are fairly unstable because of the many variables involved. There are two main scenarios in which a GAN is unbalanced. Either G is too strong, this is called mode collapse, or D is too strong, which is called vanishing gradient [28].

Figure 17 shows two examples of mode collapse after epoch 0. G generates a single pattern, and does not change the pattern during training. It's possible that D is stuck in a local minimum. G has then found a pattern which is able to fool D in that local minimum. Since D cannot escape the minimum, G will continue generating the same pattern.

The vanishing gradient problem occurs whenever D is too powerful. If D is very good at classifying data, it's loss function will converge to some minimum. This means that D's gradient to improve becomes smaller and smaller. G learns by ascending the gradient calculated according to D's predictions. If D's gradient gets smaller, G's gradient will also get smaller. If D has reached a minimum, G will not improve.

During the training of BeatGAN, we encountered both mode collapse and the vanishing gradient problem. Solving mode collapse would often result in vanishing gradients and vice versa. When balancing D and G, we accumulated the sentiment that these two problems are very much connected. Given that theoretically, GANs always converge, there is some form of balancing D and G, such that neither of the problems occur. However, since there are so many variables to adjust and design choices to be made, it is hard to say how much room for error there is when training a GAN. It seems as though one of the bottlenecks in this balancing problem is computational power. A lot of trade-offs are made during training of neural networks. These trade-offs often involve computational speed and performance of the model. Increasing computational power would allow us to move towards better performance. We think that the margin for error when balancing a GAN will increase as computational power increases.

A lot of research relating to GANs is focused on improving stability of the networks. Authors often propose a method which offers a solution for some GANs, but not for all GANs. The general view is that these methods should be used as heuristics. When arriving at this point in the implementation of the GAN, the approach becomes much more trial-and-error, without much theoretical argumentation for the heuristics one chooses to implement. Again, with the many variables involved, one can imagine that not every heuristic will work for every GAN.
Our research eventually stranded at the trial-and-error phase, in which we made adjustments to both networks and the data set, and used alternative algorithms for decreasing the loss functions of the networks. Repeatedly training the networks costs time, which ran out before we were able to achieve GAN convergence.

6.3 Review of Method

We created a DCGAN which we trained on a custom data set. The custom data set contains images or rhythmic patterns. A balance between D and G was not achieved. BeatGAN produced inconclusive output.
Our GAN did not output the desired results we aimed to produce. No version of BeatGAN was able to output images that resemble the data set images, nor MIDI files that resemble the data set MIDI files. It is interesting however, that the generated patterns are often very structured. We think that this is a result of the data representation. We do not question the power of GANs as generative models. There exist many GANs which are able to produce high-quality images, similar to the data set the GAN was trained on.
Our main concern with our method lies with the data set. Our data set is fairly small, in comparison to more widely used data sets like CIFAR-10 and MNIST, which contain around 60.000 and 70.000 images respectively [26][27]. We do not think that the variation in the data set is too large. Like CIFAR-10 and MNIST, we included many different classes (genres) of images. GANs have been shown to work without having to specify a class label for all classes in a data set. The real image/fake image distinction should be sufficient to generate novel data.
Another concern regarding the data set is the usability of the chosen image representation for the DCGAN architecture. Our main concern is the sparsity of white pixel values in the data samples. We think that there might not be enough pixel data present in the samples to correctly learn the relations these pixels have to one another. Gillick et al. [12] used a similar music representation. However, they did not use CNNs to generate new music. The use of a DCGAN in combination with our music representation has not yet produced any results as far as we are aware. We conclude that more research is needed to determine if images with a sparse number of informative pixels can be used successfully in a DCGAN.
We do think that it is possible to obtain a balanced network using our method. However, the steps we need to take are obscured by the complexity of the overall project.

6.4 Future Research

Goodfellow [19] suggests adding some form of labelling to the GAN. Adding more labels seems to always improve performance of the GAN. Adding labels would involve expanding the discriminator and the generator, such that they can be trained and nudged in some direction using additional labels. This type of framework results in a conditional GAN.
Karras et al. [29] propose a method for training GANs with limited data. Using datasets consisting of a couple thousand examples, the authors show that it is possible to train a GAN. This method seems promising for our research, given that we identified our limited data set as a possible problem in our research.
Krizhevsky et al. [20] apply data augmentation in their GAN. Data augmentation involves making small adjustments to copies of samples in the data set, increasing the amount of data in the data set. For example, adjustments can include rotation and adding noise.
Since training a GAN involves a lot of trial and error, and adjusting the parameters of the model, we suggest applying evolutionary algorithms as another layer of abstraction on top of the GAN. These algorithms could learn the optimal values of all parameters. We think that this would help tremendously in the trial-and-error phase of training a GAN. Training a single version of a GAN costs a lot of time. A large drawback to this method is computational power, as it would require training many different versions of the GAN. Another drawback of this method is implementing a fitness function for GANs. GANs are fairly difficult to evaluate, and evaluating G's output requires a human eye.

7 Conclusions

We will try to answer our research question by first highlighting the sub-questions.

How can rhythm be represented in a visual format?
We chose an image format which was able to account for the three dimensions we defined for music: time, pitch and velocity. We placed time on the x-axis of the images, pitch on the y-axis of the images, and represented velocity as the brightness of a pixel. This type of implementation has seen success in earlier work by Gillick et al. [12].

What types of neural networks should be selected within the GAN structure?
Following a commonly used type of GAN, the DCGAN, we chose the neural networks in our GAN to be convolutional neural networks. CNNs are often used in image generation and classification [19][20][22]. A reason CNNs are useful in this field is because they take pixel context into account.

How does the output of the GAN compare to man-made rhythms?
Since the output of BeatGAN was inconclusive, we are not able to make a fair comparison with man-made rhythms. A pulse in the output could be identified,

but we concluded that the pulse was an effect of our image representation of the output. The output of BeatGAN does not represent the man-made rhythms the network was trained on.

How suitable are generative adversarial networks for generating rhythmic patterns represented in a visual format?
We do not think that our visual representation is necessarily the cause of non-convergence. The use of a CNN is also not the cause of non-convergence. However, we do feel as though a DCGAN is not the best fit for generating images like the samples in our data set. The sparsity of white pixels in the data set does not provide much information for the D to process. The white pixels are sparse, but also very dense with information. We suspect that CNNs are not very useful to learn this kind of representation. Feature map visualisation would allow us to investigate what kind of features D is learning. This can provide clues for how BeatGAN operates, which in turn can steer us in a direction to better balance D and G.
We also suspect that balancing the network may become easier if we increase the size of the data set, which would allow us to use a larger batch size. This in turn allows for more stable training since more data is taken into account when using gradient descent, which allows for better informed updates of the weights and biases. We also might achieve better results if we use different types of neural networks in the GAN.
The MIDI results do not contain much rhythmic information. Looking at the generated visual patterns however, we do conclude that the networks are learning. The output of G seems to show intelligence, especially considering the fact that the input of G is a random noise vector.
Even though the results may be inconclusive, we have added knowledge to the field of DCGAN training, by showing that the data representation can possibly have a large impact on the difficulty of training a DCGAN.
We conclude that our implementation of a DCGAN, trained on our visual representation of music, was difficult to train. The difficulty of training BeatGAN could indicate that our method is not very suitable for generating rhythmic patterns. However, the complexity of the project obscures the drawing of any strong conclusions.

Reflections and Acknowledgment[3]

At the beginning of this project, I was very excited about the path that lied ahead. Diving into the literature and seeing what results were generated in preceding research, opened my eyes for what can be achieved within the field of generative modeling. My excitement only grew while taking the first steps along the path, searching for useful data sets, and orienting myself on what type of networks can be used to achieve the results I set out to achieve. During these first stages, my personal goal was to create a system that could generate new music with a single button press.
At some point in time, I was introduced to the GAN architecture, which immediately grabbed my attention. The elegance of the idea and the proof of it's convergence amazed me. I immediately felt that I wanted to use this architecture in my research. Further exploring GANs did bring me to a realisation that the process of implementing BeatGAN would be arduous and would require a lot of time. On top of the implementation, the research would also require a comprehensive report.
At this point, I would like to thank Dr. Tomas Klos for his supervision of the project. He not only provided me with tips on how to schedule my work, but was also an excellent sparring partner when presenting design choices in our weekly meetings on monday. I thoroughly appreciated the interest Dr. Klos showed in me and my research and I wish him well.
The magnitude of the project became increasingly clear as time went on. The first realisation came when I finished the processing of the data set, which by itself took up a lot of time. At that point, I had not done any research on how to implement the GAN, which worried me, as I still needed to write the full data set chapter.
Actually implementing the GAN was easier than I expected, leaving me with a lot of time to train the GAN. Training the GAN however, was harder than I expected. Listening to the results made me realise that I did not achieve my personal goal. Strangely, I was not let down by this realisation. At this point in the research, my excitement for GANs had only grown, and I was proud of the knowledge I gained and the work that I had done.
In the last few weeks my day to day consisted of writing and improving the research. Living in this type of rhythm resulted in a tunnel vision mindset of finishing the thesis as best as I could. I thoroughly enjoyed writing the discussion and conclusion chapters, at which point I realised that the project was coming to a close. This realisation made me take a step back and look at the project that was the result of weeks of effort. Being able to see the project in its entirety was relieving, and I realised that I was proud of my work.
My sentiment for the future is hopeful, and I am excited to see what new techniques and architectures the future will bring. GANs offer us proof that neural networks are not just programs to be used in isolation. As computational power increases, I argue that more architectures combining different neural networks

[3]This section was purposely written in the first person.

will emerge. I think that Ian Goodfellow has shown a level of neural network abstraction which promotes creativity for programmers and researchers; not seeing neural networks as the complex mathematical functions that they are, but seeing them as tools for sub-problems in the increasingly complex problems the field of machine learning presents to us. API's such as PyTorch, TensorFlow and Keras all follow the trend of abstracting away the complex implementation details of neural networks. One can implement powerful neural networks with just a few lines of code. Understanding the general workings of a neural network is now sufficient to create one. As these API's introduce machine learning to a broader audience, my excitement only grows, wondering what type of new and creative solutions the field will bring.

References

[1] Ravignani, A., Delgado, T. & Kirby, S. (2017) Musical evolution in the lab exhibits rhythmic universals. *Nat Hum Behav 1, 0007*

[2] Merriam-Webster. (n.d.). Rhythm. In Merriam-Webster.com dictionary. *Retrieved May 27, 2021, from https://www.merriam-webster.com/dictionary/rhythm*

[3] Olah, C., Mordvintsev, A., & Schubert, L. (2017). Feature visualization. *Distill, 2*(11), e7.

[4] *Obvious – Art & AI.* (2018). Obvious – Art & AI. https://obvious-art.com/

[5] Karras, T., Laine, S., Aittala, M., Hellsten, J., Lehtinen, J., & Aila, T. (2020). Analyzing and improving the image quality of stylegan. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 8110-8119).

[6] Hadjeres, G., Pachet, F., & Nielsen, F. (2017, July). Deepbach: a steerable model for bach chorales generation. In *International Conference on Machine Learning* (pp. 1362-1371). PMLR.

[7] Yang, L. C., Chou, S. Y., & Yang, Y. H. (2017). MidiNet: A convolutional generative adversarial network for symbolic-domain music generation. *arXiv preprint arXiv:1703.10847.*

[8] Dhariwal, P., Jun, H., Payne, C., Kim, J. W., Radford, A., & Sutskever, I. (2020). Jukebox: A generative model for music.*arXiv preprint arXiv:2005.00341.*

[9] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2014). Generative adversarial networks. *arXiv preprint arXiv:1406.2661.*

[10] Radford, A., Metz, L., Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434.*

[11] Mogren, O., (2016). C-RNN-GAN: Continuous recurrent neural networks with adversarial training. *arXiv eprint arXiv:1611.09904*

[12] Gillick, J., Roberts, A., Engel, J., Eck, D., & Bamman, D. (2019, May). Learning to groove with inverse sequence transformations. In *International Conference on Machine Learning* (pp. 2269-2279). PMLR.

[13] Roberts, A., Engel, J., Raffel, C., Hawthorne, C., & Eck, D. (2018, July). A hierarchical latent vector model for learning long-term structure in music. In *International Conference on Machine Learning* (pp. 4364-4373). PMLR.

[14] General MIDI (GM 1). (n.d.). General MIDI (GM 1). *Retrieved May 27, 2021, from https://www.midi.org/specifications-old/item/general-midi*

[15] Walker, J. (2004, February). MIDICSV: Convert MIDI File to and from CSV. Fourmilab. *https://www.fourmilab.ch/webtools/midicsv/*

[16] McKinney, M. F., Moelants, D., Davies, M. E., & Klapuri, A. (2007). Evaluation of audio beat tracking and music tempo extraction algorithms. *Journal of New Music Research, 36*(1), 1-16.

[17] Sullivan, J. (n.d.). Beat Detection Using JavaScript and the Web Audio API — Beatport Engineering. Joesul. *Retrieved May 27, 2021, from http://joesul.li/van/beat-detection-using-web-audio/*

[18] B., A. (2016, June 29). Principals of Basic Drum Beats for Rock, Part 1. Musika Lessons Blog. *https://www.musikalessons.com/blog/2016/06/basic-drum-beats-for-rock/*

[19] Goodfellow, I. (2016). Nips 2016 tutorial: Generative adversarial networks. *arXiv preprint arXiv:1701.00160.*

[20] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems, 25,* 1097-1105.

[21] Education, I. C. (2021, January 6). Convolutional Neural Networks. IBM. *https://www.ibm.com/cloud/learn/convolutional-neural-networks*

[22] Zeiler, M. D., & Fergus, R. (2014, September). Visualizing and understanding convolutional networks. In *European conference on computer vision* (pp. 818-833). Springer, Cham.

[23] Bjorck, J., Gomes, C., Selman, B., & Weinberger, K. Q. (2018). Understanding mini-batch normalization. *arXiv preprint arXiv:1806.02375.*

[24] Ioffe, S., & Szegedy, C. (2015, June). mini-batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning* (pp. 448-456). PMLR.

[25] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1), 1929-1958.

[26] Krizhevsky, A., & Hinton, G. (2009). Learning multiple layers of features from tiny images.

[27] LeCun, Y., Cortes, C., & Burges, C. (1998). MNIST handwritten digit database. yann.lecun.com. *http://yann.lecun.com/exdb/mnist/*

[28] Common Problems — Generative Adversarial Networks —. (2020). Google Developers. *https://developers.google.com/machine-learning/gan/problems*

[29] Karras, T., Aittala, M., Hellsten, J., Laine, S., Lehtinen, J., & Aila, T. (2020). Training generative adversarial networks with limited data. *arXiv preprint arXiv:2006.06676.*

[30] Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., & Courville, A. (2017). Improved training of wasserstein gans. *arXiv preprint arXiv:1704.00028.*

[31] Arjovsky, M., & Bottou, L. (2017). Towards principled methods for training generative adversarial networks. *arXiv preprint arXiv:1701.04862.*

[32] Miyato, T., Kataoka, T., Koyama, M., & Yoshida, Y. (2018). Spectral normalization for generative adversarial networks. *arXiv preprint arXiv:1802.05957.*

[33] Hearn, D. (2021, 31 maart). A Complete Guide to Time Signatures in Music. Musicnotes Now. *https://www.musicnotes.com/now/tips/a-complete-guide-to-time-signatures-in-music/*

[34] Anderton, C. (2018, 27 februari). 5 MIDI Quantization Tips. The MIDI Association. *https://www.midi.org/midi-articles/5-midi-quantization-tips-1*