TECHNICAL UNIVERSITY OF CRETE

Music generation with Neural Networks

Argiris Papadopoulos

Electrical Computer Engineering

Supervised by

Dr. Michael G. Lagoudakis
Department of Computing

14 February 2022

Acknowledgements

First and foremost, I would like to give my warmest thanks to my supervisor Dr. Michael G. Lagoudakis, whose guidance and advice gave shape to this work. I would also like to thank the rest of the committee members, professors Euripides G. M. Petrakis and Georgios Chalkiadakis for their thoughtful comments and suggestions.

Second, I would like to give special thanks to my partner, her existence and presence the greatest gifts I have ever received, and for standing with me at my best and worst at the most crucial years of my life. My family for their continuous support and understanding while undertaking my research.

Last but not least I want to express my thanks to my friends and classmates whose support was essential for the completion of this work.

List of Figures

List of Tables

Contents

Abstract

In this thesis it is presented how deep neural networks can be utilized to generate musical sequences. The implemented models train from MIDI files that are converted to images. After the training process the models generate images that are decoded to MIDI tunes. These files are ideal due to their discrete nature. Generative modeling is a very popular task in deep learning and has many applications. While the automated music generation is not deemed important for scientific purposes, it is an interesting topic, the process can be used for other tasks similar to such image generation, plus with desired results it could help musicians improve and explore original tunes based on genres or types of music. This project utilizes famous generative models, Variational Autoencoders and Generative Adversarial Networks. Such networks proved to work well and produced interesting results while maintaining simplicity and low computational power and time required. By undertaking this task we learnt the engineering and some tricks used for creating great generative models, as well as how to shape and form data in a way that helps those models learn easier and faster. The goal of this work is to construct models that are able to learn the basic behavior of each input dataset and try to replicate such samples. We seek to compare the models and infer results, not to generate consumer-grade music, as such task would over-complicate this work.

1 Introduction

Artificial intelligence and specifically machine (or deep) learning has been rising in popularity in the last decades. It is a thriving field whose applications are dedicated to automate laborious routines, conceive the essence of speech or images, identify patterns that can contribute to medical analysis and so on. AI is a buzzword which consists of hundreds subsets and one of them is deep learning. Even deep learning has no consensual definition, it consists of machine learning (ML) techniques based on artificial neural networks. Those networks are inspired by the biological human neural system and how they process and weigh information is fairly similar. A set of neurons fire up given that information from our senses as input and the network outputs a feeling, a thought, a memory, or even possible actions or decisions for a matter. Same goes to the process of artificial neural networks, which if trained well, they can be used for classification, choosing an action (Deep Q Agent), data compression (Autoencoders), data generation (generative models such as GANs, Variational Autoencoders), and much more.

In the case of generative modeling, as this work is based on, the networks are trained upon processed musical files (MIDI) which are translated to images. As the models train, they are able to produce images that resemble closely the ones from the input set of data. The goal in such cases is not to produce same results, but to explore and generate original data.

Neural networks are ideal for such tasks as they are able to handle massive amounts of data, while being as efficient as possible with affordable computing power. The same is presented in this works case as we offer models that can run even in CPU environments, as they train upon thousands of samples.

1.1 Computer-Based Musical Systems

Researchers have been trying to create a bridge between computers and music as the first computer systems were created. It has been an active topic for many decades with different iterations and approaches (Briot, Hadjeres and Pachet 2017).

The first music generation program was introduced in the early 1960s by Max

Mathews at Bell Labs, a research center run by phone company AT&T. Mathews was mainly researching in phone related fields, but worked on music software in his spare time. He named this program MUSIC and later on he created different versions indicated by Roman numerals. MUSIC produced its first sound in 1957 playing a 17 seconds long melody named “The Silver Scale”. The same year, the first score was composed by a computer named "The Illiac Suite". It was produced by the ILLIAC I computer at the University of Illinois at Urbana-Champaign (UIUC) in the United States by Lejaren A. Hiller and Leonard M. Isaacson, both musicians and scientists. It was the first example of algorithmic composition, by utilizing stochastic models, such as Markov chains, for generation as well as rules to filter generated material according to desired properties.

In 1983 was released a model of synthesis, Yamaha DX7 synthesizer, based on frequency modulation (FM) by Chowning. The same year the MIDI interface was launched which created a link between software and instruments (including the DX7 synthesizer). Another case was the development of the Max/MSP environment, by Puckette at IRCAM, which allowed for real-time synthesis and interaction.

In the year of 1962, Iannis Xenakis used computers to composite music with a stochastic approach. He created ST Program, a stochastic music program, written in FORTRAN, which used probability distributions and calculated various possibilities designed by the composer in order to generate musical samples. Another approach was that of Ebcio Glu, which followed the direction of "The Illiac Suite", which had certain rules and constraints to specify the style of a given corpus. He created CHORAL, a composition program of a four-part chorale in the style of Johann Sebastian Bach. In the late 1980s, a system called Experiments in Musical Intelligence (EMI), developed by David Cope, extended that approach. His program learned from a corpus of a composer’s scores and was able to create its own contraints and rules.

1.2 History of MIDI

MIDI format played a huge role in the music industry as it offered a general compatibility between programs and electronic instruments. This work's interaction with music is done solely by the use of MIDI files, as they are excellent in offering discrete information, which is ideal for the deep learning method presented.

MIDI which is an acronym for Musical Instrument Digital Interface, first saw the light in the early 1980s. It's main purpose was to synchronize electronic musical instruments manufactured by different companies, as each company had their own standards of synchronization such as CV/gate and Digital Control Bus (DCB). This lack of synchronization led Ikutaro Kakehashi, founder of Roland to bridge this gap, as he felt this limitation would delay the growth of electronic music. Ikutaro went on to propose this idea of creating such a machine of synchronization to Oberheim Electronics found by Tom Oberheim developer of Oberheim Systems, which he thought was difficult to carry and use and wanted a simpler design. Together with Dave Smith from Sequential Circuits, they discussed a simpler and cheaper alternative with American and Japanese companies (Yamaha, Korg, Kawai), and initially, only Sequential Circuits and Japanese companies were interested.

Dave Smith and engineer Chet Wood from Sequential Circuits managed to create a universal interface for communication between equipment from different companies. Together they proposed the standard in a paper, Universal Synthesizer Interface, at the Audio Engineering Society show in October 1981. The standard went under modification by Roland, Yamaha, Korg, Kawai and Sequential Circuits, and Kakehashi showed preference for the name Universal Musical Interface (UMI), pronounced you-me, but Dave Smith felt it was banal. Also Smith liked the word "instrument" instead of "synthesizer" so the proposition Musical Instrument Digital Interface (MIDI) was accepted and announced in the October of 1982 issue of Keyboard.

Smith demonstrated his design by connecting a Prophet 600 to a Roland JP-6, two different manufactured synthesizers, at the 1983 NAMM Show. Those were the first synthesizers that were released with the MIDI standard (1982), and later

in 1983, Roland TR-909, the first MIDI drum machine, and Roland MSQ-700, the first MIDI sequencer, were released. The first computer to support MIDI, the NEC PC-88 and PC-98, was released in 1982.

Later in 1984 at the Summer NAMM Show in Chicago the MIDI Manufacturers Association (MMA) was formed. The MIDI 1.0 Detailed Specification was then published at the 1985 Summer NAMM show by MMA. The standard advanced in 1991 by adding standarized song files (General MIDI) and adapted to new connection standards such as USB and FireWire. In 2016, the MIDI Association was formed to continue overseeing the standard. The MIDI 2.0 standard was proposed in January 2019 and was introduced at the 2020 Winter NAMM Show.

1.3 Autonomy versus Assistance

Referring to computer-based music generation, or any generative problem, there are two approaches when tackling such a task.

- Autonomy: developing autonomous music-generating systems aimed to create musical patterns from scratch.
- Assistance: programs that aim to assist musicians in further exploring musical possibilities, by receiving inputs, rules or restrictions by human hand.

The task of automated music generation is quite interesting as it can offer exploration about the process of composition. Two recent examples being the deep-learning based Amper and Jukedeck systems/companies aimed at the creation of original music for commercials and documentary

On the other hand, assistance computer-based music programs are developed in order to assist humans in various steps in music creation: composition, arranging, orchestration, production. In practice, composers rarely create their own music from scratch, as they reuse or adapt based on their experience, consciously or unconsciously, while following music principles and rules, such as harmony and scales theory. The assistance program then suggests or complements the various musical steps, which the human composer tackles. Such examples of assistance computer-based music programs are the FlowComposer environment developed at

Sony CSL-Paris and the OpenMusic environment developed at IRCAM.

This work is aimed towards automated music generation, more specifically at small 2 measure (depending on the time signature) musical parts. Most music generative tasks follow the autonomous approach, although more and more systems begin to address the option of assistance and human interaction.

1.4 Deep Learning

The generality that deep learning techniques offer is ideal for such generative tasks. As opposed to handcrafted models, grammar-based, or rule-based music generation systems, a machine learning-based generation system can be agnostic, as it learns a model from an arbitrary corpus of music. Furthermore, a certain model can be used to train upon various and different musical genres and themes. Therefore, as more large scale musical datasets are made available, a machine learning-based generation system can detect the musical style from a corpus and generate new relevant content. Rather than creating a complicated system filled with rules and grammars, there is the approach of a deep learning model which is able to process raw unstructured data, from which it identifies musical distributions and patterns, as its layers extract higher level representations adapted to the task.

2 Background

The knowledge required to understand this work is split into two sections, music and the machine learning technique called neural networks. Basic principles are only required for both of the fields in order to comfortably read this work. Specifically,

- little to no music theory knowledge is required, as with the use of MIDI files the data is discrete, and
- with basic understanding of dense neural networks and training processes, other layers and concepts are easily absorbed.

2.1 Basic Music Theory

Music theory examines the fundamentals of music and provides principles and rules, which are a guideline for musicians and allow them to create musical sequences. Music theory has been practiced for thousands of years and it branches to many topics and levels. This section will focus only to the very basics, as it is deemed enough to understand the following sections (Schmidt-Jones 2013).

2.1.1 Measure and Bar Lines

Sheet music is mostly read and written in a format of five horizontal parallel lines also known as the pentagram which is then sequenced into measures (Figure 1). Measures exist to organize long pieces of music into smaller units. Professional musicians read music or perform a piece of music in real-time as they are able to read each measure at a time.

Measures are split by the use of vertical bar lines. There are different bar line types that offer musicians easy sight-reading and better overall performance.

Bar line types (Figure 2):

- Single bar line, a single vertical line that indicates the end of one measure and the beginning of another.

Figure 1: Measures and pentagram

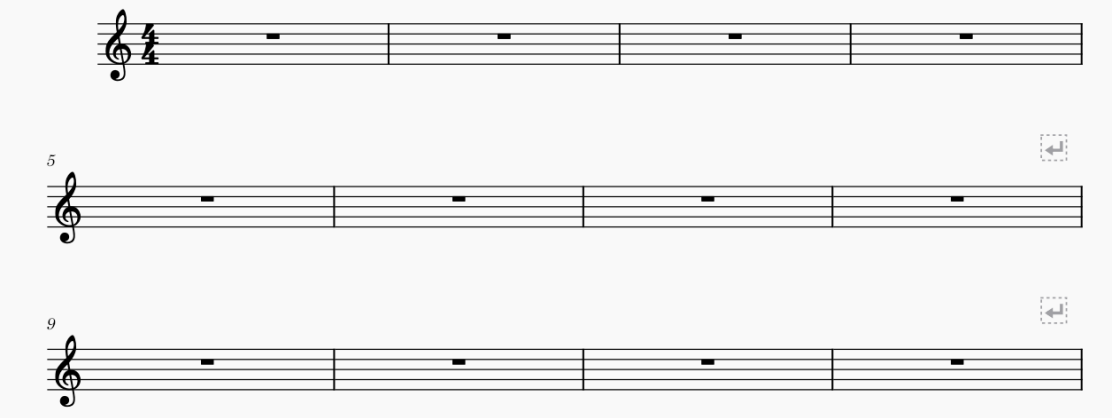
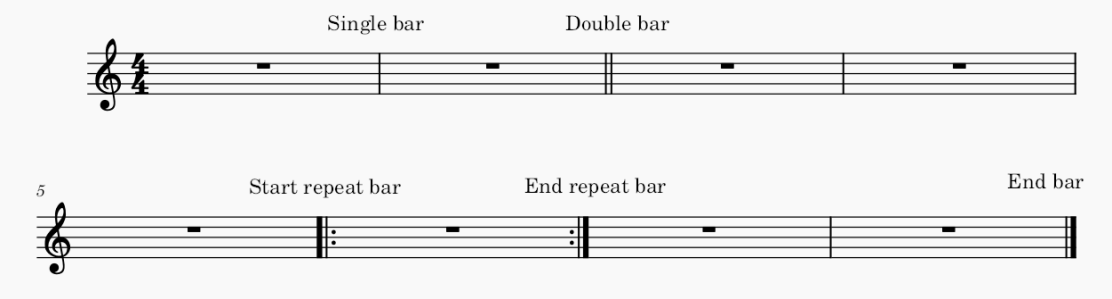


Figure 2: Bar line types



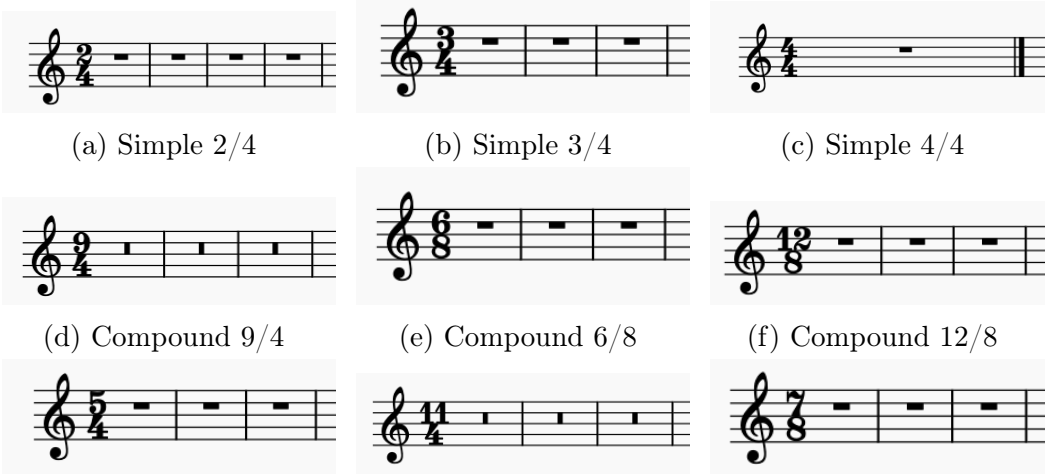
- Double bar line, two side-by-side vertical lines, indicating the end of one section and the beginning of another.
- End bar lines, two vertical lines, one bolder, which indicate the end of the musical composition.
- Start repeat, two vertical lines, one bolder, with two dots on the right, indicating the following sequence will be repeated.
- End repeat, two vertical lines, one bolder, with two dots on the left, indicating the musician should return to the last start repeat bar lines and replay the sequence once.

For obvious reasons there cannot exist start repeat-type bar lines without end

repeat, and the other way around. Furthermore the sequence inside the repeat bar lines is repeated once, unless indicated otherwise, usually with a number next to the end repeat bar lines.

2.1.2 Time Signature

Musical time signatures is a fraction, which show the number of beats per measure (the top number in a time signature, numerator) and the duration of each beat (the bottom number in the time signature, denominator). For instance, 3/4 time signature indicates there are three beats per measure and each beat has the duration of a quarter note.



(a) Simple 2/4 (b) Simple 3/4 (c) Simple 4/4

(d) Compound 9/4 (e) Compound 6/8 (f) Compound 12/8

(g) Complex 5/4 (h) Complex 11/4 (i) Complex 7/8

Figure 3: Different types of Time Signatures.

The three basic types of time signatures:

- Simple: The most common simple time signatures are 2/4, 3/4, 4/4, and 2/2. The 4/4 time signature indicates that there are four beats per measure (numerator) and each beat counts as a fourth note (denominator). Furthermore for 2/4 and 3/4, there are two and three quarter note beats per measure, respectively.
- Compound: Common compound time signatures include 9/4, 6/8, and 12/8. The beat of a piece of music with a compound time signature is broken into

a three-part rhythm. In each of the cases above, quarter or eighth notes are combined in multiples of three.

- Complex: Complex time signatures are more common in music written after the nineteenth century. Complex time signatures do not follow typical duple or triple meters. Examples of complex time signatures include: 5/4, 11/4, and 7/8.

While there are many types of time signatures, below are the most commonly used by musicians.

- 2/4, two quarter-note beats per measure.
- 3/4, three quarter-note beats per measure.
- 4/4, four quarter-note beats per measure, also known as common time and notated as a "C".
- 2/2, two half-note beats per measure, also known as cut time and notated as a "C" with a vertical slash through it.
- 6/8, six eighth-note beats per measure.
- 9/8, nine eighth-note beats per measure
- 12/8, twelve eighth-note beats per measure.

The first note of every bar or measure is called the downbeat. Every measure has strong (loud) and weak (silent) beats. In a time signature like 4/4, the first beat of every measure is the strongest beat, and the third beat is also a strong beat. Beats two and four are weak beats. Regardless, this can vary between different genres and compositions.

2.1.3 Tempo

Tempo refers to the speed of a section of music. It can be indicated in metronome markings using beats per minute (BPM) or by using descriptive words (traditionally Italian words describe tempo like adagio or andante).

Typically the metric of BPM is self-explanatory. As its name suggests, it indicates the count of beats a minute contains. For example, a musical part with 60 BPM results in one beat per second, and with 120 BPM it is twice as fast resulting in two beats per second.

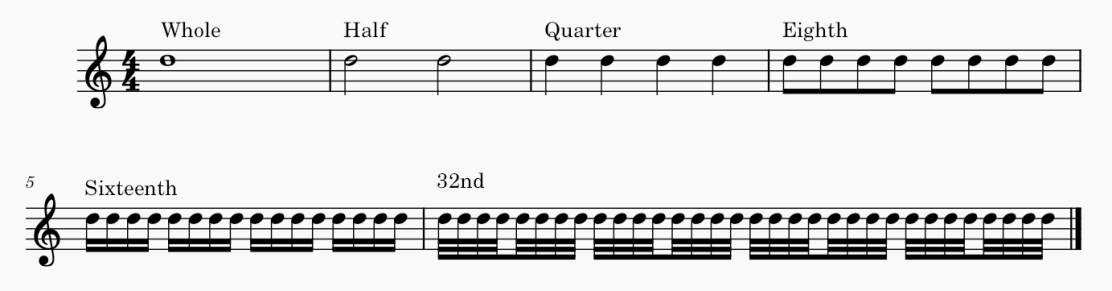
In terms of musical notation, a beat almost always corresponds with the piece's time signature. This means that the denominator of the time signature counts as one beat. For example in 2/4, 3/4, 4/4 a quarter note counts as a beat, in 6/8, 9/8, 12/8 an eighth note counts as a beat and so on.

BPM is the most precise way of tracking real time of musical sequences, both as a whole duration and as a time offset between notes. It is effective due to its precision, offering parallel play between instruments without temporal confusion in environments like bands or orchestras, as well as time consistency in solo play. If the tempo is not respected, the sound can feel out of place even in untrained ears. Typically professional musicians are able to sustain tempo during their play, as well as understand the time of a composition before performing it. Usually the most skilled musicians in tempo are percussion players as their instruments are specialised in rhythm rather than melody.

2.1.4 Note Values

Individual notes within a measure last for a specific fraction of the duration of that measure. They indicate both their pitch depending on where they are set in the pentagram, and their duration or time offset from the previous played note.

Figure 4: Note values



Notes in a part or sheet of music indicate their duration and their offset from the previous note played inside a measure. In a common 4/4 time signatures the types of notes are:

- Whole note: a note that covers the whole 4-beat measure, an empty circle
- Half note: covers half of a 4-beat measure, an empty circle with a vertical bar on top
- Quarter note: covers a fourth of a 4-beat measure, filled circle with a vertical bar on top
- Eighth note: covers an eighth of a 4-beat measure, filled circle with a vertical bar on top and a tail
- Sixteenth note: covers a sixteenth of a 4-beat measure, filled circle with a vertical bar on top and a double tail

Some musical parts have smaller subdivisions, such as 32nd, 64th and 128th notes, but musicians typically set tempos that work around the need for such small durations and offsets.

2.1.5 Music Theory in this work

In summary, musical parts have principles and rules that allow musicians to translate and read them. In any case this section is not intended to be a lesson about music theory. It just contains some basic principles that are required in order to assist in the engineering of generative modeling. Furthermore, understanding the concepts of reading sheet music, mainly serves the purpose of evaluation, to ensure that the music to image translation is correct.

The most essential part for this work is tempo, time signatures and note durations. In practice, it will be apparent that MIDI files contain information about note values and durations in a discrete way. Regardless, most time attributes are set into high time values, or ticks, and we aim to reduce that value for the image translation, e.g. a note being played with a duration of 256 ticks, needs to be reduced in order to have smaller images (less pixels) with the same information.

Furthermore the images that will be created need to have enough pixels for at least two measures, while at the same time being able to contain different time signature types, as well as normalizing the time durations and offsets in a way that information is not lost.

It is also apparent that other concepts of music theory, such as keys, scales and so on are not present due to the fact that they are not needed. This work is based on training upon musical parts that already follow music principles and rules. Regardless understanding more musical aspects could help evaluate the trained generative models, as in a more complex and better performing environment such concepts could be helpful.

2.2 Deep Learning

Machine learning has been thriving in the last decades and its techniques and tasks in everyday laborious tasks are showing great results. Their ability to process unstructed data and extract higher level representations adapted to the task suit very well for our music generation purpose. It is a field that branches in many categories, such as machine learning algorithms (Linear Regression, Logistic Regression, Decision Tree, SVM and so on), they are categorised as supervised or unsupervised learning, agent-based algorithms (Q-learning), neural networks and the list goes on. Our main focus is a branch called Deep Learning, which utilize neural networks that are trained upon a certain dataset and are able to identify patterns in order to execute a decision, usually a classification task.

The aim of this section is to present the very basic concepts of deep neural networks, which will allow a smooth transition for more unique and complex techniques used by the generative models (Piyush Madan 2020).

2.2.1 Deep Learning and the Human Brain

In order to create systems that learn similar to how a human is able to learn, the architecture of a deep learning network tends to mimic the human brain and its process in learning. Deep learning essentially was inspired by neurology, as similar to how neurons form the fundamental building blocks of the brain, its

architecture is assembles by layers of perceptrons, which are computational units allowing modeling of nonlinear functions.

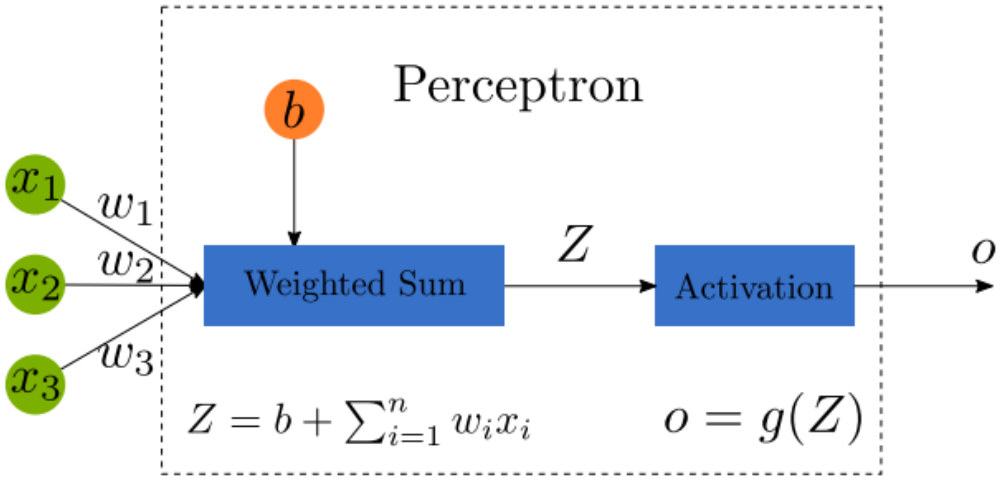
The main point of a deep learning network is the perceptron, which receives a list of input signals and transforms them into output singals, inspired by the human neuron, which in a biological manner transmits electrical pulses throughout our nervous system.

The perceptron's purpose is to understand data representation by stacking together many layers, each layer responsible for understanding some pattern of the input. A layer is a collection of perceptrons, with the aim of learning to detect a repeating occurence of values. Each layer of perceptrons is responsible for interpreting a specific pattern within the data. The interconnection of neurons in the human brain is replicated by the use of layers forming a network, so the architecture is called neural networks (or artificial neural networks).

2.2.2 Artificial Neural Networks

An Artificial Neural Network (ANN) processes input data by the use of perceptrons. Each layer contains perceptrons that will in turn feed their outputs to the ones of the next layer.

Figure 5: Perceptron (Artificial Neural Network)



The processing of input data by each perceptron is done by two steps of calculation:

- compute weighted sum and
- pass the weighted sum through an activation function.

Step 1: Compute weighted sum

$$Z = b + \sum_{t=1}^{n} W_i \cdot X_i \tag{1}$$

- Inputs x_1 through x_n, are denoted by a vector X. X_i represents the ith entry from the data set. Each entry from the data set contains n dependent variables.
- Weights w_1 through w_n, are denoted as a matrix W. They represent how strong a connection is between a certain input and the perceptron, indicating this part of the input affects little or much the outcome of the perceptron.
- A bias term b, which is a constant.

Step 2: Activation Function

$$o = g(Z) \tag{2}$$

The output of step 1, Z, is passed through an activation function resulting in the output of perceptron o. The function g is a mathematical function that allows transformation of the outputs to a desired non-linear format before it is passed through the next layer. It essentially normalizes the summation result to a desired range which helps identifying whether the neuron needs to be "fired", meaning whether its output will affect the next layer.

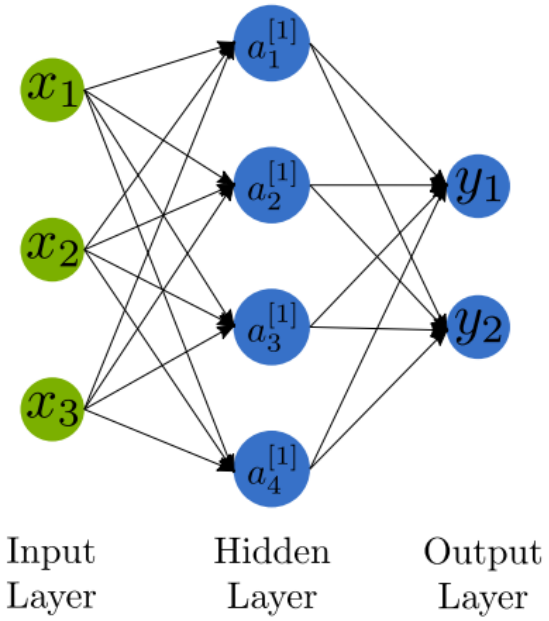
2.2.3 Shallow Neural Network

The most basic form of a neural network is the shallow neural network, which essentially consists of three layers: input layer, hidden layer and output layer. The input layer refers to the data sent by the input set. Hidden is the layer that consists of perceptrons each being connected by all of the inputs, dedicated in computations presented in the previous section. The last layer, the output layer, consists of a certain number of perceptrons (depending on the task), that determine the output of the network, in other words the prediction. A neural network with just one hidden layer is called shallow.

The direction in which the data is passed once through the network (usually presented from left to right) is called a forward propagation. After one forward pass is completed, the output layer compares its results to the actual ground truth labels and adjust the weights based on the differences between the ground truth and the predicted values. The process of a backward pass through the neural network is also known as back propagation.

Each layer consists of n_i neurons, i indicating the label of each layer, total i layers. Between two layers there are connections based on the number of neuron each layer consists of. For example, between layer l_1 and l_2 there are $n_1 \cdot n_2$ connections as each neuron of l_2 connects with every item of l_1. Usually this helps determine the load of the network, as it indicates the number trainable parameters there are,

Figure 6: Shallow Neural Network



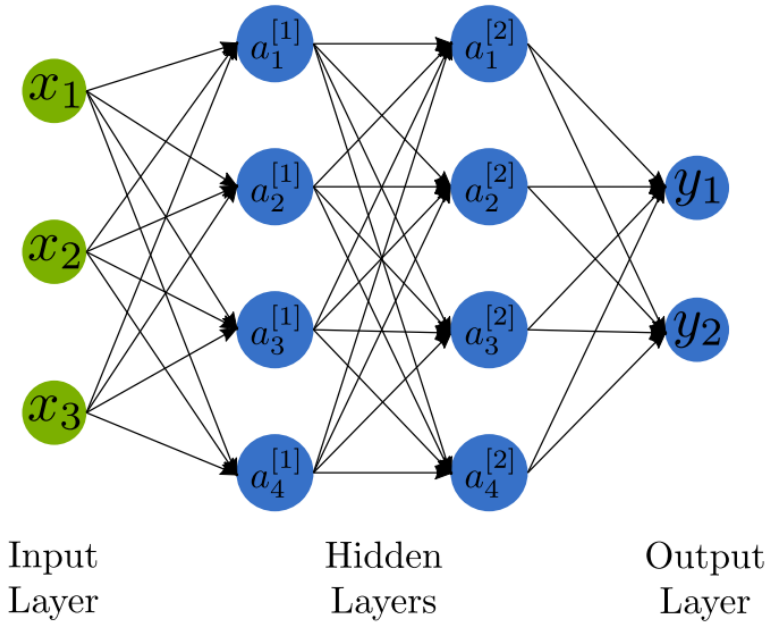
by the summation of all the connections. They are called trainable parameters, because the weights of each connection will be changed during back propagation based on the error of prediction.

2.2.4 Deep Neural Network

Deep Neural Networks resemble the architecture of a shallow neural network, with the difference being they consist of more than one hidden layer. As its name suggests, they have depth based on the number of extra hidden layers. Each neuron in a hidden layer is connected to all the others of the next layer. Selecting the number of hidden layers is determined by the nature of the task or size of data set.

The calculations and processes of forward pass or backward pass for each layer and neuron or perceptron remain the same as suggested in the previous sections.

Figure 7: Deep Neural Network



2.2.5 Back Propagation

The Back Propagation algorithm is used to effectively train a neural network through a method called chain rule. After each forward pass, the algorithm performs a backward pass while adjusting the trainable parameters (weights and biases). The algorithm is used to effectively train a neural network through a method called chain rule. In simple terms, after each forward pass through a network, backpropagation performs a backward pass while adjusting the model's parameters (weights and biases).

the basics of the process of back propagation follow these steps:

The network works to minimize an objective function, for example, the error incurred across all points in a data sample.

At the output layer, the network must calculate the total error (difference between actual and predicted values) for all data points and take its derivative with respect to weights at that layer. The derivative of error function with respect to weights

is called the gradient of that layer.

The weights for that layer are then updated based on the gradient. This update can be the gradient itself or a factor of it. This factor is known as the learning rate, and it controls how large the steps are that you take to change your weights.

The process is then repeated for one layer before it and continues until the first layer is reached.

During this process, values of gradients from previous layers can be reused, making the gradient computation efficient.

2.2.6 Convolutional Neural Networks

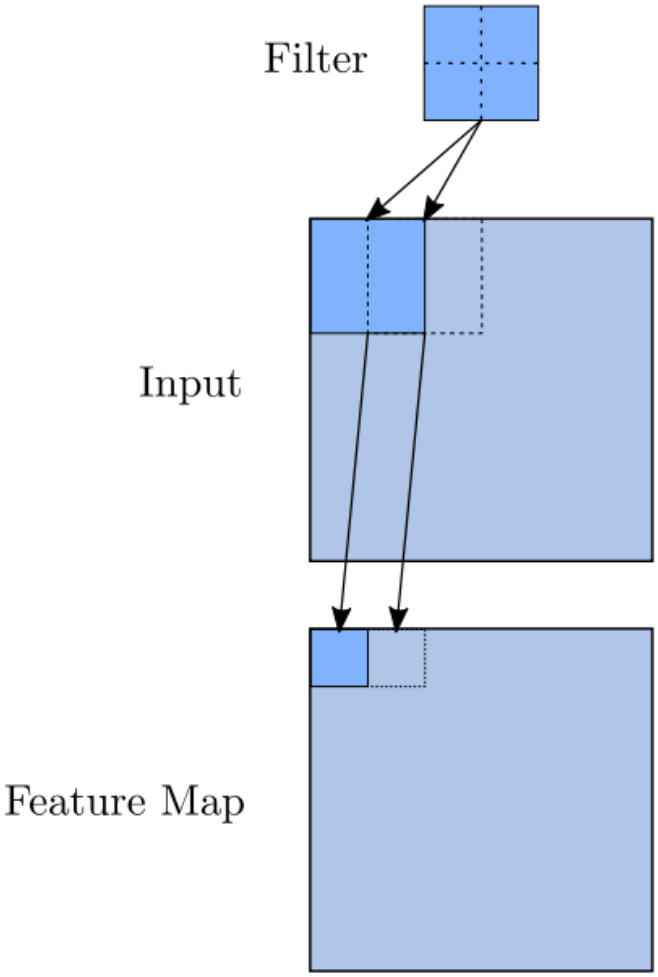
Convolutional layers (Brownlee 2019b) are the basis of creating convolutional neural networks. A convolution is an application of filters to an input data, resulting in a feature map, indicating the location of certain features in an input, such as an image.

Convolutional neural networks learn automatically a large number of filters specified to identify patterns and features among the training dataset, usually predicting such features in image data. This feature detection is usually applied in classification tasks. CNNs are specialized to work with two-dimensional image data most of the times, but the input can be one-dimensional and three-dimensional data.

The convolutional layer performs an operation called a "convolution", that is a linear operation that involves a multiplication of a set of weights with the input, like a typical neural layer. The main distinction is that the set of weights, also called filter or kernel, and the input data are both two-dimensional.

The operation between the data and the filter is a dot product. Each element of the filter is multiplied with each element of a filter-size patch of the input image, then those values are added resulting to a single value (this operation is also referred to as a "scalar product"). This process repeats for each filter-size patch of the input image, resulting in a set of dot products which is the feature map. This means that the filter size must be smaller than the input image, to allow detection in

Figure 8: Filter application



multiple regions in the image. The filter is applied using this operation from left to right and top to bottom of the image.

Convolutions have been a common technique used in computer vision. Filters were designed to be applied to images for feature map extraction. For example there are filters that are able to identify vertical or horizontal lines. Those filters are dragged through the whole input resulting in a feature map indicating the presence of vertical or horizontal lines. The application of both of those filters result in a feature map highlighting the edges of the input image.

Utilizing multiple filters offers a greater feature detection in image data. The use of multiple weighted filters that are able to be altered for desired feature extraction is what makes CNNs successful. The network will naturally learn what types of features to detect from the input dataset. Specifically, training under stochastic gradient descent, the network learns to extract features from the image that minimize the loss for the task the network is designed to solve, e.g. extract features that are most useful for distinction between images of cats and dogs.

Convolutional layers are often stacked allowing for a hierarchical decomposition of the input. The filters that operate directly on the raw pixel values will learn to extract low-level features, such as lines. The next set of layers are able to extract features that are combinations of lower-level features, such as ones that compromise multiple lines to express shapes. This process continues until the deepest layers are able to identify complex shapes such as faces, animals etc. This is the reason that CNNs are desired in solving tasks that often require complex feature detection.

3 Methods

In this section there will be presented the steps taken to achieve the goal of this work. The whole process is split in two sub-sections, first the preparation and preprocessing of the training data (MIDI files) is presented, and then the architecture of the training models (neural networks). The training part by itself contains two solutions, using a Variational Autoencoder (VAE) and a Generative Adversarial Network (GAN), for the purpose of research and comparison. Both of these models are trained with the same type of training data.

3.1 Preprocessing of training data

MIDI files are used broadly today in the music industry. If opened with certain applications there can be seen sequences of notes in a tabular form. The current work utilizes a python library called Mido (Bjørndalen 2013), which offers easy access to such files and their information, as well as the creation of MIDI files, which is handy for generation purposes. Aside from Mido, music21 (**music21_docs**) is also a great alternative to extract information from MIDI files.

By opening a .mid (.midi) file from any directory and assigning it to an object, this object contains a `tracks` attribute that is a list of tracks. Each track on its own is a list of messages and meta messages. Each message has a `time` attribute. Time is set to delta time (in ticks) which will be explained later in detail.

The following sections determine the type of a MIDI file:

- type 0 (single track): all messages are saved in one track
- type 1 (synchronous): all tracks start a the same time
- type 2 (asynchronous): each track is independent of the others

When creating a new file, the type can be selected by creation or can be changed anytime.

Also each Mido object contains a playback `length` item, the total musical sequence time in ticks. It should be noted that it's not possible to access the `length`

property of a type 2 file, as a `ValueError` will be raised, because the playback time of an asynchronous file cannot be computed.

3.1.1 Messages

As mentioned previously, each track is a list of messages. Those messages hold the information of the musical sequences. Most common messages are ones that press or release notes:

```
>>> mido.Message('note_on')
<message note_on channel=0 note=0 velocity=64 time=0>
```

Arguments can be passed like this:

```
>>> mido.Message('note_on', note=100, velocity=3, time=6.2)
Message('note_on', note=100, velocity=3, time=6.2)
```

Where the message type can be:

- 'note_on': a note is pressed
- 'note_off': a note is released

Note refers to all the possible notes as integers in range of 0 to 127. This data value d corresponds to frequency f in Hz by the MIDI Tuning Standard (MTS) which is a specification of precise musical pitch agreed to by the MIDI Manufacturers Association in the MIDI protocol. To get the frequency f of any data value d the following is used:

$$f = 2^{(d-69)/12} \cdot 440\,\text{Hz} \tag{3}$$

Inversely, the conversion from frequency f to midi note number d can be done with this formula:

$$d = 69 + 12\log_2\left(\frac{f}{440\,\text{Hz}}\right) \tag{4}$$

Velocity is the volume at which the note will be played (high value of velocity is loud and low value is silent).

As far as the time attribute is concerned, it is used as delta time in ticks. This essentially means that each message is accompanied by that attribute and determines how much time has passed since the previous message. In music there is also the beat which is essentially a quarter note. In a signature of a 4/4 song each measure has 4 beats. Each MIDI file has a value stored called `ticks_per_beat` which declares how many ticks a beat counts. The last time-related attribute that a track requires is the BPM (beats per minute) of a song (usually known in music as tempo, although in MIDI tempo refers to something else), and it counts how many beats a minute contains. For example a track that has 4 beats per minute and 3 ticks per beat is presented in 9:

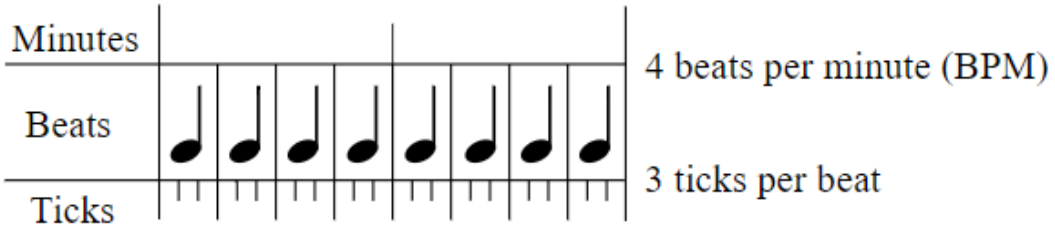


Figure 9: Example of a track with time attributes

Tempo in MIDI is not given as beats per minute, but rather in microseconds per beat. The default tempo is 500000 microseconds per beat, which is 120 beats per minute. There can be used some methods like `bpm2tempo()` and `tempo2bpm()` for conversions to and from beats per minute.

There are many other types of messages that refer to program, control, channel (used for parallel instruments and note effects), but for simplicity and due to the fact that the model is trained only on one instrument, the piano, the MIDI files are filtered and simplified to contain only 'note_on' and 'note_off' messages.

3.1.2 Meta Messages

Meta Messages behave like normal messages and can be created the usual way. They store information either related to the song or musician in general, like copyright information, text, track name, lyrics, or attributes that affect the track directly, like tempo, time signature.

Name	Keyword Arguments / Attributes
note_off	channel note velocity
note_on	channel note velocity
polytouch	channel note value
control_change	channel control value
program_change	channel program
aftertouch	channel value
pitchwheel	channel pitch
sysex	data
quarter_frame	frame_type frame_value
songpos	pos
song_select	song
tune_request	
clock	
start	
continue	
stop	
active_sensing	
reset	

Table 1: Supported Messages

Name	Valid Range	Default Value
channel	0..15	0
frame_type	0..7	0
frame_value	0..15	0
control	0..127	0
note	0..127	0
program	0..127	0
song	0..127	0
value	0..127	0
velocity	0..127	64
data	(0..127, 0..127, ...)	() (empty tuple)
pitch	-8192..8191	0
pos	0..16383	0
time	any integer or float	0

Table 2: Message parameter types

For the pre-processing of the training data, meta messages were not utilized in any way. Meta messages were implemented and inserted at the generation of the musical sequences, purely for presentation reasons.

3.1.3 Music to data

In this section it is presented how the MIDI files were processed into proper training data for the models. The generative models chosen for this task were VAE (Variational Autoencoder) and GAN (Generative Adversarial Network). Due to their nature of utilizing convolutional layers, translating MIDI files to images is the ideal approach. The goal of the pre-processing was to convert MIDI files, or message sequences, to images.

Message filtering

Firstly, as stated before, the messages need to be filtered in order to contain only message types of note pressing or releasing. This means that each message that is not type of "note_on" or "note_off" is discarded. Regardless, irrelevant typed messages are still processed and their time attribute is taken into account, otherwise the normalized track would not be the same.

The general idea is, each message gets parsed, if it's not type "note_on" or "note_off" its time attribute is saved to a variable and the message deleted from the sequence. That variable is utilized and its value is added to the last message that was saved. This basically means that each message of type "note_on" or "note_off" is kept as is, and the other types are deleted, but their time values are added to their previous messages, so the sequences remain the same. This filtering allows for an easier mapping of the musical sequence to an image. Also messages that are "note_on" with velocity value 0, that are "note_off" but there was not a previous "note_on" with the same "note" value are also filtered out. Some examples are presented below which indicate the application of such filtering.

```
"Filtering of a note_off with no prior note_on"
"Initial sequence:"
note_on         channel=0   note=74     velocity=81     time=0
note_off        channel=0   note=70     velocity=30     time=512
```

```
note_off        channel=0   note=74     velocity=0      time=256
...

"Filtered sequence:"
note_on         channel=0   note=74     velocity=81     time=0
note_off        channel=0   note=74     velocity=0      time=768
...
```

```
"Filtering of a note_on with velocity=0"
"Initial sequence:"
note_on         channel=0   note=74     velocity=81     time=0
note_on         channel=0   note=80     velocity=0      time=512
note_off        channel=0   note=74     velocity=0      time=256
...

"Filtered sequence:"
note_on         channel=0   note=74     velocity=81     time=0
note_off        channel=0   note=74     velocity=0      time=768
...
```

```
"Filtering anything besides note_on, note_off message types"
"Initial sequence:"
note_on         channel=0   note=74     velocity=81     time=0
control_change  channel=0   control=6   value=0          time=128
control_change  channel=0   control=101 value=0          time=128
set_tempo       tempo=800000                             time=128
set_tempo       tempo=850000                             time=128
note_off        channel=0   note=74     velocity=0      time=256
...

"Filtered sequence:"
note_on         channel=0   note=74     velocity=81     time=0
note_off        channel=0   note=74     velocity=0      time=768
...
```

Time normalization

Besides message filtering, while parsing through messages, their time attribute is reduced. This is essential due to the fact that each time tick will be treated as a pixel when converted to an image. Most MIDI files contain different "ticks_per_beat" values (which is how much ticks a beat counts, 9), and those values are high (for example: ticks_per_beat=1024) which would produce high scale images. It is impossible to find perfect MIDI files for this task, with small value of ticks_per_beat, due to the multiple range of applications or programs dedicated to creating digital music, plus the various musicians and creators use them. Each new time attribute is calculated given the file's "ticks_per_beat" and its value is set to 12 (the reason is given in the next section), in order to attain uniformity among the tracks.

$$new\_time = round\left(\frac{current\_time * 12}{ticks\_per\_beat}\right) \tag{5}$$

Message sequences to Images

Images prove essential for generational purposes using VAEs and GANs. The converted images have width and height set to 96 by 96 respectively. Each image's x axis refers to time and y axis refers to every possible piano pitch or note.

The ticks per beat is set to 12 for each track, so $\frac{96}{12} = 8$ quarter notes are contained to each image. In a 4/4 time signature (which is the most common case) this counts as 2 measures. The ticks per beat attribute is set to 12 to cover for most time signatures and speed of note presses. If the ticks per beat is set to 1, and a track contains eighth or sixteenth notes, then information is lost and the track gets corrupted. Speed exceeding sixteenth notes is rather rare so a ticks per beat set to 4 is acceptable, but some songs contain triplet notes (3 notes counted in a quarter) so $4 \cdot 3 = 12$ is the safest choice. This value essentially covers cases where information is not lost if the track contains sixteenth triples. Regardless, files can still be found with unexpectedly low message time values (grace notes) and by rounding, information can be lost.

The conversion process is done by creating an image of a whole song at first. An

empty list, or image, is initialized with dimensions 96 height and song_duration width with values set to 0 (black pixels). Also a dictionary is used to contain the active notes and each note is deleted when a note_off message with the same pitch value is presented. Each message is parsed and if it's type is note_on, it is added to the dictionary, plus its time value is saved to a message_time variable. For each item in the dictionary (active notes) pixels are set to 255 (white) depending on the message_time variable. After each note from the dictionary is assigned to the list (white pixel), if the message was type note_off, the note gets deleted from the dictionary, meaning it will not change the respective pixel to white in the next iteration.

For example note dictionary contains active notes: 45, 50, 55. Current message type is note_on, note=70 and time=12 and current time value is 120 (which is the pointer that points to the current x value of the image). The image is filled like this:

- image[96-45][120 to 120+12] from 0.0 to 255.0
- image[96-50][120 to 120+12] from 0.0 to 255.0
- image[96-55][120 to 120+12] from 0.0 to 255.0
- current_time=120+12
- note_dict=45, 50, 55, 70

The values of note 70 are not yet set to white but in the next iteration, the pixels will be swapped as it is present in the note dictionary.

A second example presents the process with a type note_off 50 message (current time=120, time=12 of message and same note dictionary):

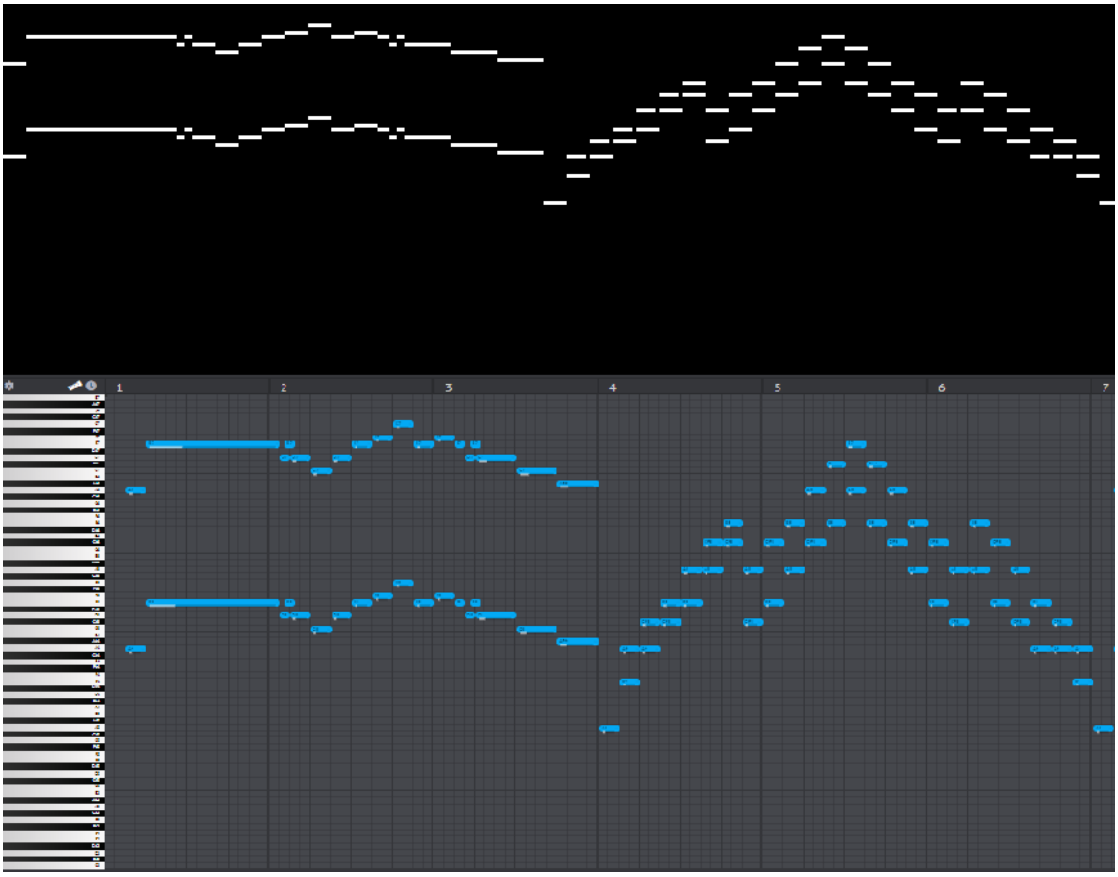
- image[96-45][120 to 120+12] from 0.0 to 255.0
- image[96-50][120 to 120+12] from 0.0 to 255.0
- image[96-55][120 to 120+12] from 0.0 to 255.0
- current_time=120+12

- note_dict=45, 55

Pixels for note 50 are changed, but the same note is discarded from the dictionary, and in the next iterations its pixels will remain black.

The converted images rows determine the note value (y axis) and columns refer to a specific time tick (x axis). It is noted that image's row value (pitch/note) is $96 - pitch$ purely for presentation purposes, otherwise it would be inverted, but the results would be the same.

Figure 10: Example of MIDI to image encoding compared to the same MIDI opened by external application



It is also noted that some files contain more than one tracks that partition the whole song (low and high notes or left and right hand), meaning the file type is synchronous (datasets do not contain asynchronous files), so the images created

for each track are then concatenated to one. After a whole song is converted to an image, it is then segmented to smaller images with width value equal to 96, so each file produces $track\_duration/96$ images.

3.2 Training Models

In this section the two approaches are presented for solving the music generation task. Two models were used, as stated previously, a Variational Autoencoder (VAE) and a Generative Adversarial Network (GAN). Both the models have powerful generative applications with different pros and cons. They are similar in their structure, as each one consists of two sub-networks, one dedicated to generating images and another assisting the generator network. Also they take advantage of convolutional layers which convert images to a more compact dense representation. After the training process is completed, the sub-network that translates data to images is used by passing random or specific input and it outputs images that are then converted to MIDI files. In the next sub-sections each approach, as well as the architectures used with their respective neural layers, are explained.

3.2.1 Variational Autoencoder (approach and architecture)

Variational Autoencoders are very well suited to generate random data that kind of resemble the initial dataset, but they work better at altering and exploring data at a specific non random direction (Shafkat 2018).

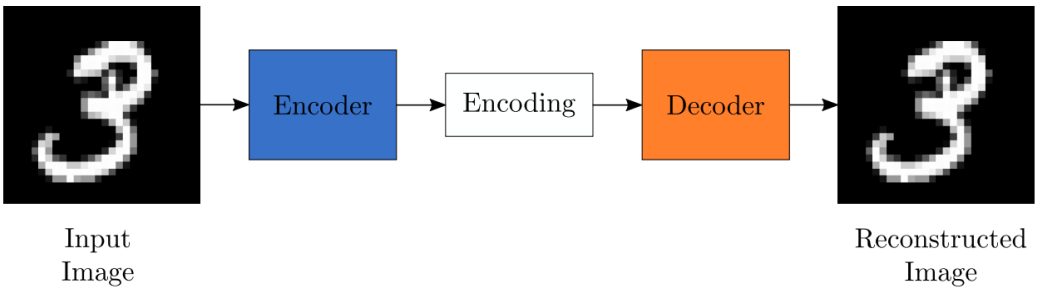
Standard Autoencoder

An Autoencoder consists of two connected networks, the encoder and the decoder. An encoder receives an input which is then translated to a small dense representation, which in turn is used by the decoder to convert it back to its original input.

The encoder is similar to Convolutional Neural Networks (CNNs). CNNs reduce a tensor of height-width-channel to a dense representation, for example a rank 1 tensor of size 1000. Then this representation is ready to get passed to the classifier. The only difference of an encoder is that it produces a representation

of a much smaller size, the encoding, enough to get passed to the rest of the network and attain a desired output. Usually, the encoder is trained with the rest of the network's parts, optimized by back propagation, to attain encodings desired for a certain task. In summary the general idea is making the encoder generate encodings used for reconstructing the initial input.

Figure 11: The Standard Autoencoder



The entire autoencoder network is trained as a whole and the loss function is either mean-squared error or cross-entropy between the output and the input, known as the reconstruction loss, which penalizes the network for creating outputs different from the input.

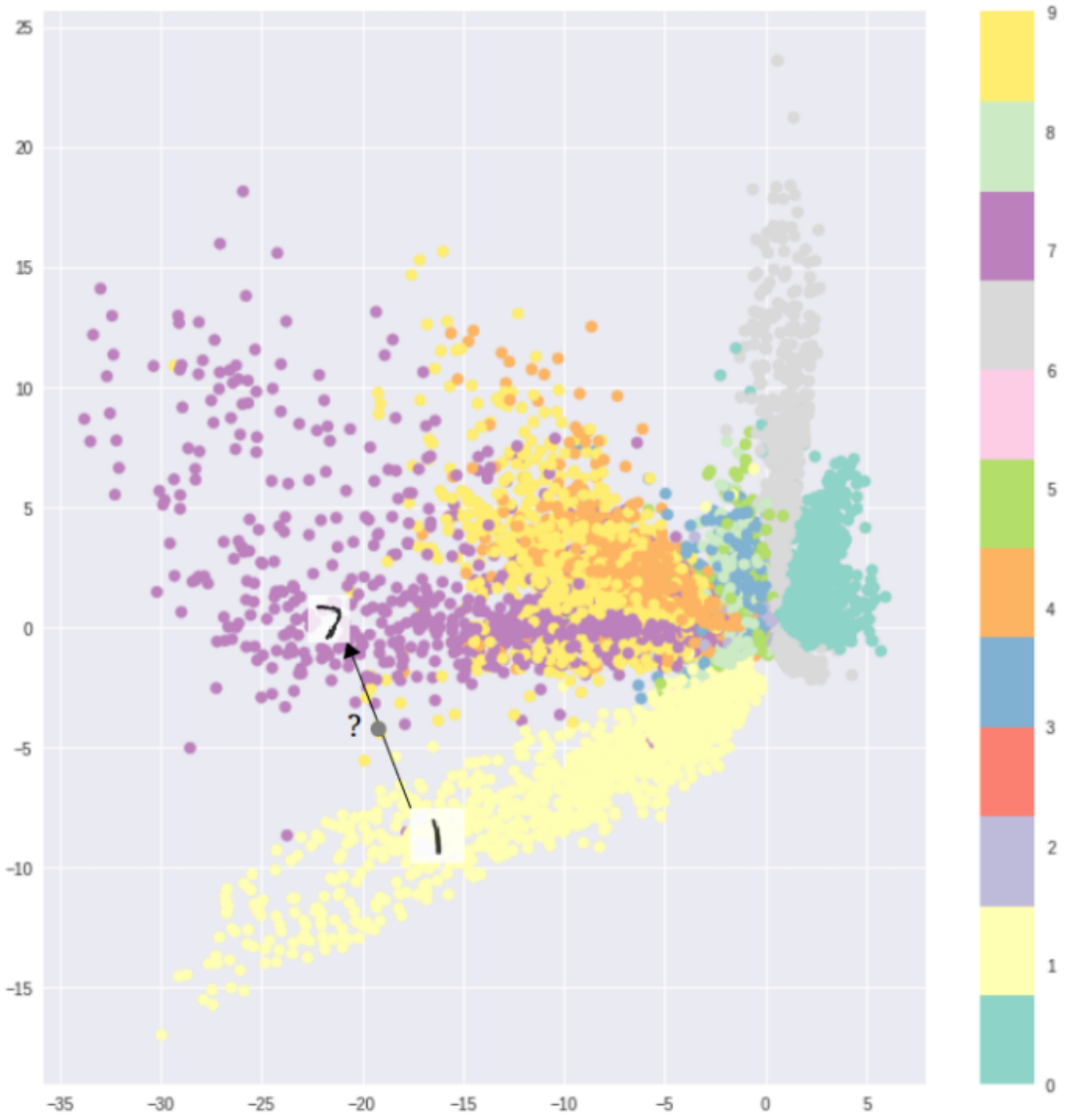
As the encoding (which is the output of the hidden layer of the encoder) is much smaller than the input, the encoder inevitably will discard information. The encoder learns to choose or preserve as much of the relevant information as possible in the limited encoding, and intelligently discards irrelevant parts. The decoder receives the encoding and is trained to properly reconstruct it into a full image. Together, they form an autoencoder.

Weakness of standard Autoencoders

Standard autoencoders learn to generate dense representations and then reconstruct their inputs fairly well. Nevertheless asides from a few applications like denoising autoencoders, their usage is fairly limited.

Their main issue, for generations, is the latent space that they convert their inputs to and where the encoded vectors lie, may not be continuous and can contain gaps.

Figure 12: MNIST Latent space optimizing only on reconstruction loss



For example a visualized 2D latent space (Figure 12) of a trained network to the MNIST dataset reveals certain distinct clusters of each class (Shafkat 2018). This is fair, as distinct encodings for each image type or class makes it far easier for

the decoder to decode them. This is fine for replication of the same images, yet it is not desired for generation purposes. The goal is to randomly sample from the latent space, or generate variations on an input image, from a continuous latent space.

If the space has discontinuities (eg. gaps between clusters), and the decoder samples encodings from those gaps, it will simply generate an unrealistic output, because it has no idea how to deal with that region of the latent space. This happens because during training, it never saw encoded vectors coming from that region of latent space.

Variational Autoencoders

Variational Autoencoders (VAEs) have one fundamentally unique property that distinguish them from standard autoencoders, and it is this property that makes them so useful for generative modeling: they are designed to produce latent spaces that are continuous, which allows for easy random sampling and interpolation.

This is achieved by outputting two vectors of size n instead of one: a vector of means, μ, and another vector of standard deviations, σ. They represent the parameters of a vector of random variables of length n, with the i-th element of μ and σ being the mean and standard deviation of the i-th random variable, X_i, from which the sampled encoding is obtained which in turn is passed onward to the decoder.

This stochastic generation means, that even for the same input, while the mean and standard deviations remain the same, the actual encoding will somewhat vary on every single pass simply due to sampling.

Intuitively the mean dictates the center of the position of an encoding, while the standard deviation controls the area of how much a sample can deviate from the mean or center. As encodings are generated randomly by the distribution, given the two vectors, the decoder learns that all nearby points close to the center refer to a certain class, and not just a single one in the latent space. This very property offers the desired exploration aspect as samples will vary, while still being part of the distribution, and in turn fill the discontinuous gaps. The latent space at which

the decoder is trained upon is denser with no discontinuities or gaps.

As long as the network offers encodings that their latent space is denser, it ideally provides the possibility of interpolation between classes, which the standard autoencoders are usually unable to do. However, since there are no limits on what values vectors μ and σ can take on, the encoder can learn to generate very different μ for different classes, clustering them apart, and minimize σ, making sure the encodings themselves do not vary much for the same sample (less uncertainty for the decoder). This allows the decoder to efficiently reconstruct the training data. In this work's case interpolation between classes is irrelevant as our approach utilizes a single class.

What is ideally desired are encodings, that are as close as possible to each other while maintaining distinction, allowing smooth interpolation, and enabling the construction of new samples.

This can be enforced by utilizing the Kullback–Leibler divergence in the loss function. The KL divergence between two probability distributions simply measures how much they diverge from each other, therefore by minimizing the KL divergence the probability distribution parameters (μ and σ) are optimized to closely resemble that of the given distribution.

$$\sum_{i=1}^{n} \sigma_i^2 + \mu_i^2 - \log(\sigma_i^2) - 1 \tag{6}$$

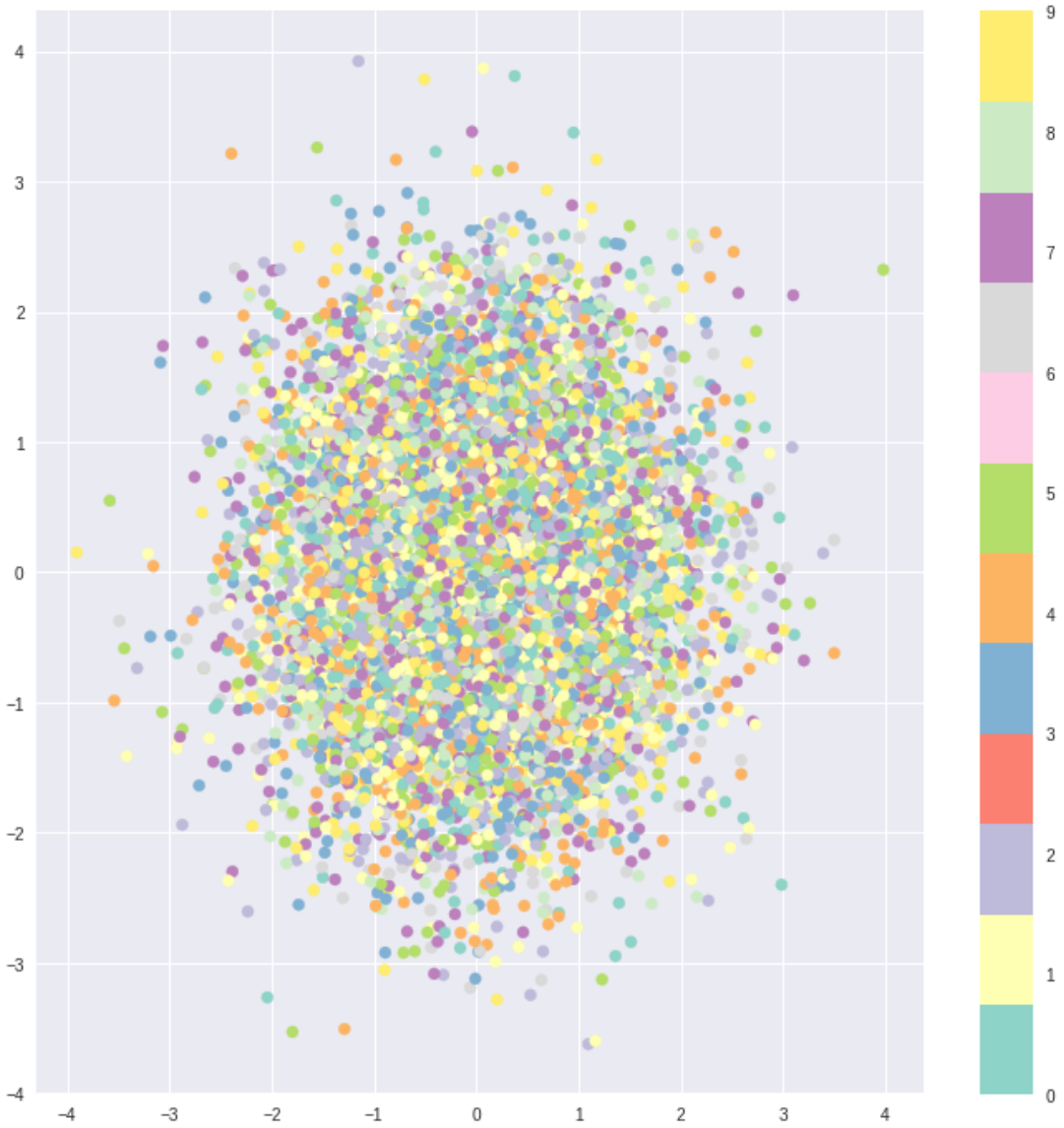
The KL loss (for VAEs) is computed by the sum of all the KL divergences between X_i $\mathcal{N}(\mu_i, \sigma_i^2)$ in X and the standard normal. This is minimized when $\mu_i = 0$, $\sigma_i = 1$.

Intuitively this forces the encoder to distribute all encodings from different classes to the center of the latent space. The encoder is also penalized if it tries to cluster apart the encodings.

Using purely KL loss, a latent space is attained (Figure 13) which its encodings appear densely randomly near the center regardless of similarities among nearby encodings (Shafkat 2018). Then the decoder is unable to distinguish anything

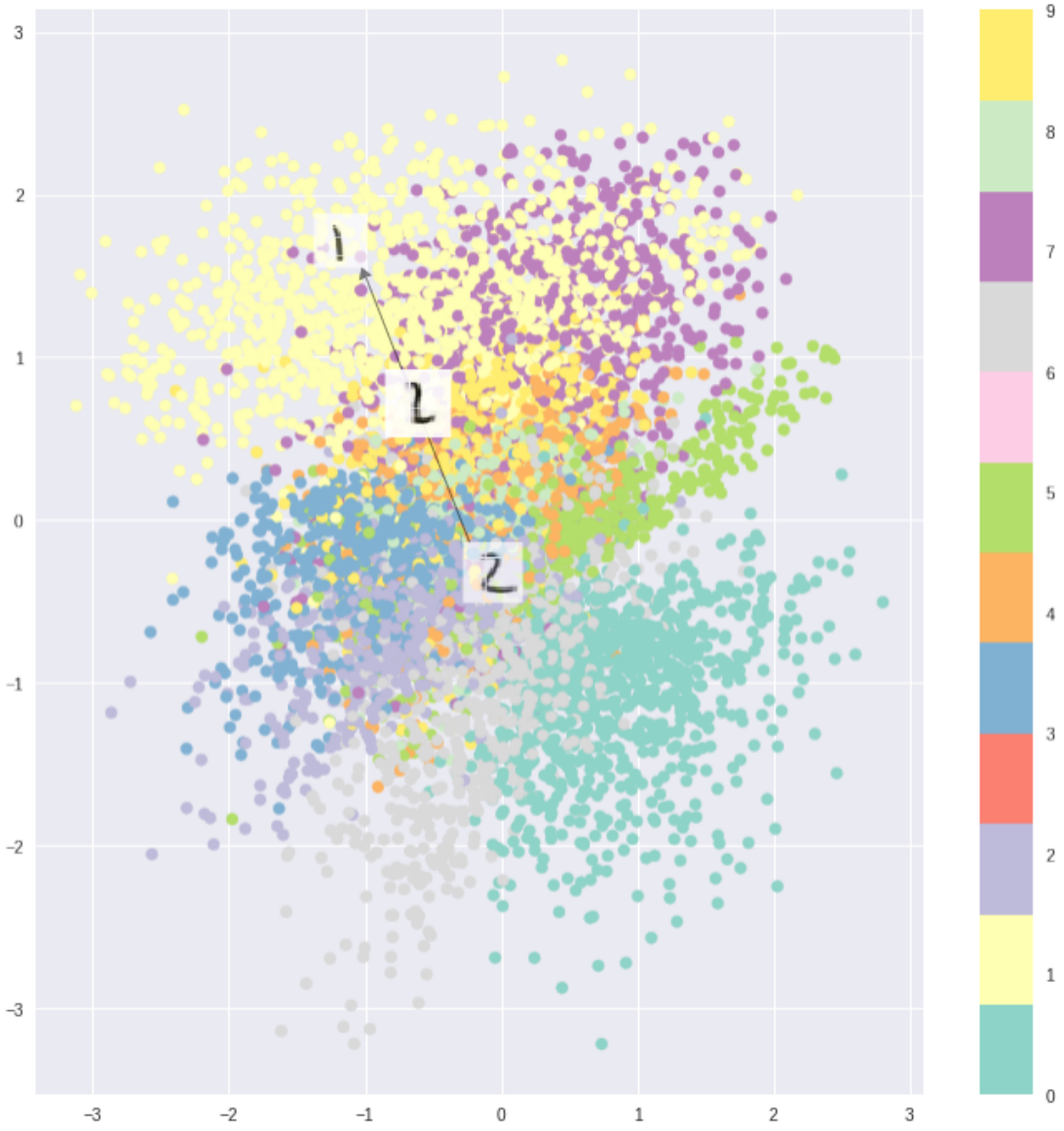
meaningful.

Figure 13: MNIST Latent space optimizing only on KL divergence loss



Optimizing the decoder as well (reconstruction loss along with KL divergence), results in the generation of a latent space (Figure 14) which maintains the similarity of nearby encodings on the local scale via clustering, and it is fairly dense packed near the latent space origin (Shafkat 2018).

Figure 14: MNIST Latent space optimizing on both reconstruction and KL divergence loss



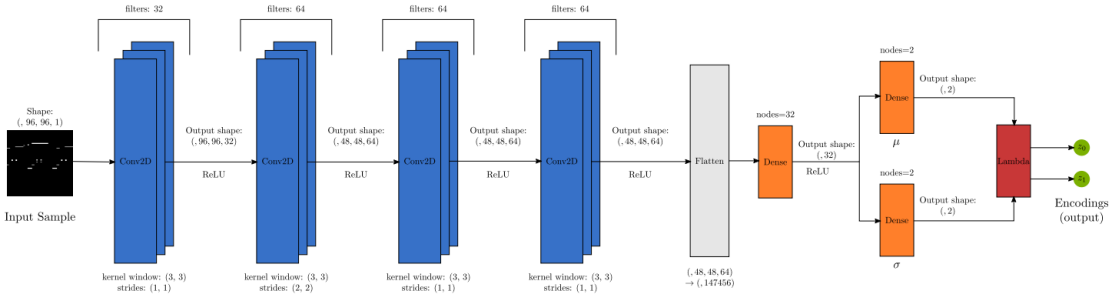
Intuitively the result of optimizing both reconstruction and KL loss, is the decoder can successfully decode encoded vectors from a certain distribution, as well as interpolate and mix features due to the lack of gaps in the latent space.

VAE architecture

In this section, the VAE architecture used for generation of images that contain musical sequenses is presented and explained. The layers of both networks (encoder and decoder) are also given in detail. This machine learning task is written in Python, which provides exceptional libraries and wrappers like Tensorflow and Keras.

Encoder Model Architecture

Figure 15: Encoder Architecutre



The encoder architecture is presented in Figure 15. This particular model receives an input image and translates it to a dense representation. The image data it receives is 96 by 96 by 1 pixel dimensions (height-width-channels), and the latent dimensions are 2, that represent the mean and standard deviation of the distribution.

All the neural layers of the encoder used:

- Conv2D: a regular convolution layer that executes a convolution through every pixel of the image, filters equate to the number of output channels (input has 32, rest of the layers have 64 filters), kernel size is 3x3 which is the window of the convolution, and the strides refer to the reduction of the output image (e.g. default of strides (1, 1) output image of same size, stride tuple set to (2, 2) outputs image of half height and width). Output of the convolutions are 96x96x1->96x96x32->48x48x64->two more convolutions of

same size

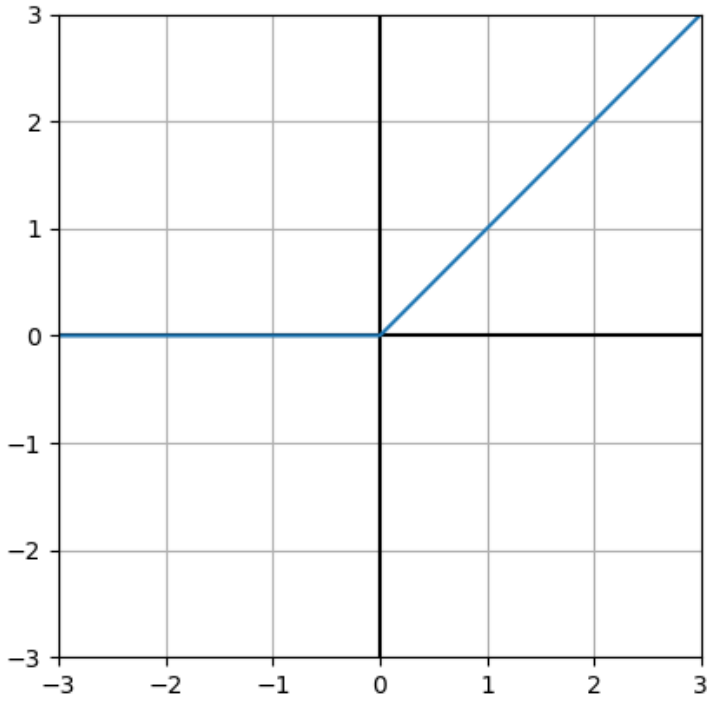
- relu: refers to the activation function of the layers, allows positives to pass through and filters out negatives to 0.
- Flatten: matrix conversion to a vector, in this case flattening the image of shape 48x48x64 to a vector of $48 \cdot 48 \cdot 64 = 147456$ items or network nodes. This is usually done as a preparation for the dense layer.
- Dense: a fully connected layer that has input times output connections, each connection has a weight and bias, that are multiplied and added to the input respectfully
- Lambda: this layer exists so that arbitrary expressions can be used as a network layer, in this case the computed expression contains μ and σ (mean and sigma) from the previous layers

Specifically this sub-network executes convolutions before feeding data through dense layers. It gets two outputs, mean μ and standard deviation σ of encoded input. We use these to sample random variables in latent space to which inputs are mapped. Then we define a sampling function to sample from the distribution. The sample is "reparameterized" based on the process defined by Gunderson and Huang into the shape of $mu + sigma \cdot eps$ where epsilon is the sample from the distribution. This is done to allow for gradient descent estimation accurately. The Lambda layer contains this exact expression added for gradient descent calculations using μ and σ.

The rectified linear unit (ReLU) (7) (Figure 16) is used as the main activation function. Activation functions such as ReLU are used to address the vanishing gradient problem in deep convolutional neural networks and promote sparse activations (e.g. lots of zero values). It is responsible for transforming the summed weighted input from the node into the activation of the node or output for that input. It essentially serves as filtering, if the input is positive it is passed as is, and if not the output is 0.

ReLU function:

Figure 16: ReLU



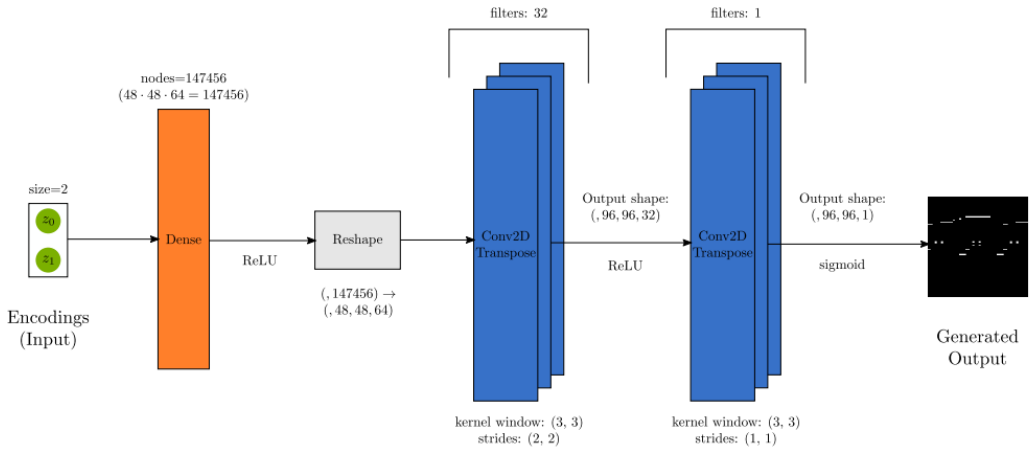
$$f(y) = \begin{cases} 0 & \text{if } y < 0 \\ y & \text{if } y \geq 0 \end{cases} \tag{7}$$

Decoder Model Architecture

The decoder architecture is presented in Figure 17. This model translates the output of the encoder, which is a sampled latent vector, to an image of shape 96x96x1.

- Dense: a fully connected layer, contains input times output connections, each connection has a weight and bias which are multiplied and added to the input respectfully. Input layer of the decoder network.
- Reshape: reshapes the input to an output with desired dimensions (error

Figure 17: Decoder Architecutre



will be raised if the total number of items between input and output are not equal), eg. in this case the input from the dense layer is 147456 nodes (or items) and the reshaped output is a 48 by 48 by 64 matrix, which $48 \cdot 48 \cdot 64 = 147456$

- Conv2DTranspose: a transpose convolution or deconvolution layer, used for upsampling the input determined by the strides of the convolution, which in this case strides are set to a (2, 2) tuple which essentially doubles the x and y axis of the input (width and height)
- relu: an activation function which simply allows positives to pass through and filters out negatives to 0
- sigmoid: an activation function also known as logistic regression, which is essentially a smooth step function around 0.5

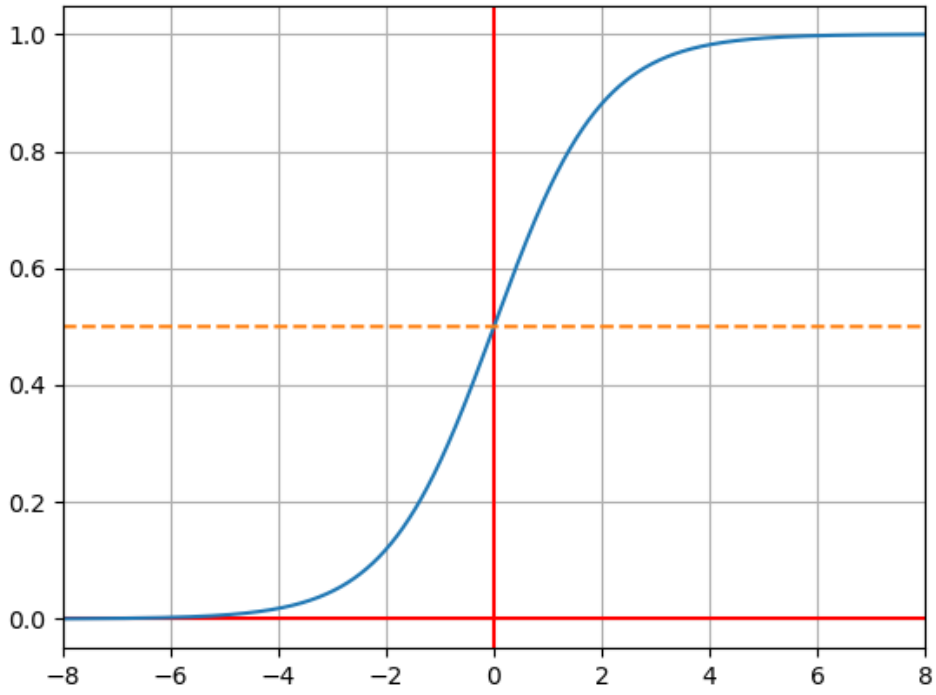
The decoder is the component that will be used after training for generating images. The main goal is to essentially train the combined model (encoder plus decoder) in order to achieve a decoder that can sample data from the target distribution so the image generation is as close to the desired results as possible.

The sigmoid function (8) (Figure 18), or logistic regression, is applied to the output

of the decoder network. This function adds non-linearity in machine learning modeling, by mapping real values between 0 and 1. It is usually added to networks that perform binary classification tasks, and in this case each pixel is mapped between 0 and 1, which determines whether each pixel (or note) will be "played" (in musical terms) or not.

$$f(y) = \frac{1}{1 + e^{-y}} \tag{8}$$

Figure 18: Sigmoid function



The combined model

The two models, the encoder and decoder, are then combined with the help of a custom layer class, which is essential if we want to define custom loss. The complete VAE is trained using two loss functions, reconstruction loss and KL divergence.

Reconstruction loss refers to the binary cross-entropy loss between the input image (from the initial dataset) and the decoded image (provided by the decoder). Binary cross-entropy compares each of the predicted probabilities, in this case for each pixel, to the actual output which can be either 0 or 1 (black or white pixel, note off or note on). It then calculates a score that penalizes the probabilities based on the distance from the expected value (how close or far).

The loss-function of binary cross-entropy:

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i)) \tag{9}$$

Kullback-Leibler (KL) divergence is used to attain sampled encodings that are as close as possible to each other while maintaining distinction, allowing smooth interpolation, and enabling the construction of new samples. The KL loss is computed as shown in (6) and then added to the reconstruction loss.

Compilations and Optimizers

Compilations and optimizers of neural network models often refer to the adjustment of trainable parameters, the weights. They are essential processes that often determine the stability of the network and offer a smooth learning curve.

The combined model, encoder along with decoder, are compiled with the custom loss, reconstruction and KL loss. The optimizer of choice is Adam.

Adam is the optimization algorithm used instead of a classic stochastic gradient descent for weight updating. It's name is derived from adaptive moment estimation. Classic stochastic gradient descent has a single term named alpha (learning rate) for updating all the weights and remains unchanged during the training process. Adam utilizes a learning rate as well, that is separately adapted for each network weight as learning unfolds. Adam is produced by combining the advantages of two other extensions of stochastic gradient descent, specifically:

- **Adaptize Gradient Algorithm** (AdaGrad) that maintains a per-parameter learning rate that best performs on problems with sparse gradients (e.g. NLP

and computer vision)

- **Root Mean Square Propagation** (RMSPror) that also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e.g. how quickly it is changing), meaning the algorithm does well on online and non-stationary problems (e.g. noisy).

Adam utilizes the advantages of both AdaGrad and RMSProp. Instead of adapting the parameter learning rates based on the average first moment (the mean) as in RMSProp, Adam also makes use of the average of the second moments of the gradients (the uncentered variance). Specifically, it calculates an explonential moving average of the gradient and the squared gradient, and parameters beta1 and beta2 control the decay rates of these moving averages. Adam is preferred between other optimizers in similar tasks because it is effective. Empirical results demonstrate the superiority of Adam in terms of training cost among other optimizers.

!!!placeholder, maybe figure with adam opt compared to other opts

Adam configuration parameters with their respective empirical good default examples:

- **alpha** also known as learning rate or step size. The proportion that weights are updated (e.g. empirical good default 0.001). Larger values results in faster initial learning before the rate is updated and smaller values in slow learning.
- **beta1**. The exponential decay rate for the first-moment estimates (e.g. 0.9).
- **beta2**. The exponential decay rate for the second-moment estimates (e.g. 0.999). This value should be set close to 1.0 on problems with sparse gradient (e.g. NLP and computer vision).
- **epsilon**. A very small number to prevent any division by zero in the implementation (e.g. 10E-8).

3.2.2 Deep Convolutional Generative Adversarial Network Approach and Architecture

Generative Adversarial Networks are broadly used for generative purposes. Generative modeling is an unsupervised learning task in machine learning that involves automatically identifying and learning the similarities or patterns in input data in such a way that the model can be used to generate or output new examples that plausibly could have been drawn from the original dataset (Brownlee 2019a). They are desired and sought after due to their interesting design of one of their sub-networks, the generator.

The generator is very interesting due to its nature that translates random noise into desired data. This is accomplished via the second sub-network the discriminator which receives the generator's output and decides whether this data is true or false. Due to parallel training those two networks compete, in order to improve themselves. The generator's purpose is to fool the discriminator and the latter is trained to distinguish real from fake images, that originate from the dataset or the generator. This results in a zero-sum game, adversarial, and the ideal situation in theory would be the discriminator is fooled about half the time, meaning the generator attains plausible examples. In practice the expected accuracy of a discriminator from a well trained GAN hovers around 80%, and this is explained in detail in Results and Discussion sections.

A typical machine learning task centers around a model which tries to make a prediction, e.g. predictive modeling.

The whole process revolves around a training dataset which is used to train a model, the dataset is comprised of multiple examples called samples, each with input variables (X) and output class labels (y). A model is trained given the inputs, it then predicts an output, and is penalized or corrected when the outputs do not match the desired result. This correction refers to the term of supervised learning.

Typical supervised learning problems include, for example, classification and regression, and supervised learning algorithms include logistic regression and random

forest.

The other main type of machine learning is the descriptive or unsupervised approach. In unsupervised learning there is no output, or labels, but the model is only given inputs, and its goal is to find "interesting patterns" in the data. This is kind of an abstract problem, as those patters are unknown, and the correction or metric, that is required for the learning process, is not straightforward (unlike in supervised learning, where the prediction of y for a given x can be compared to the observed value). This lack of correction refers to the term of unsupervised learning.

Discriminative vs. Generative Modeling

Usually, in supervised learning, models are constructed for predictive purposes, to classify an example of input variables to a specific class. This predictive modeling task is called classification. It is also known as discriminative modeling. From the training data, the goal is to attain a discriminant function $f(x)$ that maps each x onto a class label, therefore combining the inference and decision stages into a single learning problem. In summary such a model's goal is to discriminate or classify sets of input variables across classes. The factor of choice or decision, as to what class a given example belongs, is presented.

On the other hand, unsupervised models that identify the patterns of input variables are able to be used for generative purposes based on the input distribution. Those models are referred to as generative models.

For example, a single input variable may have a known data distribution, such as a Gaussian distribution, or bell shape. A generative model may be able to identify patterns and summarize this distribution, as well as generate new variables that could have been part of the initial dataset. The ideal scenario would be a generative model that produces data that is indistinguishable from real samples.

Standard GANs

The GAN model architecture consists of two sub-models, a generator model capable of producing new examples and a discriminator model for identifying whether input data originates from the domain (is real), or is fabricated by the gener-

ator model (is fake). GANs are based on a game theoretic scenario in which the generator competes against an adversary.

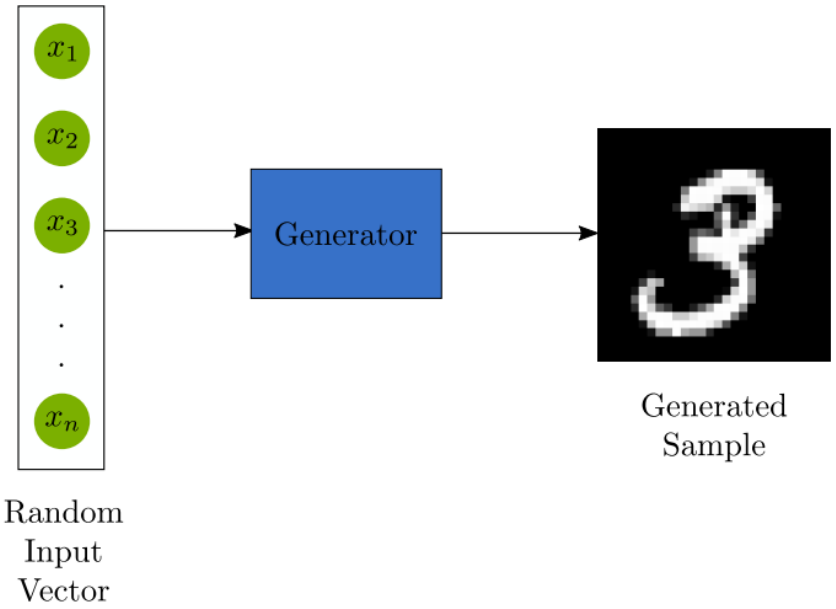
Generator Model

The generator model receives as input a fixed-length random vector and generates a sample in the domain (Figure 19).

The vector is produced randomly by a Gaussian distribution, and it is used to seed the generative process. After the training process, points in this multidimensional vector space will correlate with points in the problem domain, forming a dense representation of the data distribution.

The vector space is also known as latent space. It is a vector space which contains latent variables. Those latent or hidden variables are important in neural network modeling, they are understood by the specific model, but are not directly observable by humans. Furthermore latent variables are often referred to as a projection or compression of a data distribution. In this specific case of GANs, the generator model learns to apply meaning to points in a chosen latent space, such that when new points are drawn from it, can be provided to the generator model as input and used to generate new and different output examples. When the process of training is deemed complete, the generator is saved and used to generate new samples, hence the point of this work.

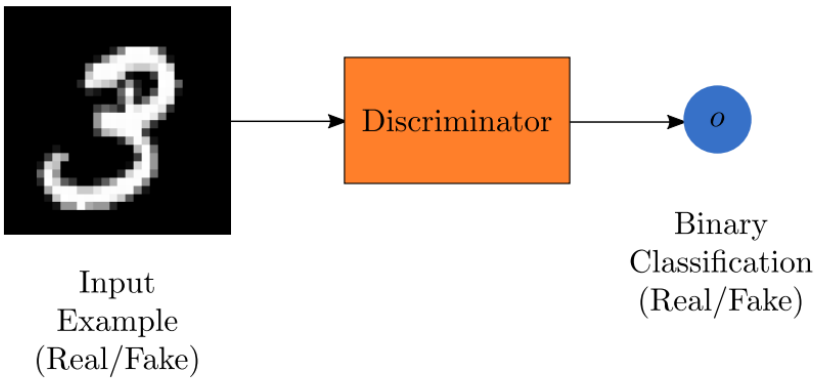
Figure 19: The Generator Model



Discriminator Model

The discriminator model receives as input an example from the dataset (real) or a generated sample (fake) and predicts a binary class label of real or fake (Figure 20). The real examples are drawn from the original dataset and the fake ones are fabricated by the generator model. The discriminator is a typical classification model, which in this case classifies images to be true or fake, meaning they originate from the initial dataset or they are produced by the generator. After the training of the GAN is completed, the discriminator is discarded, as the model of interest is the generator purely for generative purposes. Regardless, it could still be used for other tasks as some of the feature extraction layers can be used in transfer learning applications.

Figure 20: The Discriminator Model



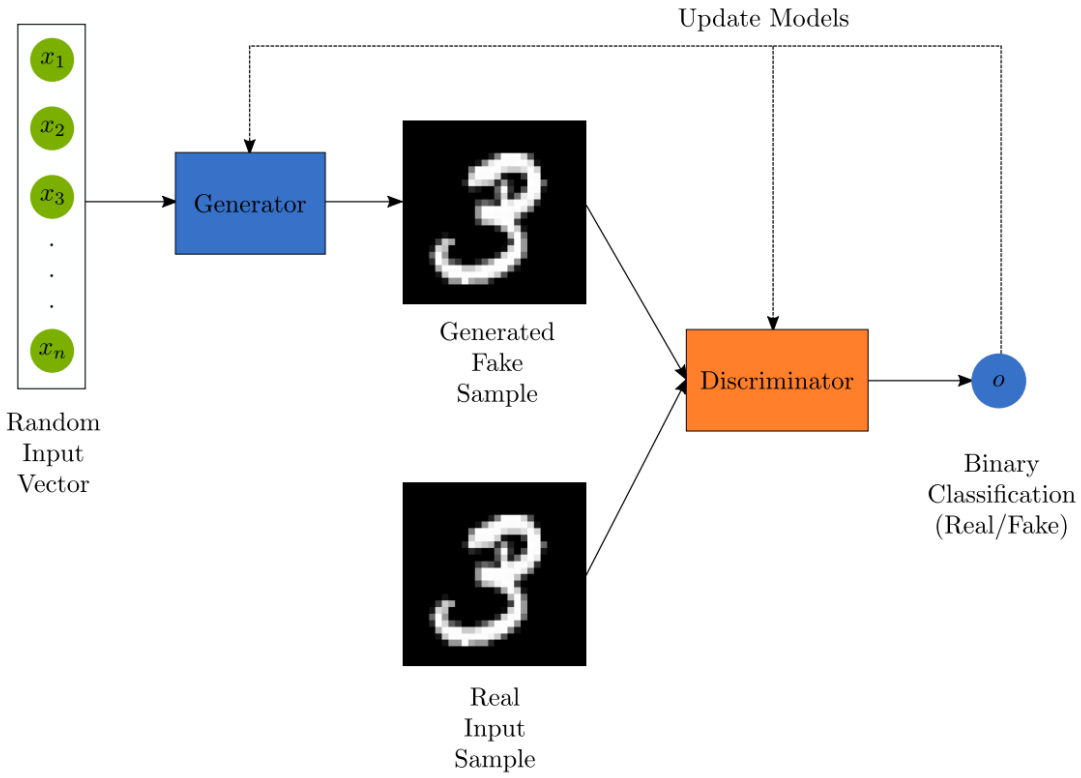
GAN - A two player competition

As mentioned in a previous section, generative modeling is an unsupervised learning problem, but the training of the generative model in GANs is framed as a supervised learning problem.

The two models, generator and discriminator, are trained together (Figure 21). The generator generates samples trying to imitate the dataset, and these, along with the real examples, are received by the discriminator and classified as real or fake.

Then the discriminator is updated to get better at discriminating real and fake samples in the next round, and the generator is updated based on how well the generated samples imitated the ones from the dataset and fooled the discriminator.

Figure 21: The Generative Adversarial Network Architecture



GANs and Convolutional Neural Networks

At most applications GANs utilize the use of convolutional neural networks or CNNs. Both the discriminator and generator consist of such layers.

CNNs are very useful in computer vision tasks as they are able to compress an image of 3 dimensions (width, height and channels, eg. 64x64x3) to a vector. This vector is a dense representation of the input that extracts its main features. They are used for the generator to translate the latent space to an image example, and for the discriminator to identify the validity of the image, whether it is real (from the dataset) or fake (produced by the generator).

Modeling image data refers to that exact process of creating a latent space, the input of the generator, that provides a dense or compressed representation of the set of images from the dataset used to train the model. The generator learns from training to receive random latent points from a distribution and generate new images.

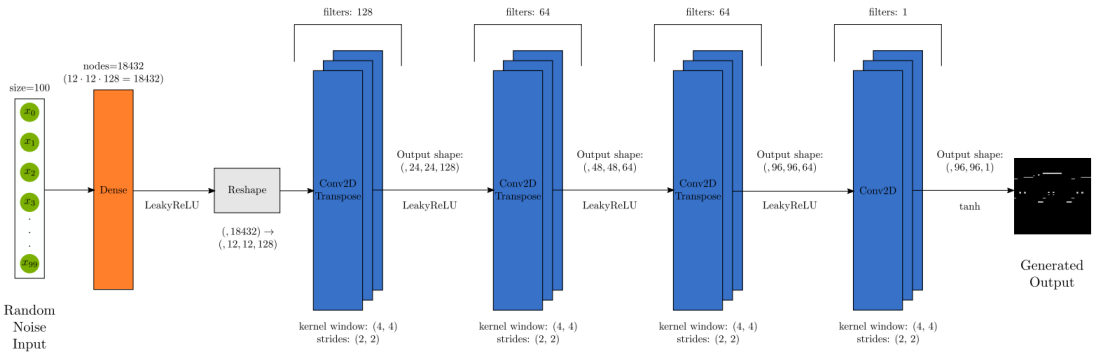
CNNs are also applied to this current task, as convolutions and transpose convolutions are essential. The discriminator model utilizes convolutions to break down an image input to a compressed representation and classify whether it's real or fake, and the generator model uses transpose convolutions or deconvolutions which is the opposite direction of a convolution, transformation of a supposed compressed representation (random points from a distribution) to an image, that still maintains connectivity and patterns as a regular convolution.

GAN Architecture for Music Generation

In this section, the architecture used for generation of images that contain musical sequenses is presented and explained. The layers of both networks (generator and discriminator) are also given in detail. This machine learning task is written in Python, which provides exceptional libraries and wrappers like Tensorflow and Keras.

Generator Model Architecture

Figure 22: The Generator Architecture



The generator architecture is presented in Figure 22. The model receives an input

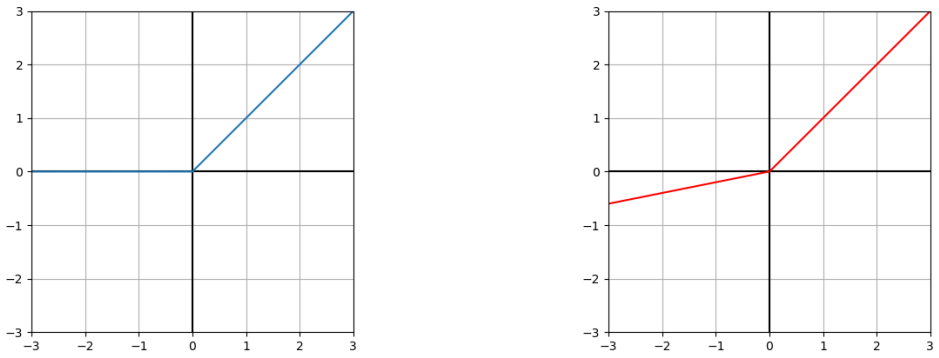
of latent points with dimension 1 by 100. This noisy information is then passed through a fully connected or dense layer, that has $24 \cdot 24 \cdot 32 = 18432$ nodes. This exact number of nodes is essential as the desired output of the model will be 96x96x1 dimensions. Then a number of upsamples take place, as the use of deconvolutional layers take place for the transformation of the output of the dense layer to an image.

All the neural layers used:

- Dense: a fully connected layer, contains input times output connections, and each connection has a weight and bias, that is multiplied and added to the input respectfully. This is used to convert the latent vector (noisy input) to an item that is then ready to be upsampled to an image with desired dimensions
- Reshape: simply reshapes the input to an output with desired dimensions (error will be raised if the total number of items between input and output are not equal), eg. in this case the output of the dense layer is 18432 nodes that is then reshaped to an output of 24x24x32 (width-height-channels), that $24 \cdot 24 \cdot 32 = 18432$.
- Conv2DTranspose: a transpose convolution or deconvolution layer, used for upsampling the input determined by the strides of the convolution, which in this case strides are set to a (2, 2) tuple, which essentialy doubles the x and y axis of the input. Also the kernel window size is 4x4 at which the convolution is calculated. Filters sizes vary.
- LeakyReLU: serves as the main activation function, a *leaky* rectified linear unit
- Conv2D: a regular convolution layer that shifts through every pixel and executes a convolution, height and width are unchanged as strides default to (1, 1), and serves are the output of the generator model
- tanh: activation function of the final Conv2D layer

The output of the reshaped dense layer is passed through two upsampling lay-

ers (Conv2DTranspose). While training, generator models with smaller number of dense nodes and more upsampling layers produced images with all the pixels valued 0 (black images), which makes sense since the more dense nodes, the more information the model is able to save and imitate the dataset. Nevertheless the training process becomes slower as the total trainable parameters increase, as well as the load is heavier for the machine used for training.



(a) ReLU (b) LeakyReLU

Figure 23: Rectified Linear Units.

The rectified linear unit (ReLU) (7) (Figure 16) is used as the main activation function. Basic ReLU is used in VAE iteration referenced in the previous section whilst the GAN utilizes LeakyReLU.

LeakyReLU (10) serves the same purpose as basic ReLU, but instead of converting every negative value to 0, it allows some negative values based on the slope variable that is set by the user, hence the name "leaky". While in the generator, regular ReLU performed just as well as LeakyReLU, in the discriminator the deviation of the results was greater, so the leaky rectified linear unit was kept as default for both of the models with a slope of value a=0.2 (example shown in Figure 23).

LeakyReLU function:

$$f(y) = \begin{cases} a * y & \text{if } y < 0 \\ y & \text{if } y \geq 0 \end{cases} \tag{10}$$

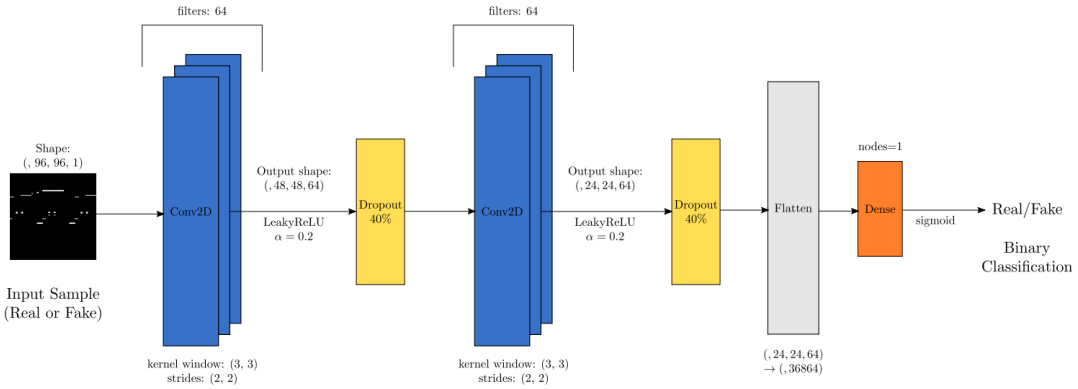
The deconvolutions in the generator model were preferred compared to other

upsampling layers (Upsampling2D). Strided convolutions allow the generator to learn its own spatial upsampling. In this case the generator upsamples the reshaped dense representation twice, resulting in the desired dimensions of the images, paired with LeakyReLU. Also the filters were set to 64 which were enough to sustain features, and the kernel size to (4, 4) (which is the convolution window). Furthermore the input upsamples from 24x24x32 to 48x48x64 and then to 96x96x64 to attain the desired height and width. The last layer, or the output layer of the model, is a regular convolution that converts the filters or channels to 1 (since the model is trained upon and generates images in grayscale) with a training function of tanh (output is normalized to range of [-1, 1]), a kernel window (7, 7) and with no strides, since the dimensions of the image match the desired result. Most GANs utilize a regular convolution layer instead of a fully connected dense layer, and by practice this output layer massively improved the results. Other parameters that refer to bias and kernel constraints or initial values were left to default.

It is noted that the generator model is compiled with an optimizer when combined with the discriminator model.

Discriminator Model Architecture

Figure 24: The Discriminator Architecture



The architecture of the discriminator as shown in Figure 24 resembles a typical binary classification model using CNNs. As stated before, the process is to feed

the discriminator with images of shape 96x96x1, either sampled by the dataset or generated by the generator model, and determine whether they are real or fake.

The neural layers used by the discriminator model:

- Conv2D: regular convolution layer, with 64 filters, kernel window size is 3x3 at which the convolution is calculated, and the strides are set to (2, 2) as they determine the size of the output image (strides of size 2 reduce the image in half) eg. output of the convolutions are 96x96x1->48x48x64 and then 48x48x64->24x24x64
- LeakyReLU: activation function that lets all positive values to pass through and allows some negatives (regular ReLU filters all negative values to 0) based on the value of the slope set to alpha=0.2
- Dropout: randomly sets input values to 0 based on the frequency or rate set by the user, in this case, rate is 0.4 which means 4 out of 10 neurons will be terminated randomly. This helps with the overfitting problem of neural network classification and it is noted that dropout only takes place during training. Also inputs not set to 0 are scaled up by 1/(1 - rate) such that the sum over all inputs is unchanged.
- Flatten: converts a matrix to a vector, in this case flattening the image of shape 24x24x64 to a vector of $24 \cdot 24 \cdot 64 = 36864$ items. This is usually done for preparation to the dense output layer.
- Dense: a fully connected layer that has input times output connections, in this case $36864 \cdot 1 = 36864$ connections each one weighted and biased calculating the outcome, which is a binary classification. The activation function used is sigmoid which is the most common and best performing for binary classification tasks.

The general idea of the discriminator is to catch the generator and identify fake images among real ones from the dataset. This model uses two downsampling convolution layers, each with the same number of filters, strides and kernel windows, to break down its input, and a dense layer as the output for classification.

The activation function used, sigmoid, also called the logistic function, for binary classification problems is like a step function around 0.5 (if y>0.5 f(y)=1, if y$\leq$ 0.5 f(y)=0), but smoother, allowing values between 0 and 1 to pass through. Sigmoid is also presented in the VAE iteration (8) (Figure 18)

Compilation and Optimizers

Compilations and optimizers of neural network models often refer to the adjustment of trainable parameters, the weights. They are essential processes that often determine the stability of the network and offer a smooth learning curve.

While the compilation of the generator takes place after the two models are combined, the discriminator is compiled separately. More specifically the discriminator is compiled with binary cross-entropy as its loss, its performance is observed by the accuracy metric and it utilizes an Adam optimizer.

Binary cross-entropy compares each of the predicted probabilities to actual class output which can be either 0 or 1. It then calculates the score that penalizes the probabilities based on the distance from the expected value. That means how close or far from the actual value.

The loss-function of binary cross-entropy:

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i)) \tag{11}$$

Where y is the label (1 for real images and 0 for fake images) and $p(y)$ the predicted probability of the input being real for all N samples. This means that for each real image originating from the dataset ($y = 1$) it adds $log(p(y))$ to the loss, which is the log probability of it being real, and conversely it adds $log(1 - p(y))$ for each fake image generated by the generator ($y = 0$), which is the log probability of it being fake.

The metric of accuracy is fairly straightforward, as it provides a percentage of how many times the discriminator was correct at its prediction. In GANs usually this metric fluctuates in the beginning and ideally converges around 80% at which

point the discriminator is fooled by the generator, indicating a desirable generator model. More details for accuracy are presented in Results and Discussion sections.

Adam is the optimization algorithm used instead of a classic stochastic gradient descent for weight updating. Adam is explained in detail in the VAE iteration referenced in the previous section.

4 Results

In this section we present the outcome of the generative modeling we approached as shown in Methods section. Both the VAE and GAN were trained in three different datasets.

- Classical MIDI Collection: 295 tracks converted to 16400 images, all of the classical genre, with great variance on composition (based on different composers from different eras and cultures)
- Undertale video game soundtrack: 110 tracks converted to 3207 images, video game music, fair variance between tracks
- Pokemon Blue/Red Rescue Team video game soundtrack: 11 tracks converted to 298 images, ambient music with little variance between tracks

By the term variance between tracks we refer to the difference between melodies and sounds, e.g. in the Pokemon soundtrack, the tracks have all fairly similar ambient rhythm, in the Undertale dataset some tracks are similar of the ambient type but there is also presence of faster rock type tracks, and in the Classical dataset the difference is greater due to different styles of composers included. Furthermore it should be interesting to determine whether that factor affects network performance.

Training two models in three different datasets (six different iterations) is believed to be enough in order to infer results, plus we get the option of comparing the models with themselves.

In an ideal situation we would want each model to generate images (that are converted to MIDI tracks) which closely resemble the ones from the datasets with the addition of exploration and originality. In practice this is not the case as the results are not melodic enough or consumer grade. This work focuses on extracting information as efficient as possible (due to the limited training time and resources) and obtaining a model that generates images as close to the desired result as possible. This distinction is important though, as good quality images, with little noise, does not necessarily mean that their conversion to music is going

to be melodic.

Furthermore this section is split between two parts, each for every model presented in the methods section. Both of the models, VAE and GAN will be analyzed based on metrics, losses, and on some statistical analysis that can help explain the nature of the models or pose questions that will be addressed in the Discussion section.

In general metrics such as loss can sometimes be misleading. In a generative task the loss between epochs is expected to behave in certain ways, nevertheless such a task is subjective and the results cannot be based solely on that factor.

The statistical analysis presented simply measures the probability of each note in each initial dataset. Then the same analysis is applied on a set of generated images from the models and are then compared. The idea is if the behavior matches, the respective model has a tendency to generate results close to the desired ones.

In conclusion good results in such a task are subjective, and good sound or melodies may vary between even humans. In practice the outcome of this work does not generate consumer-quality music, although there are some notable results.

4.1 Variational Autoencoder training results

First of all the training process will be presented, shown by examples of some epochs along with their respective latent space representation (which is crucial for generating results close to the dataset), followed by a loss plot. At last we present some generated examples along with a statistical analysis of note frequency between the images of the dataset and the generated ones. This whole process is replicated for all three datasets.

It can be expected that the Undertale dataset will perform better, as it is rather simplistic musically compared to the Classical dataset, and it contains more samples compared to the PMD dataset.

4.1.1 Training the VAE model

The VAE model proved to be heavier computation-wise due to the high number of trainable parameters. Regardless, the training time is fairly low, as it reaches

convergence after few epochs, based on the dataset provided.

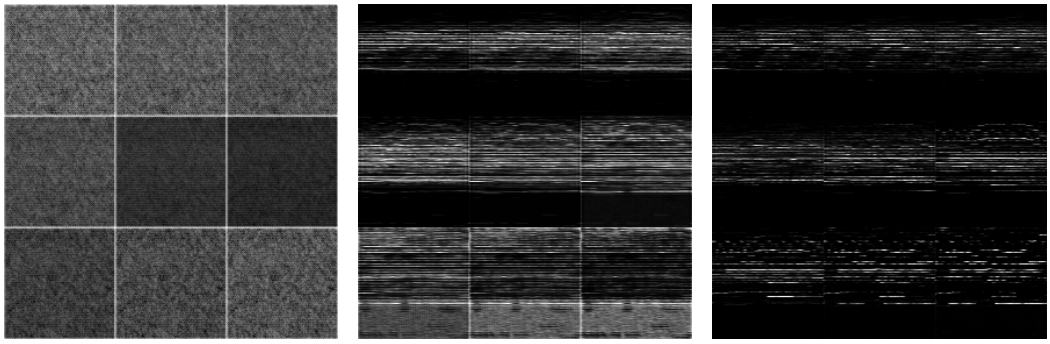
For each epoch the model trains upon batches, meaning that it executes back propagation considering a number of inputs, a batch, rather than updating the variables after each input is fed through. This number is set to x, as in typical training situation it is set to 32, 64, or 128. In addition the model reserves a portion of the dataset for validation at the end of each training epoch and the validation loss is saved for plotting. For practical reasons the dedicated script also saves the complete model, as well as the decoder model, offering the ability to retrain or generate samples.

The training history of VAE for each dataset is presented in Figure 25 with some generated examples. Those samples are the result of utilizing the decoder model, feeding it with random values in range of -4 to 4. As it was stated in Methods, the VAE modeling builds a latent space during training, which indicates the position of the clusters of each class in space. In this case the class is one, so we observe a single cluster, as well as the random variables are two, so the space will be a two-dimensional plot. The range of the random variables is set in range of -4 to 4, as we expect the cluster to be a circle positioned around 0,0 with a radius of 4.

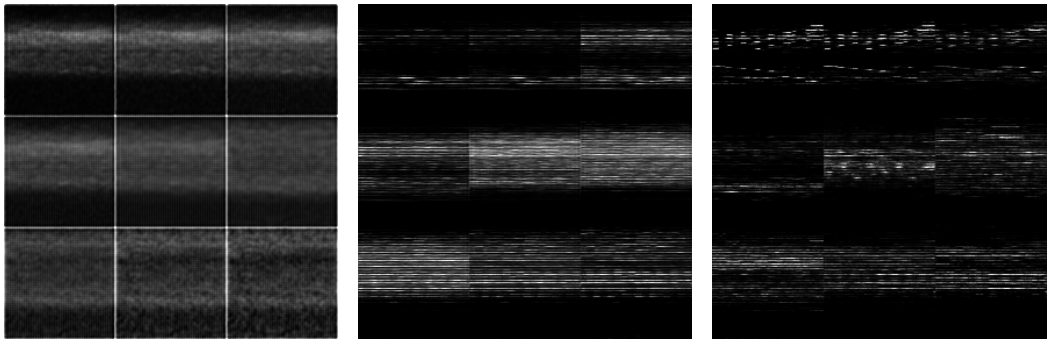
It is observed that initially, epoch 0, the model generates images with random noise for each dataset. Afterwards it is able to identify some patterns of the initial datasets. For PMD, the model at epoch 40, is observed to have a tendency to generate musical samples, although noise is still present. Same principle can be seen in the Undertale dataset at epoch 15, as well as the Classical dataset at epoch 10. At last the training is stopped at epoch 100 for all three datasets for comparison.

In Figure 26 are presented the said latent spaces for each respective epoch of the generated results.

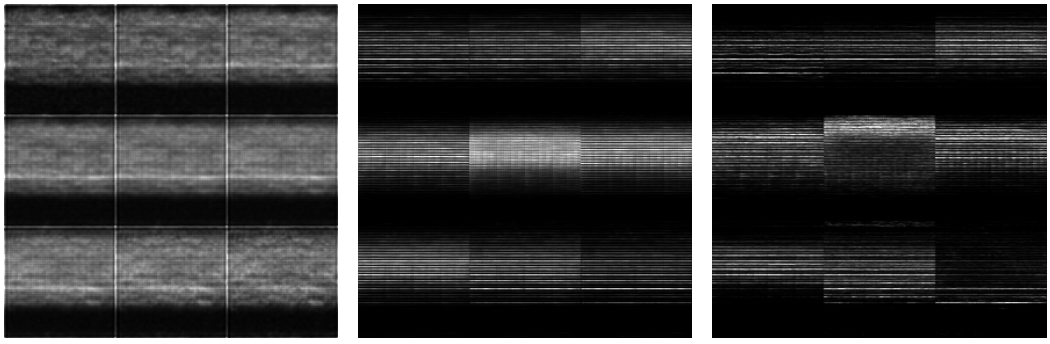
In Figure 27 are presented the losses in a graph based on each epoch. The loss with color red corresponds to the average loss of the samples the model is trained upon, and the loss with color blue, to the validation loss, which is calculated based on a small portion of the dataset (20%) which is calculated at the end of each



(a) First epoch (PMD) (b) Epoch 40 (PMD) (c) Last epoch 100 (PMD)



(d) First epoch (Undertale) (e) Epoch 15 (Undertale) (f) Last epoch 100 (Undertale)

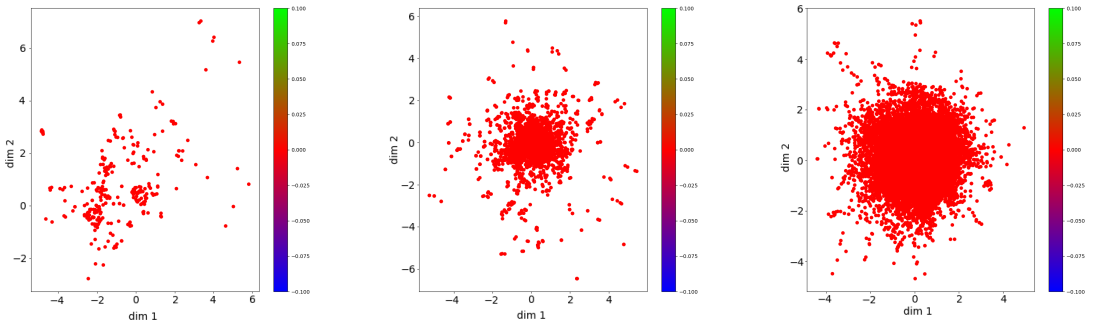


(g) First epoch (Classic) (h) Epoch 10 (Classic) (i) Last epoch 100 (Classic)

Figure 25: Generated samples for VAE during training

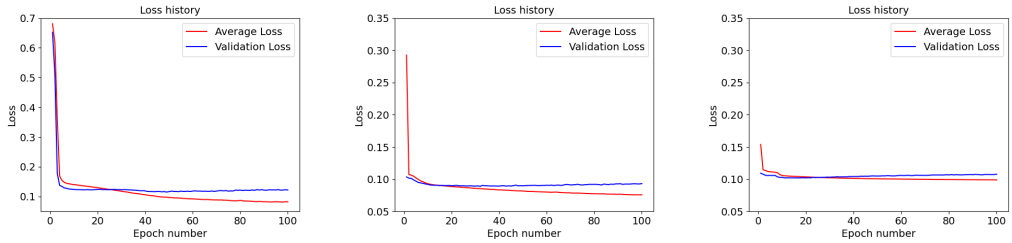
epoch.

Finally it is obvious that the number of epochs required to generate images that resemble samples from the initial pool vary for each dataset, due to the fact that each dataset contains different number of input samples. It is expected that less samples will execute less back propagation calculations for each epoch, resulting in



(a) Latent space (PMD) (b) Latent space (Undertale) (c) Latent space (Classic)

Figure 26: Latent space for each dataset



(a) Loss History (PMD) (b) Loss History (Undertale) (c) Loss History (Classic)

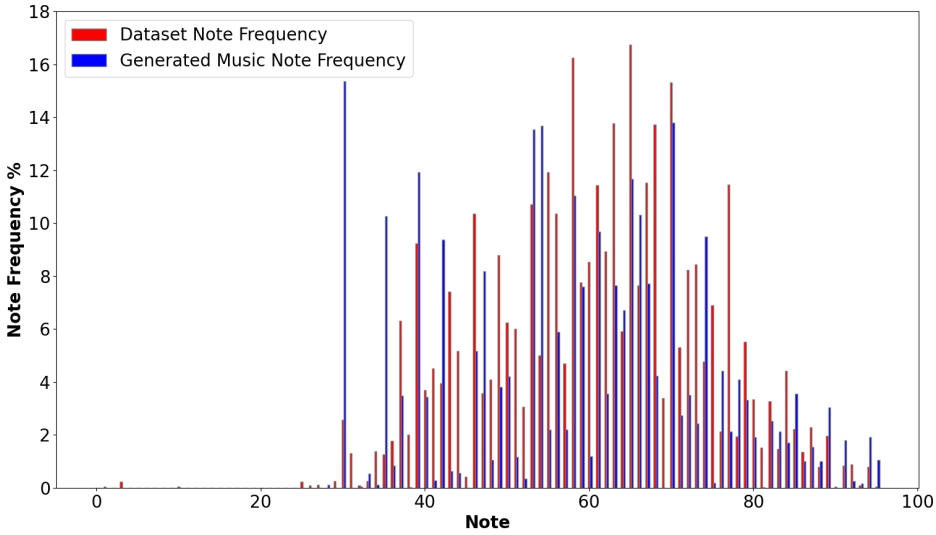
Figure 27: VAE Loss history for each dataset

requiring a greater number of epochs. In this case all three iterations are trained for 100 epochs for comparison reasons presented in Discussion section.

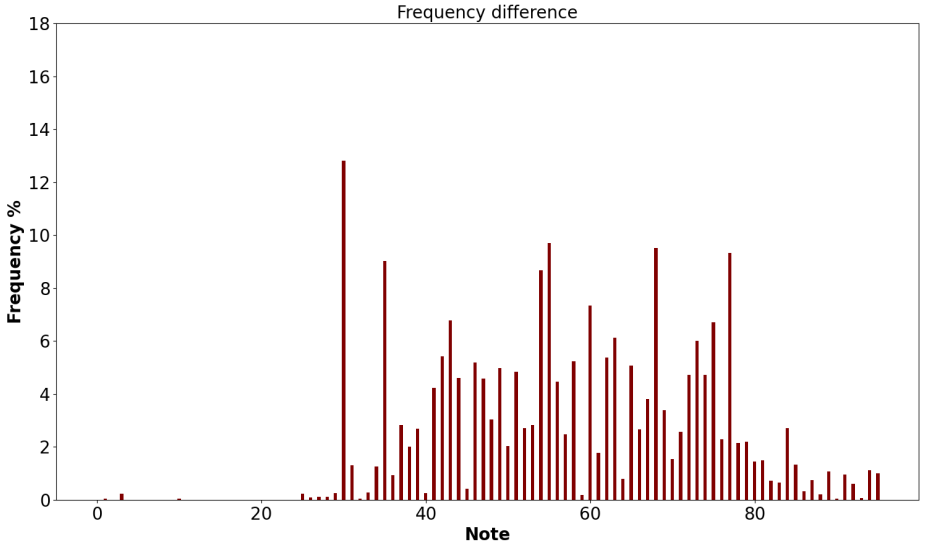
4.1.2 VAE evaluation

Evaluating a generative task is not easy, as the results are usually subjective. There is the choice of evaluation based on musical theory, but the approach of this work is to be as simplistic as possible.

In Figures 28, 29, 30, we present a statistical analysis on note frequency between tracks from the dataset and generated tracks. Each iteration generates 1000 samples and calculates a frequency for each note presented. Then we present a second figure referencing the difference between original and generated frequencies for each note for presentation purposes. This analysis aims to indicate the



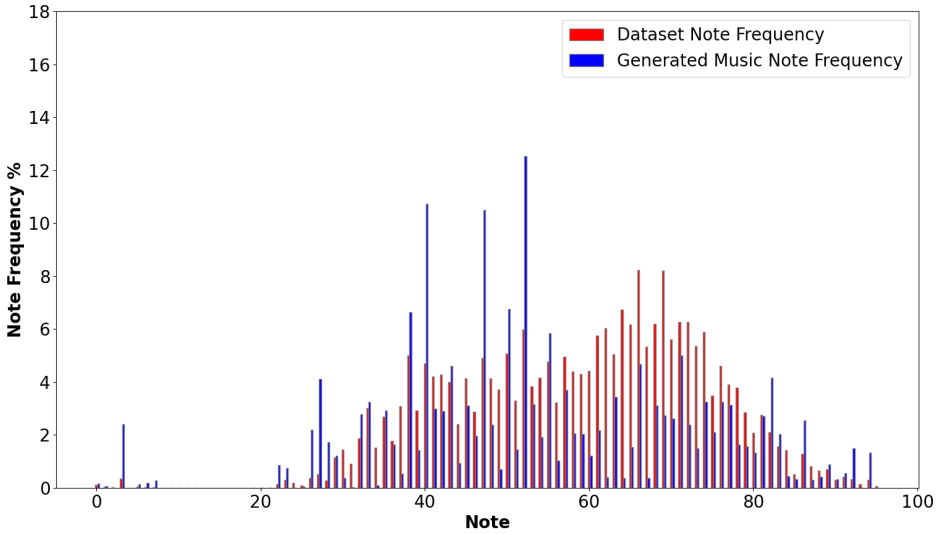
(a) Note Frequency (PMD)



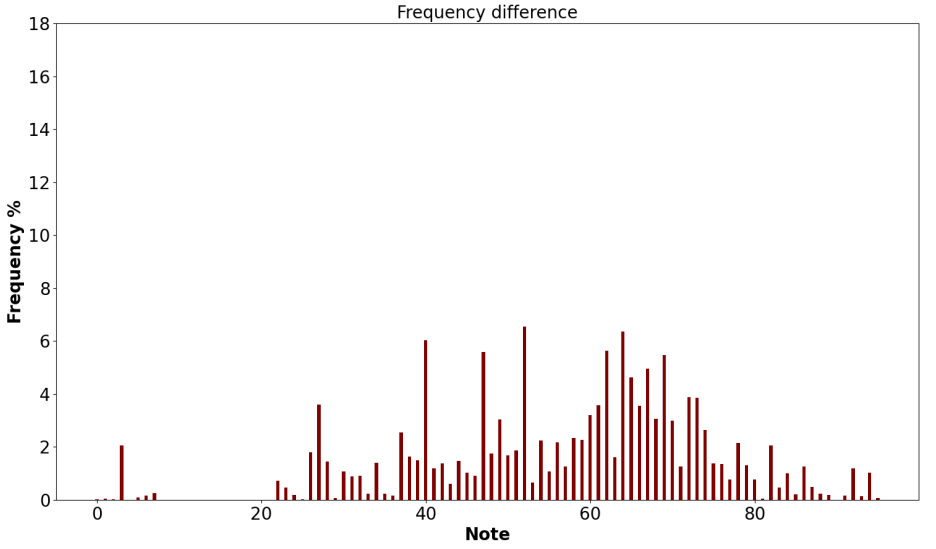
(b) Frequency Difference (PMD)

Figure 28: Note frequency bar graphs (for the PMD dataset) between real and (1000) generated samples by the VAE

behavior of each iteration, whether the model is able to generate samples that rep-



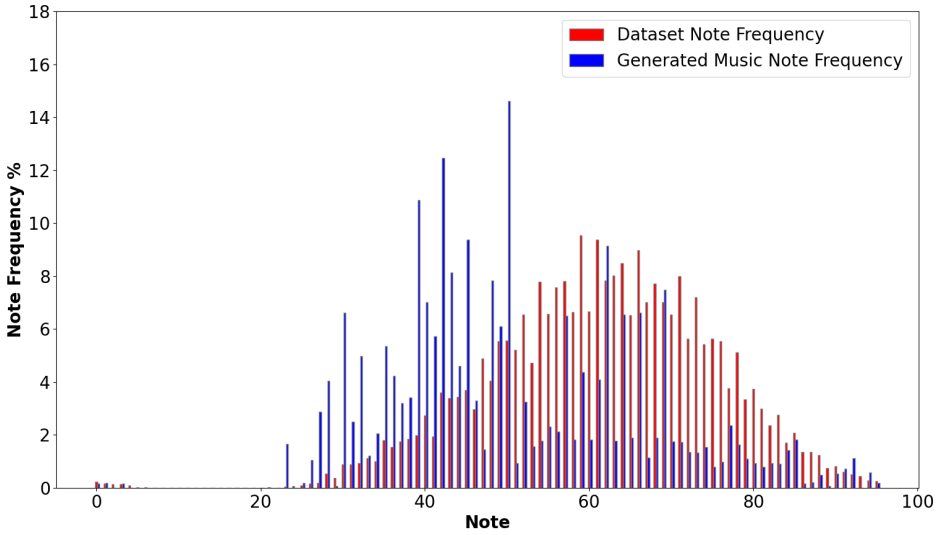
(a) Note Frequency (Undertale)



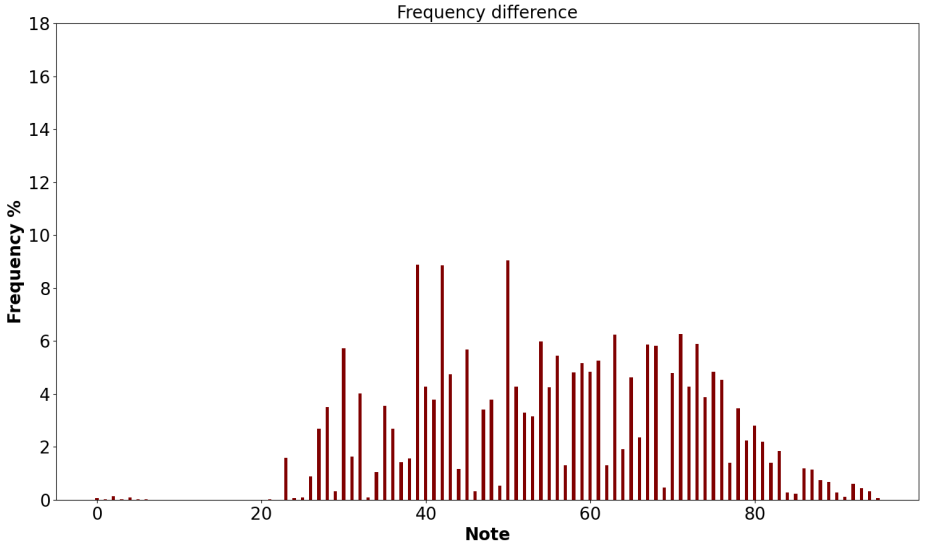
(b) Frequency Difference (Undertale)

Figure 29: Note frequency bar graphs (for the Undertale dataset) between real and (1000) generated samples by the VAE

licate each original dataset. This feature will be further analyzed in the Discussion



(a) Note Frequency (Classic)



(b) Frequency Difference (Classic)

Figure 30: Note frequency bar graphs (for the Classic dataset) between real and (1000) generated samples by the VAE

section.

4.2 GAN training results

In this subsection the GAN modeling results are presented, replicating the VAE format. Firstly training examples are shown along with loss plots. Afterwards the generated examples are analysed with the same statistical process as used for the VAE model. Each subsection evaluates the model based on all three datasets.

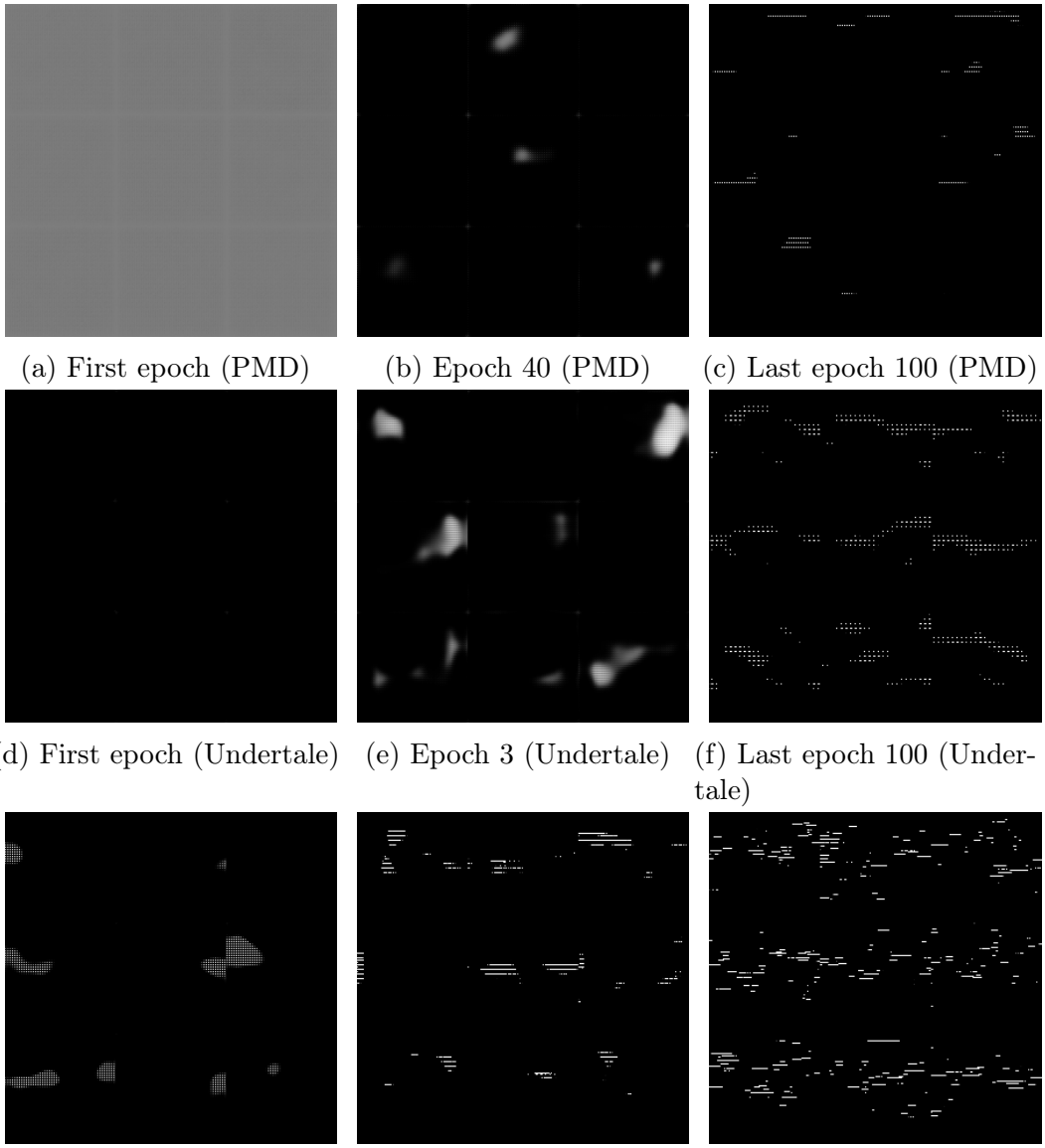
4.2.1 Training the GAN model

The GAN model, as it was built in the Methods section, consists of x number of trainable parameters (Generator and Discriminator combined). Also the number of epochs required to reach a desired stage was higher for all three datasets compared to the VAE.

Just like the VAE iteration, the GAN model trains upon batches, splitting the dataset into groups of 64, executing back propagation after each batch, rather than updating the trainable parameters after every input sample is passed through, reducing the training process while still maintaining effectiveness. Compared to the VAE, the GAN does not utilize the validation method, since loss is observed based on the two sub-models (generator, discriminator) rather than based on the complete model. For practical reasons the script that is responsible for training also saves the model as a checkpoint, along with the generator, offering the ability to retrain or generate samples outside of the training process.

In Figure 31 is presented the training history of the GAN based on some generated samples. Those samples are the result of utilizing the generator model, feeding it with random noise input vector of 100 values ranging from -1 to 1. Compared to the VAE, there is no latent space representation or clustering of the random input that can be visualised.

It is observed that initially, epoch 0, the model generates images that cannot be decoded to sound as they appear either gray (PMD), black (Undertale). The Classic dataset appears to output some gray spots that are still unable to be converted to sound. Afterwards, for PMD, at epoch 40, and for Undertale, epoch 3, the models are observed to generate some non-black spots, although conversion to music is still impossible. At last the training is stopped at epoch 100 for



(a) First epoch (PMD) (b) Epoch 40 (PMD) (c) Last epoch 100 (PMD)

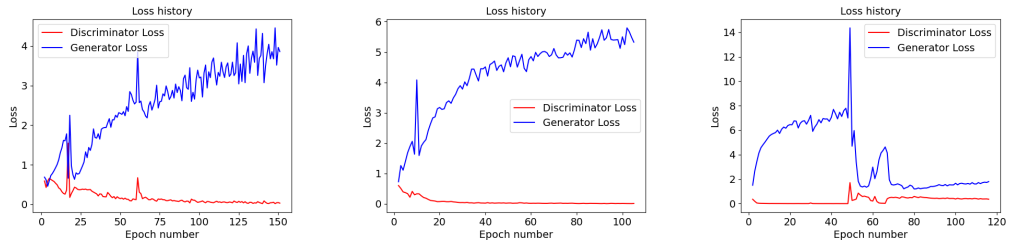
(d) First epoch (Undertale) (e) Epoch 3 (Undertale) (f) Last epoch 100 (Undertale)

(g) First epoch (Classic) (h) Epoch 50 (Classic) (i) Last epoch 80 (Classic)

Figure 31: Generated samples for GAN during training.

PMD, epoch 100 for Undertale, and epoch 80 for the Classic dataset, as the first two iterations seem unable to generate consistent images compared to the Classic iteration which is closer to what an ideal result would be.

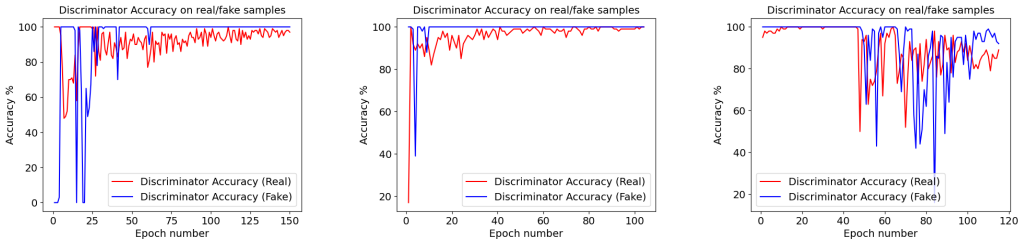
In Figure 32 are presented the losses in a graph based on each epoch. The loss with color blue corresponds to the average generator loss for all the batches in



(a) Loss History (PMD) (b) Loss History (Undertale) (c) Loss History (Classic)

Figure 32: GAN loss history for each dataset

each epoch, and the loss with color red, to the average discriminator loss. The loss values of generator and discriminator are expected to behave in an opposite manner, meaning that if the generator is performing well, low loss value, then the discriminator is fooled, thus outputting high loss value. More details and analysis on GAN performance based on losses are presented to the Discussion section.



(a) Discriminator Accuracy (PMD) (b) Discriminator Accuracy (Undertale) (c) Discriminator Accuracy (Classic)

Figure 33: Discriminator Accuracy history for each dataset

In Figure 33 the accuracy of the discriminator is presented. This accuracy value refers to the ability of the discriminator successfully identifying whether its input is real (given by the dataset) or fake (generated by the generator).

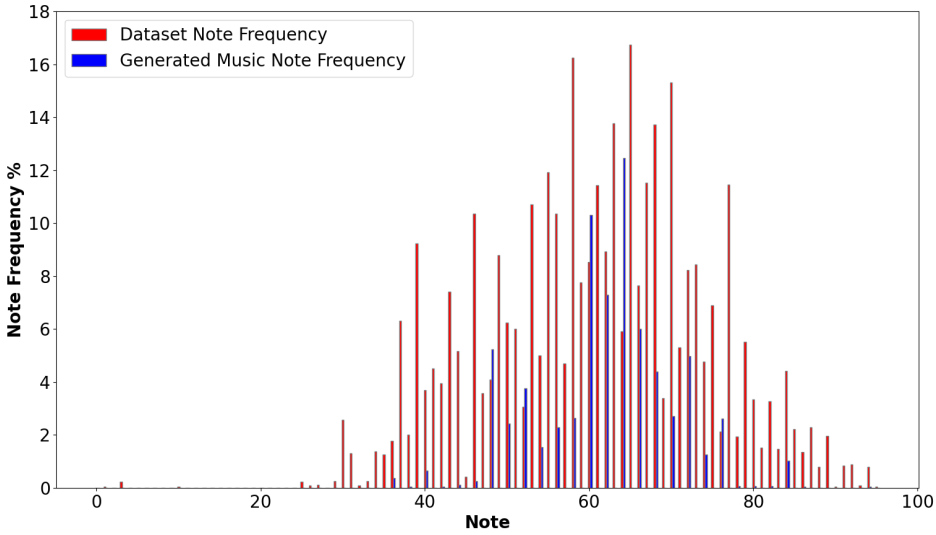
Finally just like the VAE analysis, some iterations output results faster than others, in terms of epochs, due to the fact that each dataset contains different number of samples. More input samples equate to more calculations, but more training time for each epoch. The PMD and Undertale iterations are stopped at epoch 100 as they are deemed unable to generate desired results, and the Classic iteration is

stopped at epoch 80 as the model generates close to ideal results. More details are presented in the Discussion section about model performance and behavior.

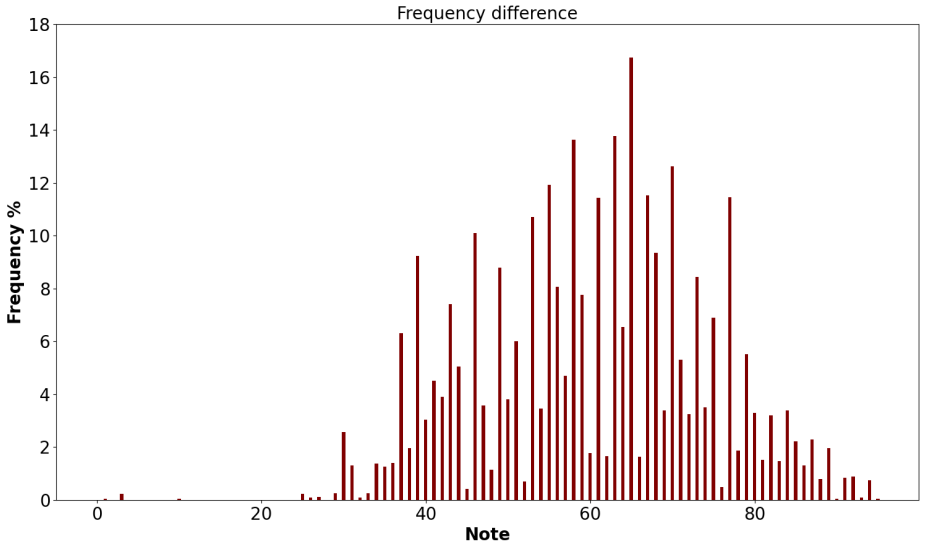
4.2.2 GAN evaluation

In this subsection the GAN model is evaluated the same way as the VAE model. Once again a generative task is one of the hardest deep learning problems, as there is no clear way to determine whether the results are desired or not (subjectivity), as in classic deep learning tasks the performance is usually dictated by metrics.

In Figures 34, 35, 36, we present a statistical analysis on note frequency between tracks from the dataset and generated tracks just like the VAE along with difference graphs. Each iteration again generates 1000 samples and calculates a frequency for each note presented. This feature will be further analyzed in the Discussion section.

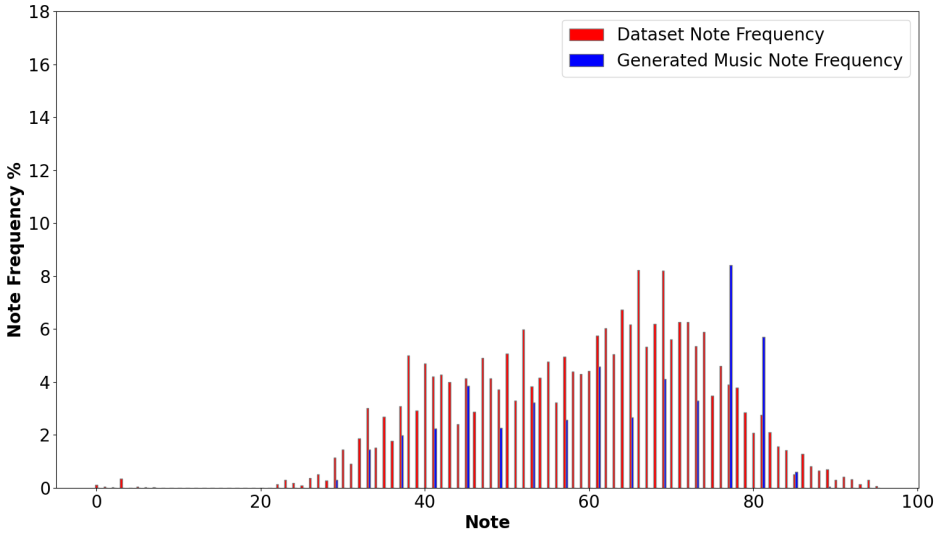


(a) Note Frequency (PMD)

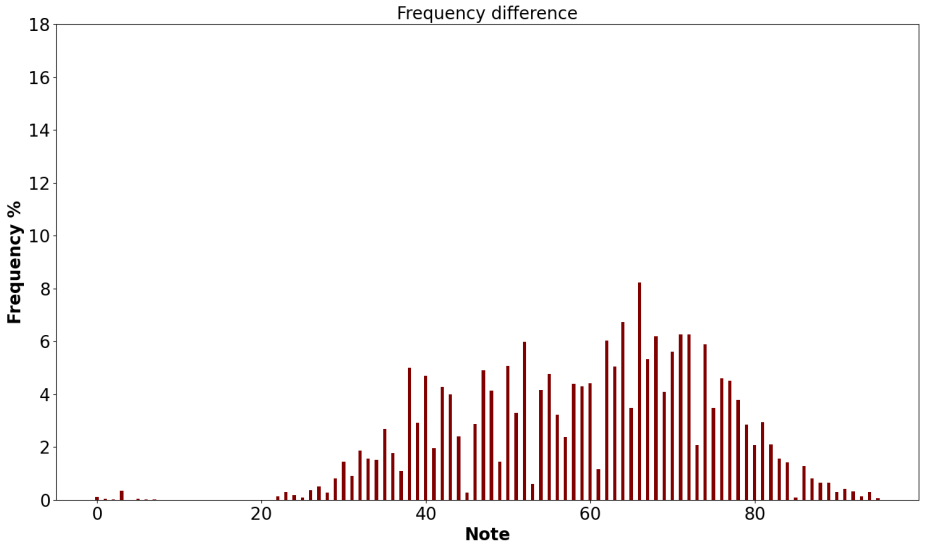


(b) Frequency Difference (PMD)

Figure 34: Note frequency bar graphs (for the PMD dataset) between real and (1000) generated samples by the GAN

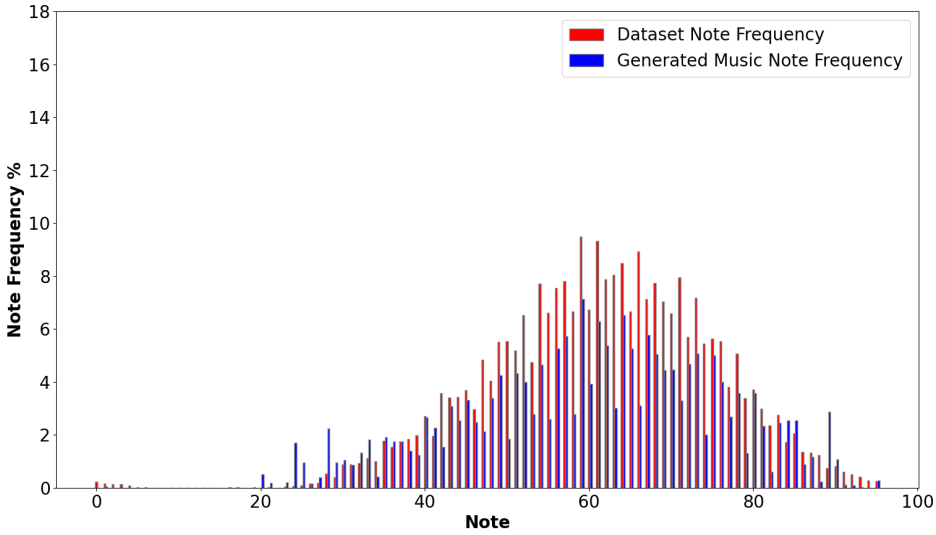


(a) Note Frequency (Undertale)

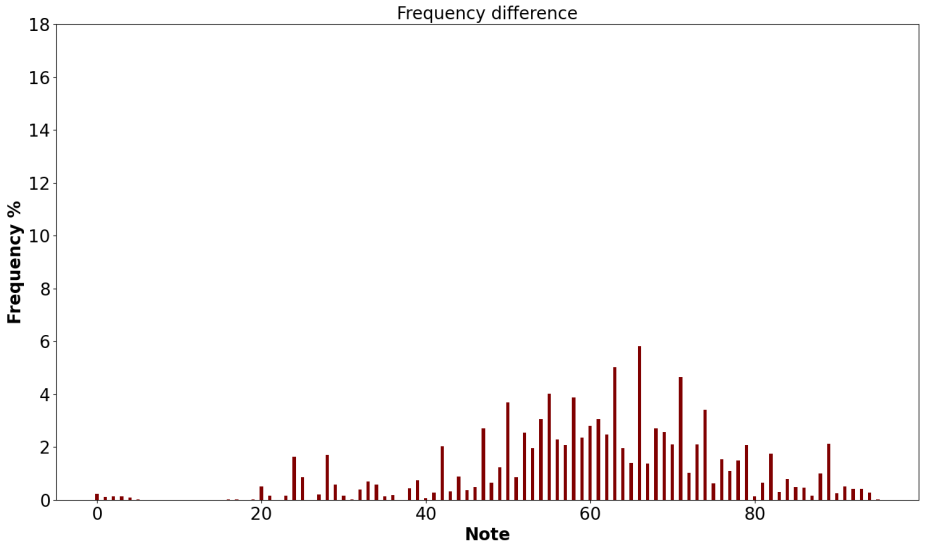


(b) Frequency Difference (Undertale)

Figure 35: Note frequency bar graphs (for the Undertale dataset) between real and (1000) generated samples by the GAN



(a) Note Frequency (Classic)



(b) Frequency Difference (Classic)

Figure 36: Note frequency bar graphs (for the Classic dataset) between real and (1000) generated samples by the GAN

5 Discussion

This section's role is to comment, discuss and observe upon the results given by the training of the VAE and GAN models presented previously. The main goal is to identify patterns among observations, give reason to possible errors as well as present possible solutions.

In general based on figures 25 and 31 the models behave differently for each dataset. The GAN in general generates cleaner images with little noise compared to the VAE. Although, in PMD there is little note selection, in Undertale the notes appear choppy and non continuous, and in Classic the tracks seem complex and somewhat random (Figure 31). In VAE's case noise is more present, PMD and Undertale iterations output some fair results, and Classic iteration is the noisiest and seems to not produce clear note sequences (Figure 25). It should be noted that those results in said figures are the raw unprocessed outputs of the networks, but for listening purposes then get filtered to either have a white or black pixel (not on the grayscale).

5.1 Note probability graphs

First of all, both models, VAE and GAN, are able to be trained upon all three datasets and generate desired images that resemble piano tablatures. The VAE iteration generates more noisy images, the Classic's case being the most noisy, and the GAN outputs cleaner images. The main goal of this work which is the extraction of images representing music and generating new ones based on that behavior, is achieved.

This offers the ability to then comment on those results and evaluate them based on melody and musical potential. Based on the probability graphs in Results section, (figures 28, 29, 30, 34, 35, 36) we observe that the notes selected from the generated samples resemble the ones from the datasets, with some offsets as seen in the difference of probability graphs. Those offsets can be present due to various reasons:

- The models are unable to replicate the datasets' images, meaning a big value

of difference is considered a non-ideal result.

- The models are defined in a way to achieve exploration or generate samples that are original compared to the inputs, meaning some value of difference is considered an ideal result.

In order to distinguish whether the models lack the ability to replicate the datasets or offer exploration and originality we observe the graphs considering spatial difference. We infer that big value of difference can be acceptable in spaces that appear frequently, but not in spaces where note probability from the datasets is small. For example in both Figures 28 and 29 differences are presented in a continuous way, yet in the first case, PMD, the average difference is greater than the Undertale dataset, which makes the latter case differences acceptable.

In VAE's case the total notes selected for all three datasets seem to resemble the notes selected from the dataset. Although the differences vary for each dataset. The Classic iteration seems to generate lower notes more frequently than higher ones, compared to the dataset which favors high notes, so the difference is rather great (Figure 30). The PMD iteration presents some great variances between some notes, indicating that the model does not replicate the dataset (Figure 30). At last, in the Undertale iteration the frequency difference is smaller, even in notes that the variance is the greatest e.g. around 6% (Figure 29). In summary the VAE generates images that contain a total number of notes close to the number of the datasets, as well as the frequency differences vary for each dataset. The Undertale iteration seems to be performing better than the other two datasets in this case.

The GAN model's total note selection varies from the VAE iteration. Training upon the Undertale dataset the GAN generates images that contain less than 20 different notes in total. The differences appear small, although that is the result of the model not selecting enough different notes (Figure 35). The PMD iteration outputs more total notes, but still not as much as the dataset, and the frequency differences are great (Figure 34). At last, in the Classic iteration the GAN generates notes close to the total number of different notes in the dataset, as well as the average frequency differences seem low, around 4% (Figure 36). In summary the PMD and Undertale iterations do not resemble the input dataset as

they select fewer total different notes, and the Classic iteration is what an ideal result should look like, selecting a great number of notes (close to equal to the dataset) and having small frequency differences.

Regardless there are two problems that are presented with the case of spatial frequency differences: firstly in practice the result of altering a note to another which is close spatially can be completely different musically in a non-ideal way, and secondly this analysis observes solely the selection of notes without considering the temporal attribute in the nature of music. For example if we alter a melodic piano track temporally, keeping the same notes but changing the order of the notes or the time offsets between them, the resulted track will sound completely different and in most cases random. Furthermore we infer that this analysis is not sufficient enough to evaluate model performance, but it offers some insights about note selection and exploration.

5.2 Metric plots

Models in Deep Learning tasks are usually evaluated based on metrics such as loss or accuracy. In a classic task, such as classification, loss is a way to determine when to stop training as well as observe the training history. Typically, in generative tasks such as the one of the current work, metric evaluation can be more complicated due to the nature of subjectivity of the results. Regardless even generative models are expected to behave in a certain way considering metric plots.

5.2.1 VAE metric plots

The training of the VAE model is evaluated based on metrics, such as validation loss, which is the loss computed based on a certain portion of the dataset reserved for prediction at the end of each epoch. The average training loss is also included for more insight. This loss is computed based on reconstruction loss and KL divergence, as stated in Methods section (9), (6). For the three datasets (Figure 27) the VAE converges to around 0.1 on validation loss. The average training loss seems to be lower than the validation loss and in cases PMD and Undertale not yet converged. In general lower training loss compared to validation loss is

an indication of overfitness, meaning the model is able to perform well on the training set, but struggles to replicate the same performance on data it has not yet processed. Ideally the difference of those two values should be close to 0, as well as the values themselves should be as low as possible. While there is a difference present, the values remain fairly low (close to 0.1). The PMD iteration starts with high losses at the first epoch and converge after a couple of iterations, also the variance between training loss and validation loss is greater due to the fact that the model overfits. Both of those reasons are the result of few input samples on the initial dataset (PMD). The greater number of samples a dataset contains, the more training it receives after one epoch, in comparison. The Undertale dataset contains more items, so the initial losses are smaller and the model does not overfit at the same rate. The same can be seen in the Classic dataset containing more items than both of the other two datasets. In summary the loss behavior of the VAE on all three datasets is what should be expected and can be classified as ideal results. With the loss histories, we could select models from earlier epochs for generation purposes, in hindsight.

5.2.2 GAN metric plots

In GAN's case, the model evaluation based on loss is completely different compared to traditional models due to the fact that we observe two losses of two competing sub-models, the generator and the discriminator (Figure 32). Also we get more information on the discriminator's decision-making based on the accuracy history (Figure 33).

Based on the Figure 32, we observe in the first two cases (PMD and Undertale) the discriminator outperforms the generator as the epochs increase, and in the case of the Classic dataset that is also present until at some point, around epoch 50, when the loss difference decreases.

The training of a GAN model is difficult for the fact that the two sub-models are trained at the same time in a zero sum game. This means that improvements to one model come at the expense of the other. GANs have plenty of problems that require solution during training such as Mode Collapse, which is the result of the generator

producing one or a small subset of different outcomes, and the Convergence Failure, which is the struggle of the whole model to reach loss convergence. The latter becomes present in this work's case, as in the first two results the model is unable to converge to a point of equilibrium.

In an ideal scenario, a stable GAN will have a discriminator loss around 0.5 and a generator loss around 1.0. Also the accuracy of the discriminator on both real and generated (fake) samples should be around 70% and 80% (Brownlee 2019c). It should be noted that initially, in the early epochs, the loss variance is expected to be great as the behavior of the sub-models are expected to start off erratic and move around a lot before convergence.

In Classic's case the GAN is able to escape from the great loss variance, at around epoch 50 (Figure 32), and the losses hover around 1.0 for the discriminator and 2.0 for the generator. This stage of convergence is close to the ideal outcome so the case of convergence failure is not as present. Also the discriminator accuracy in Figure 33 for the Classic dataset seems to behave close to the ideal scenario, as the discriminator seems to struggle to succeed in distinguishing fake and real samples 100% of the time.

In the other two cases (PMD and Undertale), the models suffer from convergence failure, loss difference is high in as it can be seen in Figure 32 and discriminator accuracy is high, especially in fake samples in Figure 33. This is the cause of the generator outputting completely random images making it fairly easy for the discriminator to distinguish fakes from reals.

Specifically in all cases the generator at some point generates full black images. In Classic's case from epoch 5 to 50 it generates only black images and then is able to recover and generate samples that replicate the initial dataset. Convergence failure is present due to the fact that the discriminator is easily identifying the fake ones from the generator by being completely black images, not resembling at all the initial dataset.

There are ways to combat this type of failure in GANs, as it is a fairly common issue in training competing models. One solution would be to add noise to the discrim-

inator input (Martin Arjovsky 2017). Another would be to penalize discriminator weights (Roth et al. 2017). Both of those solutions force the discriminator to fail more frequently giving the generator a chance to recover.

5.3 Listening to the results

Due to the nature of music, it is not possible to describe music, or in this case dictate whether the generated images decode to melodic parts, without considering music rules or principles (which can complicate this work). By decoding the generated images to MIDI files, we are able to listen to our results. Due to the complexity of input values to the generator sub-models (Decoder on VAE, Generator on GAN), it is not feasible to listen to every possible track that can be generated. The decoder receives two inputs with random values in a specific range (dictated by the space representation of clustering), and the generator receives a vector of 100 random values between -1 and 1.

By listening to many tracks generated based on all three datasets for each model, we infer that the models struggle to produce consumer-grade music. A future goal of the work would be to automate music production for certain situations, such as music for television or radio advertisements, but at this stage this is not feasible.

5.4 Model comparison

Training two different models on three different datasets (6 iterations total) we get the option of comparing each combination for the same task we initially set to solve.

VAE pros and cons:

- early loss convergence, less training time required
- performed well on Undertale dataset generating some notable MIDI files, PMD iteration was fair but tracks generated were rather random, Classic iteration performance was poor
- can manipulate input in generating desirable results based on latent space

representation

- generates noisy images

GAN pros and cons:

- sharper images
- hard to train, many possible problems such as convergence failure
- good performance on Classic dataset, recovering from convergence failure and having small note frequency differences, although tracks sound poor
- PMD and Undertale were not able to recover from convergence failure, generated images are poor for the task
- many training hours required compared to VAE
- output is harder to manipulate based on input dimension (random vector of 100 items)

6 Conclusions

Initially the main goal of this work was to produce a model that could somewhat replicate images encoded from MIDI files, that could then be decoded to music.

In summary we present the steps taken:

- A script converts MIDI files to 96 by 96 images, one axis for the note values and the other for the time ticks.
- The models, GAN and VAE, are trained upon the image data sets, saving model objects, loss values and space representation (for the VAE) at the end of each epoch. Training stops at a stage when desired samples are generated.
- Another script imports the generator sub-models (generator for GAN, decoder for VAE), generates and saves image samples for evaluation. Those images are also converted to music format and saved for listening.
- At last an evaluation script is ran to create loss plots of training history, and bar graphs that indicate note probability of generated and input samples for comparison.

By extensive research and multiple iterations of deep learning models we managed to create two solutions that were able to generate musical samples in a very efficient way regarding time and memory. Those models were trained upon three different, in length (number of samples) and genre, data sets each. Some iterations outperformed others, but all successfully achieved what was initially desired.

The GAN model trained upon the Classic dataset escaped from convergence failure and based on loss plots and frequency difference bar graphs its training was successful and close to ideal, yet the decoding of images back to MIDI files was not that ideal. The VAE model in the Undertale case produced some samples that were notable musically, yet the samples produced were not as clean as the in the GAN's case. Both models were somewhat successful, although ideally we would want a model that is able to be trained upon different types of datasets and generate similar results.

The reason this work was undertaken was due to the fact that music has a huge

role in human lives. With such a basic solution we are able to create a bridge between computer science, and more specifically deep learning, with music. Deep learning being a growing field of research in a very fast rate, could be a very useful tool for musicians and composers, as well as offer real-time generated and original music that the consumer has never listened to.

This work is a starting point, a simplistic solution for music generation, that other researchers may continue and perfect what was initiated. Furthermore our results were not immaculate, as there are very few instances of tracks that could be enjoyed and listened to, regardless, we want this work to be evaluated based on its efficiency and effectiveness of model training.

7 Recommendations

The nature of this work is mostly technical, aside from the engineering of the data pre-processing. That being said there are actions that can be taken to further improve training and results.

Some solutions are presented that could result in more desired generated samples. Most complex deep learning tasks usually require more training. Such is the case for the GAN, being a cost-intensive model to train (many parallel calculations, many epochs), which could output better results if trained more. Specifically it is observed to output full black images after a certain epoch (for each dataset), with high generator loss and low discriminator loss, but with further training it could stabilize and outperform past epochs, such as the case of the Classic training where the model is able to escape from convergence failure.

Also there is much engineering that could be done concerning the deep learning layers, meaning the addition of certain layers, the number of repetition of some layer sequences (number of Conv2Ds along with activations), and the tuning of hyperparameters or other attributes (such as batch number, filters, kernel sizes etc.) could prove more efficient. Regardless this typically is a matter of trial and error.

In addition the pre-processing of the datasets could be altered. One idea is to utilize three channels instead of one, meaning colored images instead of gray-scaled. This extends the time ticks of each image by three times the initial range, or it could be utilized to indicate the power at which each note is being pressed, offering an extra layer of possible melodic results.

This work trains upon datasets that consist of piano tracks. Given that such a basic iteration of music generation worked in a desired way, there could be multiple instrument training, for parallel music composition, or inputs from multiple genres. This is feasible as VAE modeling can be represented in space, and clustering could be classified, meaning we could potentially train upon multiple instruments or genres (classes) at the same time and generating samples based on the values at which each class clusters. Another option would be to utilize Conditional GAN

models as they offer the option to implement conditions, as their name suggests, based on musical principles allowing for more control in training.

While researching, we initially tried out a note prediction approach, utilizing Recurrent Neural Networks (RNNs) and Long-Short term Memory (LSTM) models, where the model would be given an input of a track and would predict the next action that should be taken, or note that should be played, acting as a classification task. Regardless, the problem that we came across was that the number of combinations of parallel notes, also known as chords, of a dataset is high, resulting in thousands of nodes at the final layer of classification. With better handling and engineering this solution could be just as good, or even better than a typical generative task, offering composers the ability of assistance in completing musical parts.

Finally, it is the nature of deep learning to require heavy computing and machines that consist of complex graphical units, as well as memory. That being said, we recommend retraining of the current models, with higher in cost values of hyperparameters and attributes such as filters of each layer. With faster rate of learning the models could be trained for more epochs offering greater insight about each model's behavior and results.

References

Bjørndalen, Ole Martin (2013). *Mido library documentation*. URL: https://mido.readthedocs.io/en/latest/index.html.

Briot, Jean-Pierre, Gaëtan Hadjeres and François-David Pachet (2017). 'Deep learning techniques for music generation–a survey'. In: *arXiv preprint arXiv:1709.01620*.

Brownlee, Jason (2019a). *A Gentle Introduction to Generative Adversarial Networks (GANs)*. URL: https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/.

– (2019b). *How Do Convolutional Layers Work in Deep Learning Neural Networks?* URL: https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/.

– (2019c). *How to Identify and Diagnose GAN Failure Modes*. URL: https://machinelearningmastery.com/practical-guide-to-gan-failure-modes/.

Martin Arjovsky, Leon Bottou (2017). *Towards Principled Methods for Training Generative Adversarial Networks*. URL: https://arxiv.org/pdf/1701.04862.pdf.

Piyush Madan, Samaya Madhavan (2020). *An introduction to deep learning*. URL: https://developer.ibm.com/articles/an-introduction-to-deep-learning/.

Roth, Kevin et al. (2017). 'Stabilizing training of generative adversarial networks through regularization'. In: *arXiv preprint arXiv:1705.09367*.

Schmidt-Jones, Catherine (2013). 'Understanding basic music theory'. In.

Shafkat, Irhum (2018). *Intuitively Understanding Variational Autoencoders*. URL: https://towardsdatascience.com/intuitively-understanding-variational-autoencoders-1bfe67eb5daf.