University of Hradec Králové
Faculty of Informatics and Management
Department of Informatics and Quantitative Methods

Music generation using neural networks

Bachelor Thesis

Author: Aleksey Yanushko

Study Branch: Applied informatics

Thesis Supervisor: Milan Košťák

Hradec Králové November 2022

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature, and other professional sources.

Hradec Králové, 17th November 2022 Aleksey Yanushko

Acknowledgement:

I would like to thank Ing. Košťák Milan for relevant remarks and supervision of this thesis.

Annotation

Title: Music generation using neural networks

This thesis focuses on the task of music production with the use of machine learning. exploring the current trends in the field of music generation, as well as providing a relevant practical solution to this task.

The theoretical part is focused on the principles of artificial neural networks, a description of their operation and an explanation of the possibilities of use. Theoretical part also gives an overview of the most prominent modern projects aimed at providing users with the ability to produce novel music without the need for any prior experience.

The practical part proposes a solution and subsequent implementation of an artificial neural network capable of generating unique melodies and providing results of the set implementation.

Key words: neural networks, artificial intelligence, music generation

Anotace

Název: Generování hudby pomocí umělých neuronových sítí

Bakalářská práce se zaměřuje na úlohu hudební produkce s využitím strojového učení při zkoumání současných trendů v oblasti tvorby hudby a poskytování relevantního praktického řešení tohoto úkolu.

Teoretická část je zaměřena na principy umělých neuronových sítí, popis jejich fungování a vysvětlení možností využití. Teoretická část také podává přehled nejvýznamnějších moderních projektů, jejichž cílem je poskytnout uživatelům možnost produkovat unikátní hudbu bez nutnosti předchozích zkušeností.

Praktická část navrhuje řešení a následnou implementaci umělé neuronové sítě schopné generovat unikátní melodie a poskytuje výsledky této implementace.

Klíčová slova: neuronové sítě, umělá inteligence, generování hudby

Content

List of images

List of abbreviations used

BNN - Biological neural network

ANN - Artificial neural network

MLP - Multilayer perceptron

RNN - Recurrent neural network

LSTM - Long Short-Term Memory

CNN - Convolutional neural network

GAN - Generative adversarial network

AE - Autoencoder

VAE - Variational autoencoder

AGI - Artificial general intelligence

KL - Kullback-Leibler

1 Introduction

One of the most complex forms of artistic endeavor is the creation of music. Contrary to books, poems, paintings, and films, music holds no bond to the context of the outside world and doesn't necessarily require an understanding of such ideas as, for instance, the meaning of words or notes. And while a person's perspective on the world unquestionably affects how he understands art, music continues to be the most independent means of evoking strong emotions in others. Furthermore, the primary content of music can be represented in a relatively straightforward data format. This fact is what gave rise to notes. Musical scores can be symbolized in a computer as a series of binary arrays, as will be demonstrated below, which makes the algorithmic processing of music simpler. Based on this, music seems to be a fertile field of research on creativity and its modeling in artificial intelligence. However, there are still no reasonable scientific theories as to why some sequences of notes are perceived as music, while others are not. It is also a philosophical question of whether these dependencies were formed naturally, or rather were the result of the historical process. One way or another, music contains certain patterns that are intuitively understandable to all people, which have not yet been mathematically formulated or modeled. But engineers and scientists are developing new and improved ways to create music at a human-like level by using contemporary technologies such as deep learning.

The purpose of this thesis is to outline the most prominent modern methods and deep learning models for creating music, propose and put into use an artificial neural network that can produce new and unique melodies, and demonstrate the results.

2 Purpose of the thesis & methodology

2.1 Purpose of the thesis

This thesis aims to explore how it is possible to teach an artificial neural network to "compose" music. It will first be necessary to become familiar with the neural network - its components, and structure, as well as consider one of the methods of its training used in the developed program. Later, familiarize with modern approaches and solutions to problems of artificial music generation. Finally, the construction and implementation of the algorithm for generating and harmonizing the melody.

2.2 Methodology

The theoretical and practical framework for the thesis was outlined with the use of both printed and online sources, such as articles from Towards Data Science and Semantic Scholar, as well as development blogs from Magenta and OpenAI. Practical implementation required multiple sources with exemplary solutions provided on GitHub.

3 Artificial neural networks

The chapter describes basic concepts and ideas behind neural networks, as well as gives a brief overview of the existing architectures and types of neural networks that are used nowadays, their specifics, and components.

3.1 Biological neuron

The fundamental feature of artificial neural networks (ANNs) is their ability to process information in a way that is similar to that of a human's nervous system, particularly regarding its capacity to learn from errors and make corrections. Since it is impossible to fully describe the functioning of biological systems due to their high organizational complexity, researchers are interested in simplifying the model of the nervous system to its most basic unit, the neuron. (Figure 1).

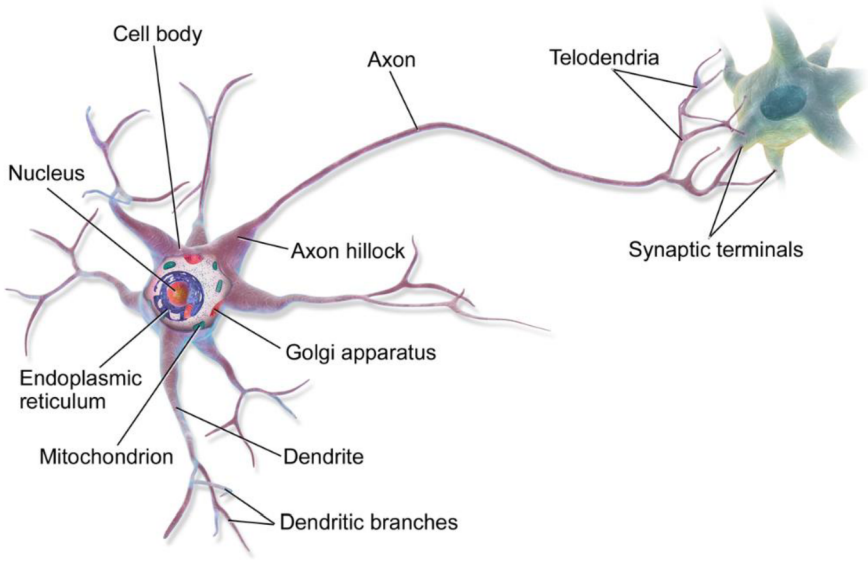

Figure 1: Biological neuron
Source: Géron (2017)

A neuron is a cell that is found in the cerebral cortex of animals. It consists of a cell body that houses the nucleus and most of the cell's intricate parts. Numerous branching extensions are called dendrites, and one extremely long extension is called the axon. The axon's length may be only a few times or even tens of thousands of times greater than the cell's body. The axon splits into numerous telodendria near its extremity, and these telodendria have tiny synaptic terminals at their tips that

connect to the dendrites of other neurons. These synapses allow biological neurons to receive brief electrical pulses from other neurons, known as signals. A neuron fires its signals when it has received enough signals from other neurons within a span of a few milliseconds.

While biological neurons appear to function in a simple manner, they are connected to thousands of other neurons and collectively form a massive network of billions of neurons. Similar to how a complex anthill can develop from the joint efforts of small ants, complex calculations can be carried out by a large network of simple neurons. Although research on the architecture of biological neural networks (BNN) is still ongoing, certain brain regions have been mapped, and neurons are frequently arranged in successive layers as seen in Figure 2 (Géron, 2017).

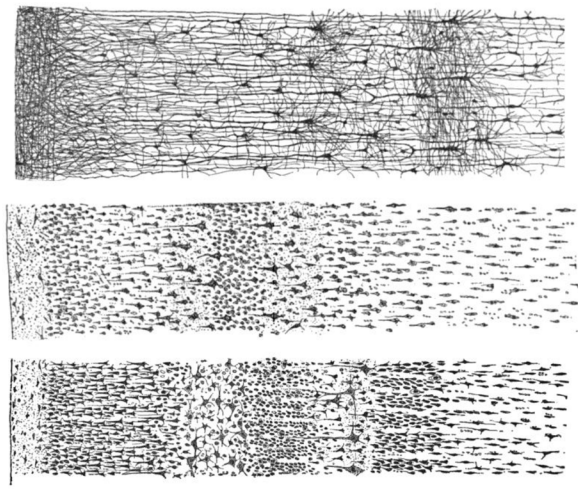

Figure 2: Neuron layers
Source: Géron (2017)

3.2 Artificial neuron

One or more binary (on/off) inputs and one binary output make up the very basic model of the biological neuron described by Warren McCulloch and Walter Pitts (Mcculloch & Pits, 1943) that eventually came to be known as an artificial neuron. When a certain number of its inputs are active, the artificial neuron simply turns on its output. McCulloch and Pitts demonstrated that it is possible to create an artificial neural network that can compute every desired logical proposition, even with such a simple model. As shown below, several ANNs can carry out a variety of logical

operations (Figure 3), presuming that a neuron is activated when at least two of its inputs are active.

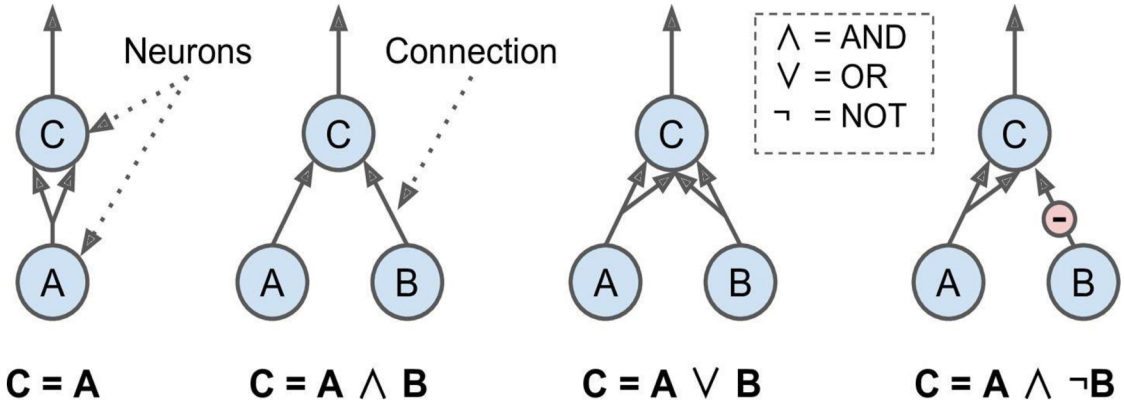

Figure 3: ANN performing simple logical computations
Source: Géron (2017)

Simply explained, the first network from the left is the identity function: if neuron A is activated, neuron C is activated as well (because it gets two input signals from neuron A), but if neuron A is turned off, neuron C is likewise disabled.

A logical AND is performed in the second network: only when both neurons A and B are active, neuron C is activated, because a single input signal is not enough to activate neuron C.

The third network performs a logical OR when either neuron A or neuron B is activated, triggering neuron C. (or both). If either neuron A or neuron B is activated, the third network performs a logical OR, activating neuron C (or both).

The fourth network calculates a somewhat harder logical assertion: neuron C is only turned on if both neuron A and neuron B are off. This is because the fourth network computes a marginally simpler logical statement assuming that an input link can block the neuron's activity (which is the situation with biological neurons). If neuron A is always active, then there is a logical NOT since neuron C is active while neuron B is off and conversely (Géron, 2017).

In the study of ANNs, neurons are seen as a system of expression from the n-dimensional space of inputs, the characteristics of which are described by signals

from other nerve cells' outputs or by signals from some external environment, into the one-dimensional space (scalar signal) at a neuron cell's output (Figure 4).

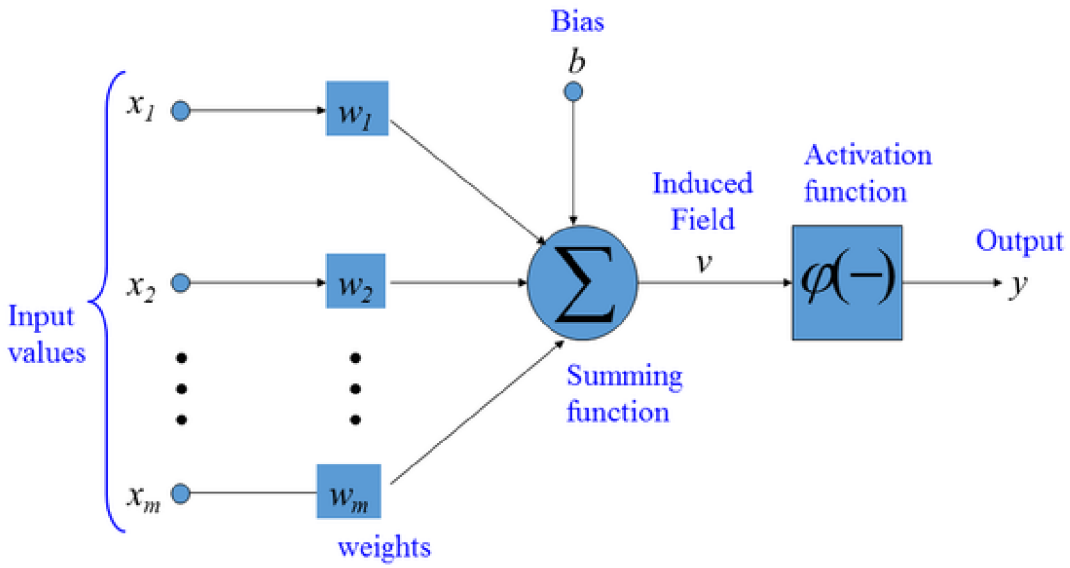

Figure 4: ANN model
Source: Gabor Mellie's Research Knowledge Base (2022)

Despite their diversity, this configuration serves as the basis for almost all types of networks. The weights used to multiply inputs, which correspond to the synaptic potency of a biological neuron, are represented by the values w1, w2, and wm. Other neurons' outputs are represented by the values x1, x2, and xm. The degree of neuron activity is later calculated by adding up all the results. The summation block, which resembles the body of a biological neuron structurally, creates the output net by pre-combining the weighted inputs algebraically. The described process has the following mathematical interpretation:

$$net = \sum_{i=1}^{n} w_i x_i + w_0 \text{ - net calculation}$$

$$y_i = \varphi(net) \text{ - activation function}$$

where w_0 is the bias

w_i is the weight of the i-th neuron

x_i is the output of the i-th neuron

n is the number of impulses entering the processed neuron

φ(net) is the activation function

y_i is the output signal of the neuron

The value of w_0 reflects the so-called displacement, which later leads to an increase in the learning speed of the network by shifting the start of the countdown of the activation function. Unlike other weights that are connected to the outputs of other neurons, the bias is connected to the signal "+1", but its training occurs on an equal footing with all weights. The sum of all the results is not the final solution, for providing the actual result the activation function is used (Foster, 2019).

The summation block's output signal is transformed into the final valid result by the activation function. It should be noted that different types of neural networks (or even layers) may have different activation functions. The following activation functions are most often used in actual practice:

1. threshold
2. sigmoidal
3. linear
4. tangential

The described system (and many of its varieties) is the basis of the perceptron, the structure of which is a layer of neurons connected by weighting factors with many inputs. But there are systems that have a higher organization.

3.3 Artificial neural network training

Training the neural network is an interactive process, it is referred to as the adjustment of the neuron's weights and thresholds. The capacity of neural networks to correct their errors in accordance with predetermined rules and to gradually improve their efficiency over time are two of its most crucial characteristics. According to ANN theory, training is the act of configuring a neural network's free parameters by modeling the environment built into the network. The type of training is characterized by predefined parameters. The following phases can be used to categorize the learning process:

1. Certain signals are given from the external environment.
2. The free parameters of the neural network are corrected, thereby changing its internal structure.
3. As a result, the network reacts differently to subsequent pulses.

This sequence is a learning algorithm. However, the described technique is not considered as being universal due to the diversity of neural network architecture types. When training artificial neural networks, a variety of methods may be utilized, each having advantages and disadvantages of their own.

In the case of multilayer network training, the error backpropagation technique is frequently utilized, which helps to get around problems with altering the weights of hidden layers. Since the backpropagation method works far more efficiently than earlier training techniques such as the Conjugate gradient or Levenberg-Marquardt algorithm that took weeks of calculations with the use of early digital computers, neural networks may now be employed to address previously unsolvable issues. And while there are multiple modern alternatives such as Genetic algorithm and Hilbert-Schmidt independence criterion etc., backpropagation has become an industry standard, and many frameworks (Tensorflow, PyTorch) have been built on it. Training using backpropagation involves modifying each layer's parameters so that, on average, there is little to no discrepancy between the network signal and the external training signal. The work of this method of training can be represented as a sequence of certain stages (Briot, Hadjeres, Pachet, 2020).

First, setting the initial conditions for all synaptic scales by means of sufficiently small random numbers so that the activation functions of neurons do not enter saturation mode in the initial stages of training (protection against "paralysis" of the network). Second, feeding the image to the network input and calculating the outputs of all neurons and later calculation of local errors for all layers according to the specified training vector and computational intermediate outputs. Clarification of all synaptic scales and repeating the process by feeding the next image to the network input (Géron, 2017).

The learning process will be carried out until the error in the output of the ANN becomes small enough, and the weights stabilize at some level. After training, the neural network acquires the ability to generalize, i.e., begins to correctly recognize patterns that are not represented in the training sample.

3.4 Artificial neural network architecture

The architectures of neural networks are closely related to the used learning algorithms. There are numerous existing architectures, ranging from the simplest single-layer architectures to more sophisticated ones comprised of dozens of distinct types of layers, each with a set of neurons that are designed to perform a specific set of calculations.

3.4.1 Single-layer Neural Networks

Single-layer network of direct propagation, where neurons are arranged in layers is one of the simplest existing architectures. In the most basic cases, such a network has a layer of source nodes serving as the input, from which information is passed to the layer of neurons serving as the output (computational nodes), but not the other way around. A network like this is referred to as an acyclic or direct distribution network. Figure 5 depicts the layout of such a network with three nodes in each layer (input and output). Source nodes are not considered when calculating the number of layers because they don't perform any calculations.

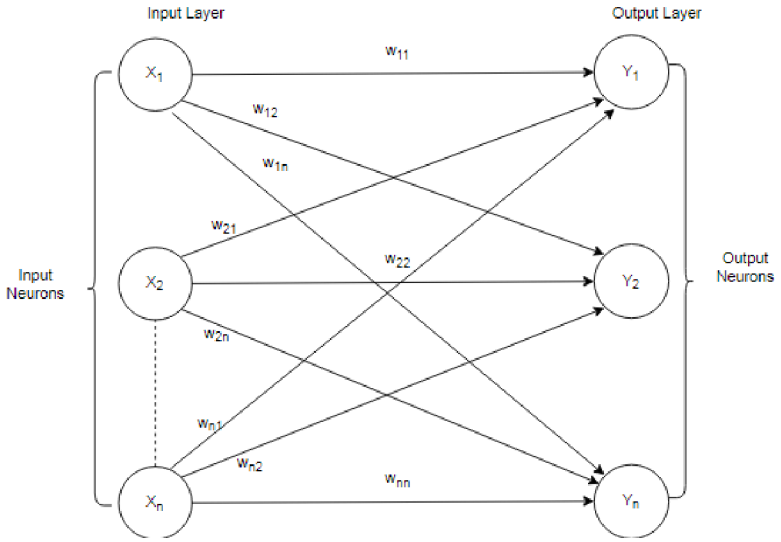

Figure 5: Single layer neural network
Source: Elsaraiti & Merabet (2021)

3.4.2 Feedforward Neural Networks

Another class of neural networks - feedforward neural networks, is characterized by the presence of one or more hidden layers or passthrough layers. The nodes of these hidden layers are called hidden neurons, or hidden elements. The hidden neurons' task is to serve as an intermediary between the neural network's output and the external input signal. By adding one or more hidden layers, it is possible to highlight high-order statistics. Such a network allows to highlight the global properties of data using local connections due to the presence of additional synaptic connections and an increase in the level of interaction of neurons. When the size of the input layer is large enough, hidden neurons have a particularly strong ability to isolate high-order statistical dependencies. The source nodes of the network input layer form the corresponding elements of the activation pattern (input vector), which make up the input signal coming to the neurons (computational elements) of the second layer. The output signals of the second layer are used as input signals for the third layer, etc. Usually, the neurons of each layer of the network are used as input signals, while the output signals of the neurons of the previous layer only. The set of output signals of the neurons of the last layer of the network determines the overall response of the network. The network shown in Figure 6 has six input neurons, seven hidden neurons and one output neuron (Thakur & Konde , 2021).

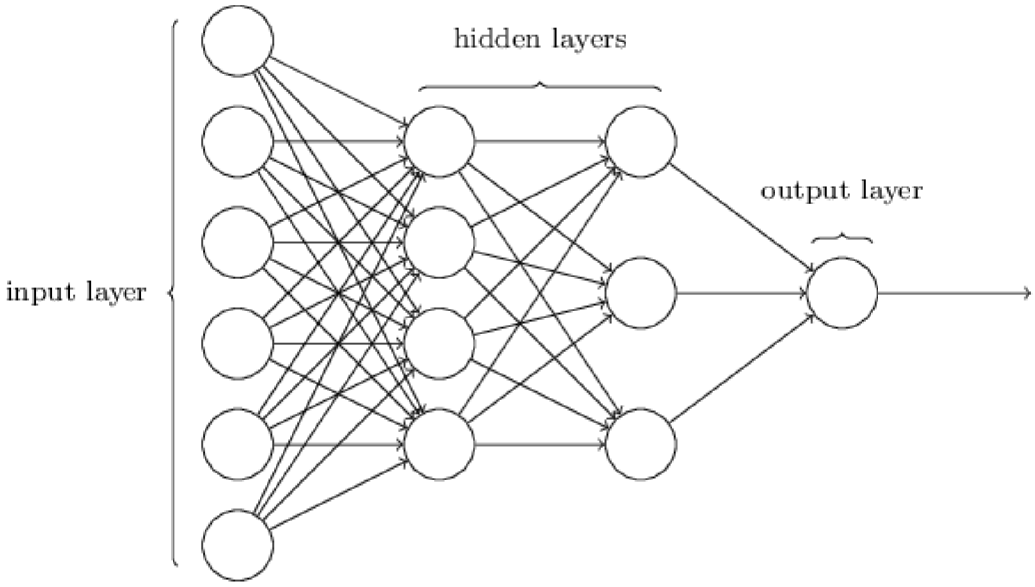

Figure 6: Multilayer neural network
Source: Thakur & Konde (2021)

It should be noted that a neural network is considered fully connected in the sense that all nodes of each layer are connected to all nodes of adjacent layers. If some of the synaptic connections are missing, such a network is called an incomplete network. It would be wrong to say that the human thought process begins every second from an empty space. A person does not discard previous experience to start thinking from scratch. Human thoughts have a certain constancy. But traditional neural networks are not capable of this, and this is obviously a serious flaw.

3.4.3 Recurrent Neural Networks

Recurrent neural networks (RNNs) are cyclic networks that are able to store information, and are designed to solve the problem that feedforward networks have presented. A recurrent neural network differs from a direct propagation network by the presence of at least one feedback. For example, a recurrent network may consist of a single layer of neurons, each of which directs its output signal to the inputs of neurons in the previous layer. The architecture of such a neural network is shown in Figure 7.

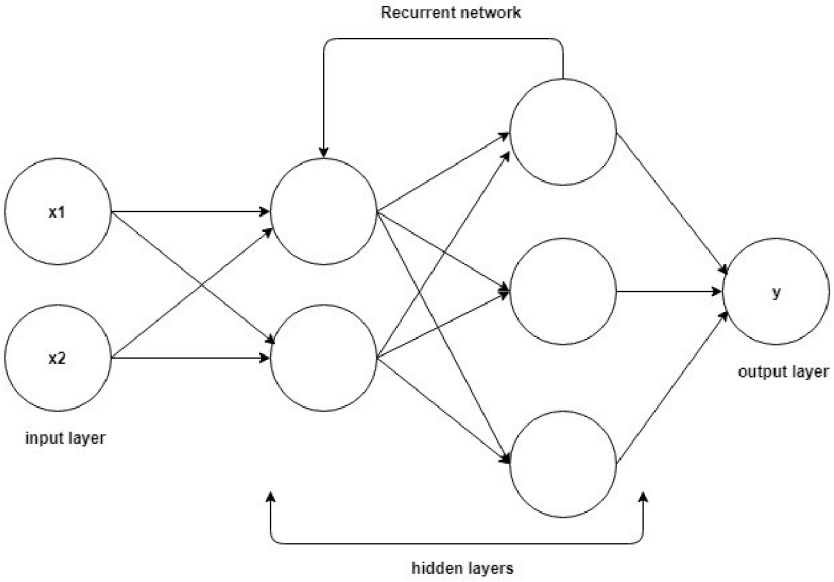

Figure 7: Recurrent neural network
Source: Maklin (2019)

The presence of feedback in the networks has a direct impact on the ability of such networks to learn and on their performance. Moreover, feedback involves the use of unit latency elements, which leads to nonlinear dynamic behavior, if the network contains nonlinear neurons (Maklin, 2019).

3.4.4 Long short-term memory model

Long Short-Term Memory networks - commonly simply referred to as "LSTM" - are a special kind of RNNs capable of learning long-term dependencies.

To better understand how LSTMs solve the long-term dependency problem it is necessary to look at their structure. In Figure 8, each line transmits an entire vector from the output of one node to the inputs of the others. Small circles represent point-by-point operators, such as vector addition, while small rectangles represent trained layers of a neural network. Merging lines indicate concatenation, while branching lines indicate that their contents are copied, and copies are sent to separate locations.

The main idea of the LSTM is its cellular state, which is like a conveyor belt (Figure 8). It moves straight along the entire chain with only a few linear interactions. Information can simply flow through it unchanged if needed (Dorian, 2021).

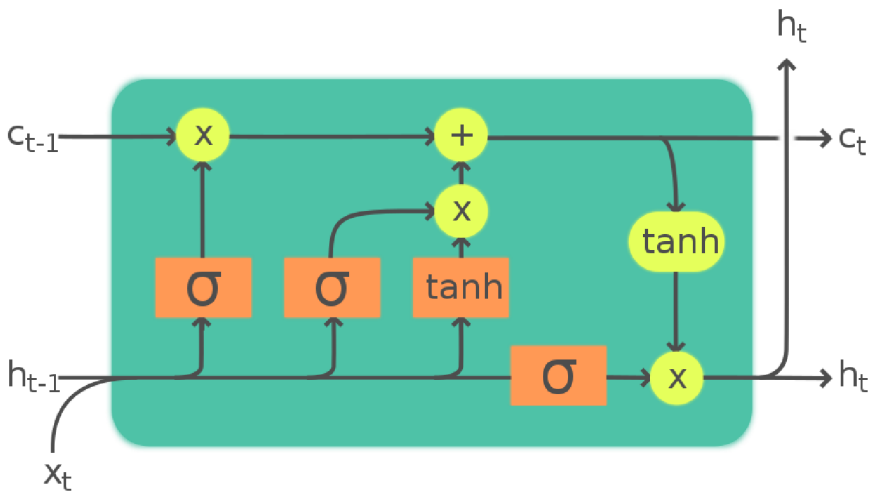

Figure 8: LSTM single cell model
Source: Dorian (2021)

LSTM can remove or add information to the cellular state. However, this ability is carefully regulated by structures called valves (Figure 9). Valves are a way to selectively let information through. They are composed of a sigmoid layer and a point-by-point multiplication operation.

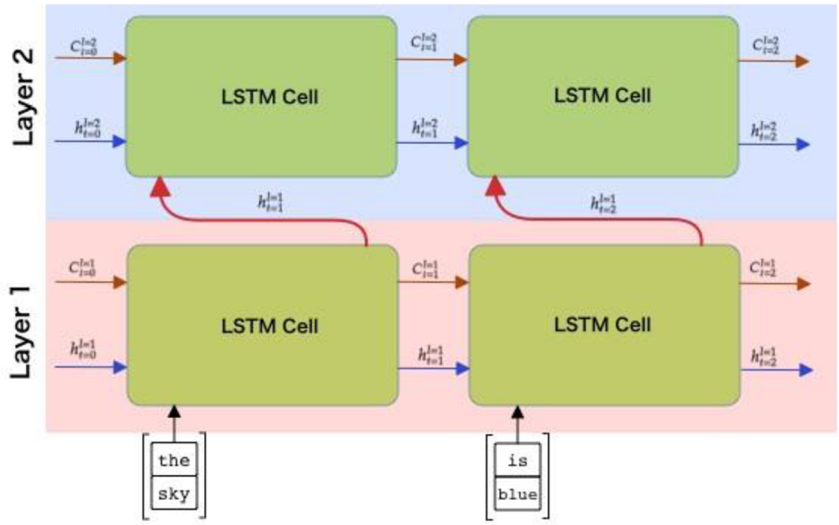

Figure 9: LSTM cells
Source: López (2020)

The sigmoid layer outputs numbers between zero and one, thus describing how much each component must be passed through the valve. Zero is "skip nothing", one is "skip everything". The LSTM has three such valves (Figure 10) to protect and control the cellular state (Shi, 2016).

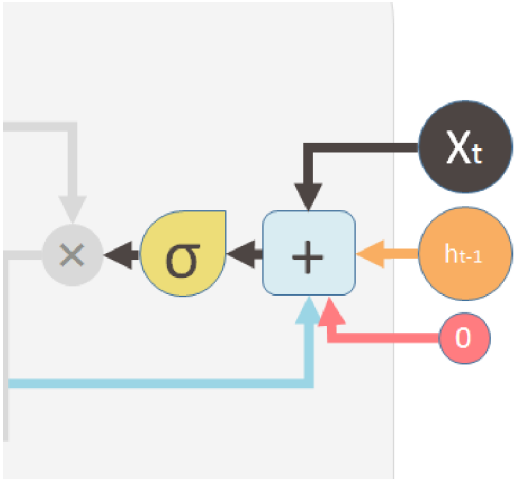

Figure 10: LSTM cell valve
Source: Shi (2016)

3.4.5 Convolutional Neural Networks

Another subtype of neural networks is convolutional networks. There are three processes at the heart of convolutional networks that are necessary to achieve invariance to transfer, scaling, minor distortions, and other image transformations. (Géron, 2017).

The first is local feature extraction. Each neuron receives an input signal from a local receptive field in the previous layer, thus extracting its local features. As soon as the sign is extracted, its exact location no longer matters, since its location relative to other signs is established.

The second is forming network layers as a set of feature maps - each computational layer consists of many feature maps or planes on which all neurons must use the same set of synaptic weights. This form complicates the structure of the network but has two important advantages: invariance to displacement, which is achieved by

convolution with a small nucleus, and a reduction in the number of free parameters, which is achieved by sharing synaptic weights by neurons of the same map.

The third is subsampling - each convolution layer is followed by a computational layer that performs local averaging and subsampling. Due to this, a decrease in resolution for feature maps is achieved. Such an operation leads to a decrease in the sensitivity of the output signal of the sign display operator to slight displacement and other forms of deformation. As such an operator is one of the sigmoidal functions used in the construction of neural networks, for example, hyperbolic tangent.

It should be emphasized that all weights in all layers of the convolutional network are taught by examples. Moreover, the network itself learns to extract signs automatically. Figure 11 shows a convolutional network implementing image recognition (Tabian, Fu, Khodaei, 2019).

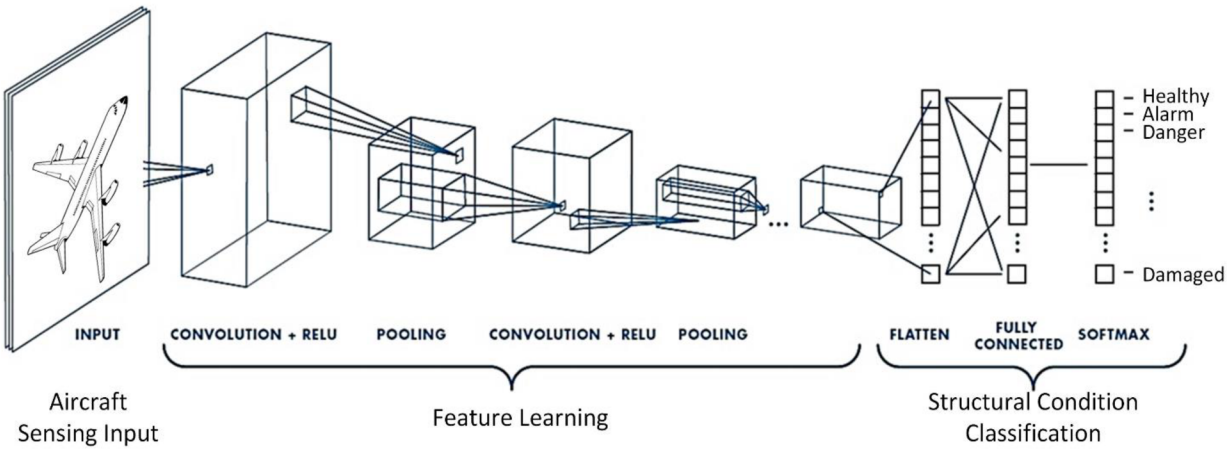

Figure 11*:* Convolutional neural network image recognition
Source: Tabian, Fu, Khodaei (2019)

3.4.6 Generative Adversarial Networks

A generative adversarial network (GAN) is an architecture consisting of a generator and a discriminator configured to work against each other. Hence the GAN was called generative-creative.

One neural network, called a generator, creates new instances of the data, and the other, the discriminator, evaluates them for authenticity. The discriminator decides whether each instance of the data it is considering belongs to the training data set or not. Meanwhile, the generator creates new images, which it transmits to the discriminator. It does this in the hope that they will be accepted as genuine, even if they are fake. The purpose of the generator is to generate images that will be skipped by the discriminator. The purpose of the discriminator is to determine whether the image is genuine. The first generator receives a random number and returns an image, then the generated image is fed into the discriminator along with a stream of images taken from the actual dataset. The discriminator accepts both real and fake images and returns probabilities, numbers from 0 to 1, with 1 representing the genuine image and 0 representing the fake one. A discriminator network is a standard convolutional network that can classify images fed to it using a binomial classifier. The generator is in a sense a reverse convolutional network: although a standard convolutional classifier takes an image and reduces its resolution to obtain a probability, the generator takes a random noise vector and converts it into an image (Figure 12) (Foster, 2019).

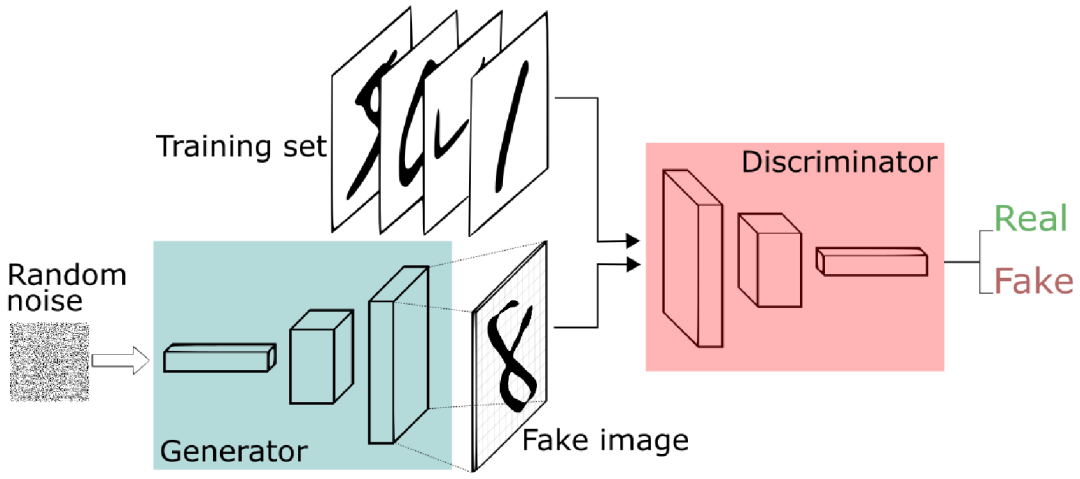

Figure 12: Generative adversarial network image genuineness
Source: Nicholson (2020)

Both networks try to optimize the objective function or loss function. When the discriminator changes its behavior, the generator changes, and vice versa (Nicholson, 2020).

3.4.7 Autoencoders

Autoencoders are neural networks of direct propagation that restore the input signal at the output. Inside, they have a hidden layer, which is the code that describes the model. Autoencoders are designed in such a way that they are not able to accurately copy the input at the output. Usually, they are limited to the dimension of the code (it is less than the dimension of the signal). The input signal is restored with errors due to coding losses, but to minimize them, the network is forced to learn to select the most important features.

Autoencoders consist of two parts: an encoder and a decoder. The encoder translates the input signal into its representation (code) and the decoder recovers the signal by its code (Figure 13) (Foster, 2019).

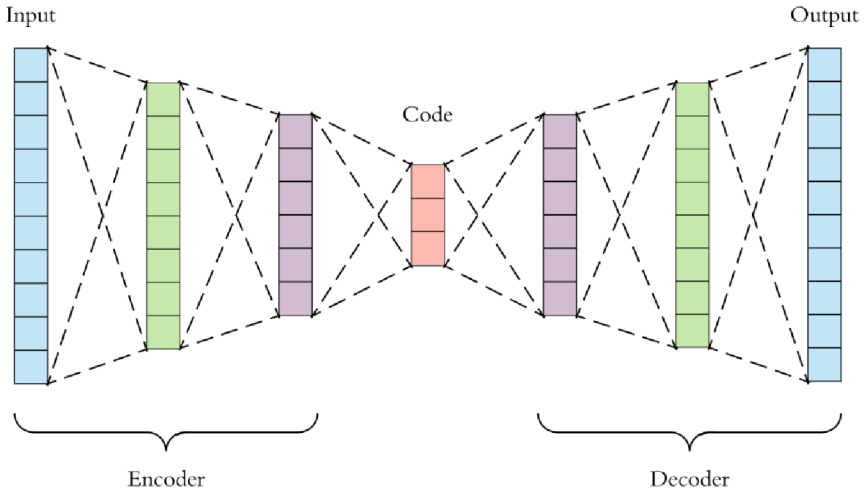

Figure 13: Basic autoencoder architecture
Source: Dertat (2017)

The simplest form of the autoencoder is a direct communicating, non-recurrent neural network, similar to single-layer perceptrons that participate in multilayer perceptrons (MLPs) - having an input layer, an output layer, and one or more hidden layers connecting them - where the output layer has the same number of nodes (neurons) as the input layer, and with the goal of reconstructing its inputs (minimizing the difference between input and output) instead of predicting the

target value of Y based on input X. Thus, autoencoders are unsupervised learning models (not requiring labeled input data to enable learning).

At the same time, the families of functions of the encoder and decoder are limited. Autoencoders are forced to reverse engineer input, keeping only the most important aspects of the data in the copy. By itself, the ability of autoencoders to compress data is rarely used, as they usually perform worse than manually written algorithms for specific types of data, such as sounds or images. And it is also critical for them that the data belongs to the general population on which the network was trained. Once an autoencoder has been trained on numbers, it cannot be used to encode something else (for example, human faces) (Dertat, 2017).

3.4.8 Variational Autoencoders

A variational autoencoder (VAE) was introduced in 2014 by Diederik Kingma and Max Welling. This variant of the autoencoder is capable of solving a variety of tasks, from generating unique melodies and pictures to replacing denoising algorithms and analyzing handwritten texts.

They are probabilistic autoencoders, meaning that their outputs are partly determined by chance, even after training (as opposed to denoising autoencoders, which use randomness only during training), which gives them an edge over RNNs and CNNs since this randomness allows for the generation of unique outputs. Most importantly, they are generative autoencoders, meaning that they can generate new instances that look like they were sampled from the training set.

The basic structure of a variational autoencoders is similar to the basic autoencoders, with an encoder followed by a decoder (Figure 14), but there is a twist: instead of directly producing a coding for a given input, the encoder produces a mean coding μ and a standard deviation σ. The actual coding is then sampled randomly from a Gaussian distribution with mean μ and standard deviation σ. After that the decoder just decodes the sampled coding normally (Foster, 2019).

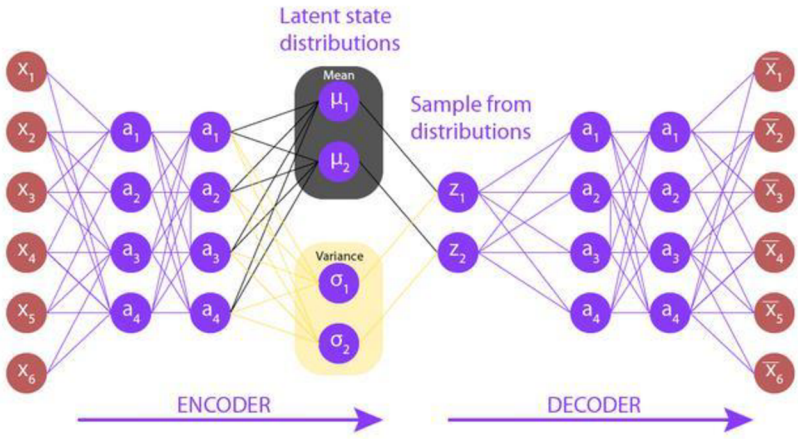

Figure 14: Variational autoencoder architecture
Source: GeeksforGeeks (2022)

Figure 15 shows how a training instance value looks after being mapped on a latent space.

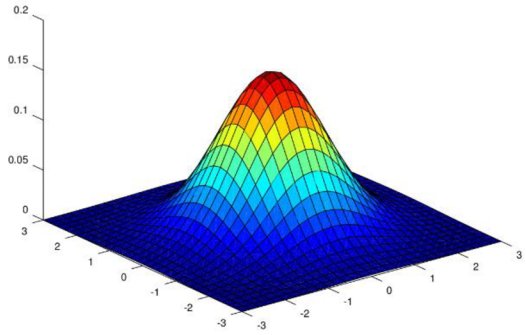

Figure 15: Gaussian value mapping
Source: RInterested (2022)

After enough instances have been mapped the distribution starts to look like a normal or Gaussian distribution (Figure 16).

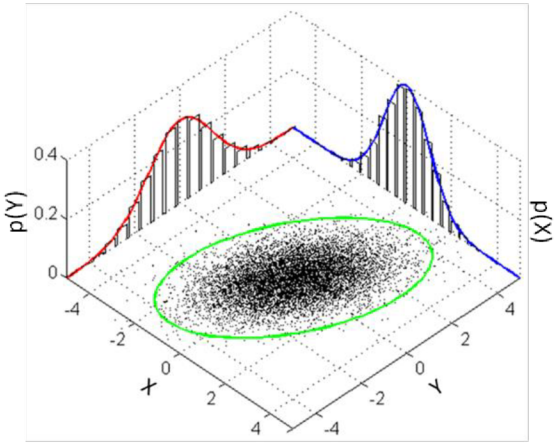

Figure 16: Multivariate Gaussian distribution
Source: Zitao (2020)

Once the mapping is done, the encoder produces μ and σ, then coding is sampled randomly (Figure 17), and finally, this coding is decoded, and the final output resembles the training instance.

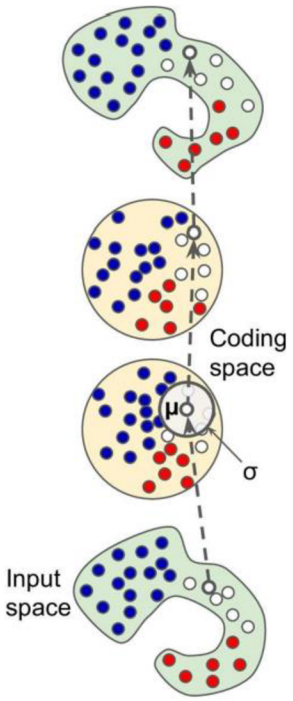

Figure 17: Sampling from a Gaussian distribution
Source: Géron (2017)

The function that describes the random sampling of the coding from the latent space is displayed below:

$$z = \vec{\mu} + \sum \varepsilon$$

$$\Sigma = e^{\frac{log(\vec{\sigma}^2)}{2}}$$

where ε is a sampled point from standard normal distribution

As seen on the Figure 17, although the inputs may have a very convoluted distribution, a variational autoencoder tends to produce codings that look as though they were sampled from a simple Gaussian distribution during training, the cost function pushes the codings to gradually migrate within the coding space (also called the latent space) to occupy a roughly (hyper)spherical region that looks like a cloud of Gaussian points.

The cost function for the VAE is composed of two parts. The first is the usual reconstruction loss that pushes the autoencoder to reproduce its inputs. The second is the latent loss that pushes the autoencoder to have codings that look as though they were sampled from a simple Gaussian distribution, for which the Kullback-Leibler (KL) divergence is used between the target distribution (the Gaussian distribution) and the actual distribution of the codings (Géron, 2017).

$$LOSS = RMSE + KL$$

Where $RMSE$ is reconstruction error,

KL is difference between normal distribution from standard normal distribution

$$D_{KL} = (\mathrm{N}\,(\mu, \sigma)) \,||\, \mathrm{N}\,(0,1)) = \frac{1}{2}\sum(1 + log(\sigma^2) - \mu^2 - \sigma^2)$$

Where N (μ, σ) is normal distribution,

N (0,1) is standard normal distribution,

$\sum$ is sum across all dimensions of the latent space

One great consequence is that after training a variational autoencoder, it is very easy to generate a new instance by sampling a random coding from the Gaussian

distribution and decoding it. Besides that, VAE architecture ensures a quasi-continuous latent space and makes the reconstruction loss small.

4 Existing models

The chapter gives a summary of the most prominent and successful existing projects in the field of music and sound generation with the use of machine learning.

4.1 Modern Models

Although neural networks form the basis of practically all current models for music composition, other methods, such as hidden Markov models for example, are still viable. Although the ability of neural networks to create new combinations from musical sequences that weren't present in the original data is their key advantage over Markov chains. Due to a rapid advancement in the field of machine learning it is now possible to construct a model that generates not only a melody but also an accompaniment to it by picking a chord from a recorded list with the use of LSTM cells, and, in part, for the first time produced a pleasing result (Briot, Hadjeres, Pachet, 2020).

Most of the latest models use the Piano Roll representation or the so-called ABC notation, in which a melody with harmonies is recorded in text form, thus reducing the problem of music generation to the problem of text generation, At the same time there are models that specialize with working with music in MIDI format, where each note can be represented as a step, and each instrument has its own separate track. Some models work with raw audio, either representing it as a spectrogram, and converting the task of music generation into a task of image generation, or simply compressing audio into discrete space, and learning the most important features of the given melody, with the use of VAEs (Müller, 2015).

The major differences between the models are represented in the datasets used, the architecture of the LSTM network, the order in which the chords are generated, and subsequently the resulting melody. It is possible to pre-generate a rhythm or, in a more advanced version, generate a percussion component in addition to chords and melody. These generators are commonly represented as a common network where each layer is one generator. At the stage of generator training, pieces of the same melody from the dataset (true samples) are fed into subsequent layers, while at the

generation stage the "layers" work with the output of the previous ones. A generator that can sample works with a global structure should ideally be able to simulate operations like editing already-created pieces and joining these fragments.

While LSTM networks have demonstrated their ability to create distinctive and generally pleasant sounds and melodies, variational autoencoder networks have received a lot of attention as of lately due to their architecture that enables effective work with latent space and its dimensions.

Many large corporations, including Google and their Magenta team, OpenAI, and many others, are interested in expanding their knowledge and experience in the field of machine learning, including the issues surrounding the creation of music and creativity. The most notable and intriguing projects are discussed below.

4.2 Magenta

Magenta is a research project exploring the role of machine learning in the process of creating art and music. It is one of the most known machine learning teams in the field of music generation. Their primary goals involve developing new deep learning and reinforcement learning algorithms for generating songs, images, drawings, and other materials. As well as an exploration in building smart tools and interfaces that allow artists and musicians to extend their processes using these models (Magenta, 2022).

Over the years they have developed an overwhelming variety of neural network models and were able to connect them into one Magenta environment. Magenta produced a variety of different recurrent neural networks (Figure 18), as well as a variational autoencoder and a generative adversarial network model, all of them can be used either in a command line or using Python development environment (DuBreuil, 2020).

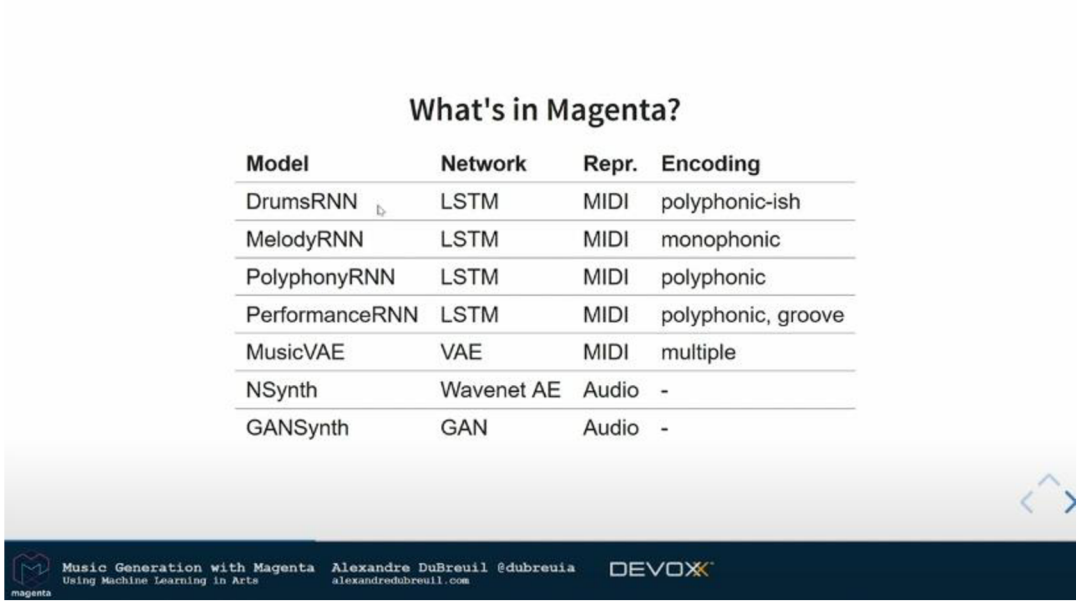

Figure 18: Magenta models overview
Source: Devoxx (2019)

4.2.1 DrumsRNN

This Magenta RNN was created in 2016. The model applies language modeling to drum track generation using an LSTM network. Unlike melodies, drum tracks are polyphonic in the sense that multiple drums can be struck simultaneously. Despite this, a drum track is modeled as a single sequence of events by mapping all the different MIDI drums onto a smaller number of drum classes and representing each event as a single value representing the set of drum classes that are struck. A practical example, that uses DrumsRNN is shown in Figure 19 (Magenta, 2022).

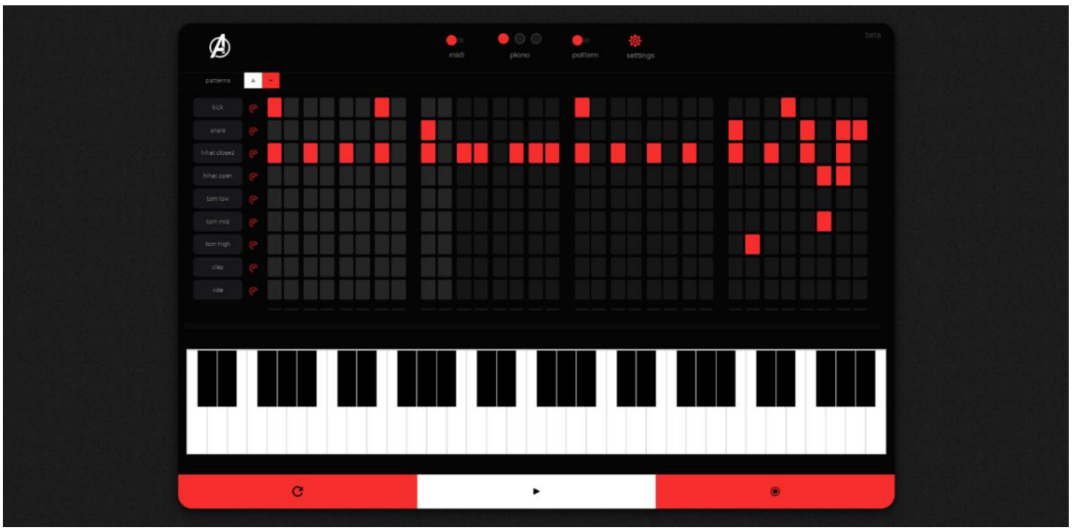

Figure 19: DeepDrum & DeepArp using Google Magenta's DrumsRNN
Source: Magenta (2022)

To understand how an RNN and subsequently DrumsRNN handles sequential data, such as a note sequence, it is necessary to look at the example of an RNN training on broken chords, which are chords broken down as a series of notes. The input data "A", "C", "E", and "G", which is encoded as a vector, for example [1, 0, 0, 0] for the first note, [0, 1, 0, 0] for the second note, and so on. During the first step, with the first input vector, the RNN outputs, for example, a confidence of the next note being 0.5 for "A", 1.8 for "C", -2.5 for "E", and 3.1 for "G". Since the training data shows that the correct next note is "C", it is necessary to increase the confidence score of 1.8 and decrease the other scores. Similarly, for each of the 4 steps (for the 4 input notes), RNN has a correct note to predict. At each step, the RNN uses both the hidden vector and the input vector to make a prediction. During backpropagation, the parameters are nudged in the proper direction by a small amount, and by repeating it enough times, predictions start matching the training data. During inference, if the network first receives an input of "C", it won't necessarily predict "E" because it hasn't seen "A" yet, which doesn't match the example chord that was used to train the model. The RNN prediction is based on its recurrent connection, which keeps track of the context, and doesn't rely on the input alone. To sample from a trained RNN, a note is fed into the network, which outputs the distribution for the next note. By sampling the distribution, a prediction of the next note is produced that can then

be fed back to the network. The process can be repeated until the sequence is long enough (DuBreuil, 2020).

4.2.2 MelodyRNN

The Melody-RNN was designed by the Magenta team in 2016, the model architecture consists of a simple dual-layer LSTM network. An example of an open-source application based on MelodyRNN is shown in Figure 20. It is called A.I. Duet - an interactive experiment that lets the user play a music duet with the computer.

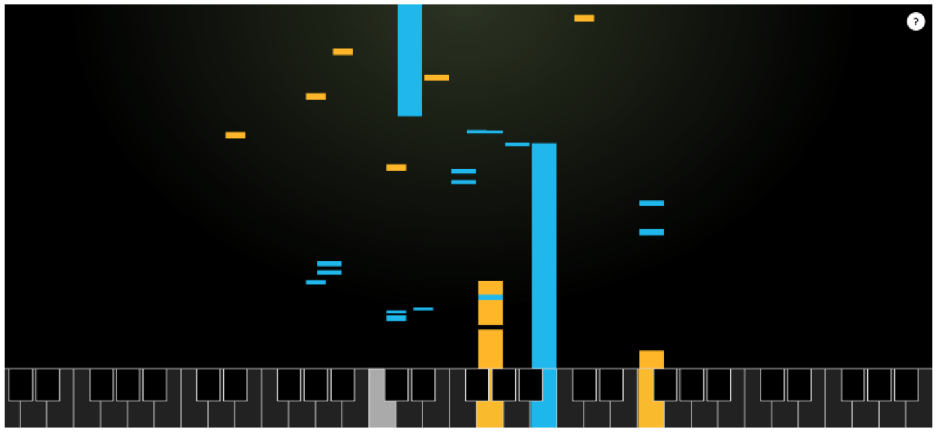

Figure 20: PianoDuo using Google Magenta's MelodyRNN
Source: Magenta (2022)

Currently, there are three types of Melody-RNN models. One is a basic dual-layer LSTM model, which uses basic one-hot encoding to represent extracted melodies as input to the LSTM; the second is Lookback RNN, which introduces custom inputs and labels to allow the model to easily recognize patterns that occur across 1 and 2 bars; the last one is Attention RNN, which introduces the use of attention to allow the model to more easily access past information without having to store that information in the RNN cell's state. The latest update from the Magenta team provided a DQN (Deep Q-Network) network that can also be applied in the Magenta generating process to work as a reward function to teach the neural network to follow certain music theories. The basic idea is represented in Figure 21 (Lou, 2016).

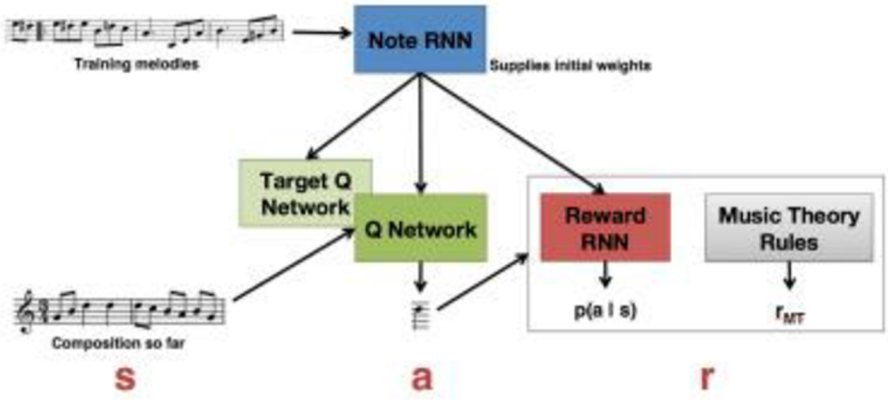

Figure 21: MelodyRNN architecture
Source: Lou (2016)

4.2.3 PerformanceRNN

PerformancRNN is a powerful model with more options and pre-trained models than the DrumsRNN and MelodyRNN. It was presented to the public in 2017, and to this day remains one of the more pleasant models of polyphonic music. Like the rest of the RNN models, PerformanceRNN does not capture the global structure but is fundamentally designed to solve this problem based on the presentation of the data itself. The network also captures and reproduces the frequent combinations of notes played almost simultaneously during generation, sounding like chords to the ear. Thus, such a presentation of data partly "solves" the problem of polyphonic sampling.

The PerformancRNN configuration supports expressive timing, where the notes won't fall exactly on step beginning or end, giving it a more "human" feel or "groove". To go a bit deeper into note timing it is necessary to acknowledge that unlike the previous models, where the output was generated for every time step, and the step size was tied to a fixed meter, in PerformanceRNN both of those conventions were discarded: a time “step” is now a fixed absolute size of 10 milliseconds, and the model can skip forward in time to the next note event. This fine quantization can capture more expressiveness in note timings. And the sequence representation uses many more events in sections with high note density, which matches human performance. Subsequently, the use of a different approach to note tempo leads to

the rejection of grid sampling, leading to the transition from MIDI representation to Piano Roll representation (DuBreuil, 2020).

The model was trained on the Yamaha e-Piano Competition dataset, which contains MIDI captures of around 1400 performances by skilled pianists. The dataset was found useful for learning dynamics (velocities) conditioned on notes. All the pieces were composed for and performed on one single instrument: piano. The model was trained on a repertoire selection from a classical piano competition (Magenta, 2022).

An example of a real-time PerformanceRNN web application implemented with the use of TensorFlow.js is shown in Figure 22.

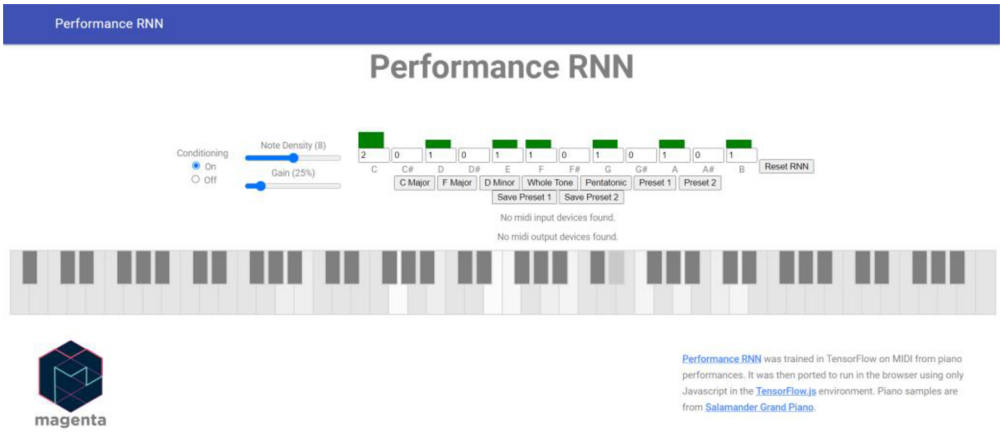

Figure 22: Web application using Google Magenta's PerfromanceRNN
Source: Magenta (2022)

4.2.4 Music Transformer

Music Transformer is yet another model developed by Magenta that uses transformers to perform music generation. It is an open-source machine learning model that can generate long musical performances, around 60 seconds of audio MIDI-files outperforming the coherence that is achieved on the LSTM-based models. Unlike the basic transformer methods, where the attention vectors infer the relationship between tokens in an absolute way, the attention layers in this algorithm use relative attention, which models the relationship between tokens

with the relative distance between them. This allows for a better modeling of periodicity, frequency, and other characteristics of the melodies in the training examples in the short term (DuBreuil, 2020).

Music Transformer has 3 methods it uses to generate music: generating a performance from scratch, generating a melody continuation, generating an accompaniment for a melody. An application using Magenta's Music transformer is displayed in Figure 23, it generates new music sequences and allows users to share them.

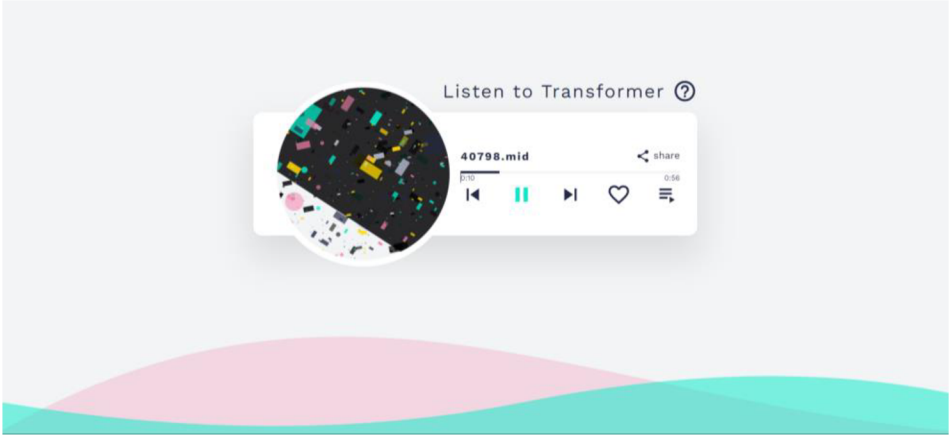

Figure 23: Application using Google Magenta's Music Transformer
Source: Magenta (2022)

The model was trained on a unique data source: piano recordings on YouTube transcribed using Onsets and Frames. Thousands of piano recordings, with a total length of over 10,000 hours were used during the training process. Using such transcriptions allows training symbolic music models on a representation that carries the expressive performance characteristics from the original recordings.

To preprocess the training data the Magenta team used an AudioSet-based model to identify pieces that contained only piano music, this resulted in hundreds of thousands of videos. To train the Transformer model, it was needed for the content to be in a symbolic, MIDI-like form. By using Onsets and Frames the team extracted the audio and processed it using an automatic music transcription model. This

resulted in over 10,000 hours of symbolic piano music that then were used to train the models (Figure 24) (Magenta, 2022).

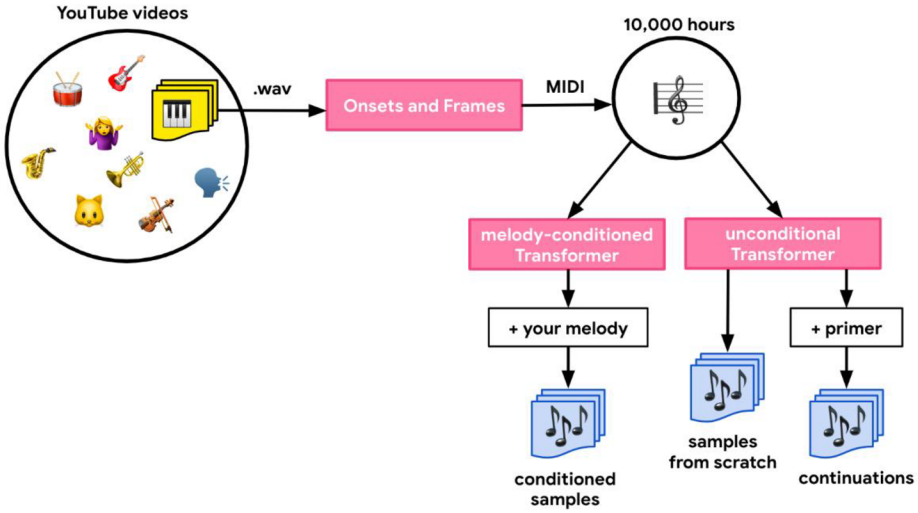

Figure 24: Transformer training data preprocessing
Source: Magenta (2022)

4.2.5 MusicVAE

MusicVAE was created in 2017 by the Magenta team, model uses hierarchical recurrent variational autoencoder architecture, that allows for learning latent spaces for musical scores. Such architecture is widely used in generative models that have yielded state-of-the-art machine learning results in image generation and reinforcement learning. Google researchers had already applied the technique to SketchRNN and have now brought the same infrastructure to MusicVAE. Because musical elements are typically more complicated than sketches engineers developed a novel hierarchical decoder for MusicVAE that can generate long-term structure from individual latent codes.

MusicVAE lets users create palettes for blending and exploring musical scores, it generates and morphs melodies to output multi-instrumental passages optimized for expression, realism and smoothness which sound convincingly like human-composed music. It is used in multiple Magenta demo applications, Figure 25 shows one of them (Magenta, 2022).

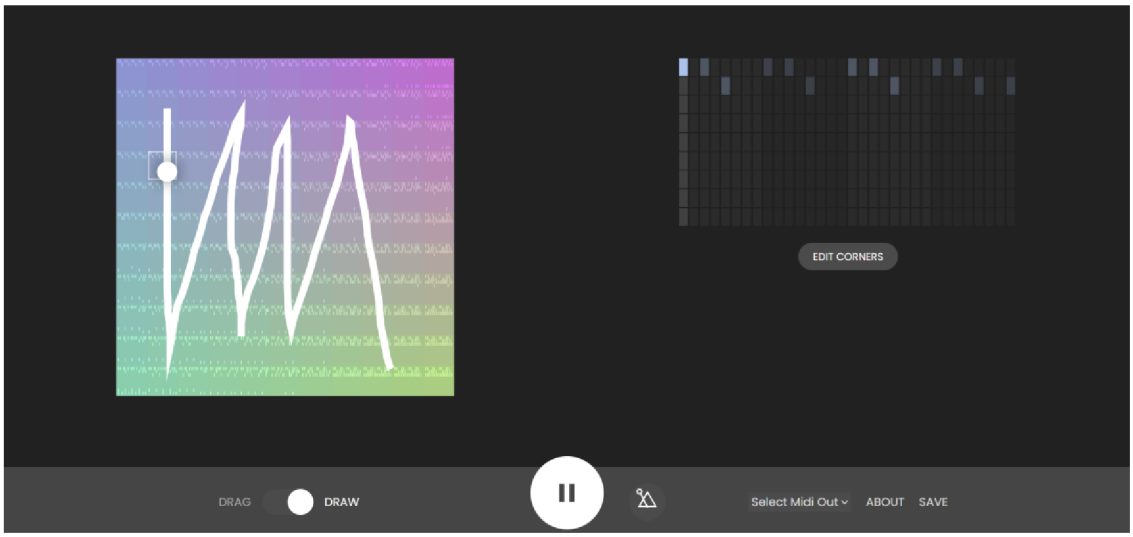

Figure 25: Application using Google Magenta's MusicVAE
Source: Magenta (2022)

4.3 OpenAI

OpenAI is a non-profit artificial intelligence research company created in 2015. Its goal is to advance digital intelligence in the way that is most likely to benefit humanity, unconstrained by a need to generate a financial return. Besides working on general artificial intelligence (AGI) OpenAI has produced a variety of complex artificial neural network models, that work with images such as DALE-2 or music - MuseNet and Jukebox, among the most popular (OpenAI, 2022).

4.3.1 MuseNet

MuseNet is a tool from OpenAI, first demonstrated in 2019, that uses transformers to generate MIDI Files. It is a deep neural network that can generate 4-minute musical compositions with 10 different instruments and can combine styles from country to Mozart to the Beatles. MuseNet was not explicitly programmed with human understanding of music, but instead discovered patterns of harmony, rhythm, and style by learning to predict the next token in hundreds of thousands of MIDI files. MuseNet uses the same general-purpose unsupervised technology as GPT-2, a large-scale transformer model trained to predict the next token in a sequence, whether audio or text. The melodies can also be generated from scratch using a primer melody or as accompaniment for a given melody.

The model uses composer and instrumentation tokens to give more control over the kinds of samples MuseNet generates. During training time, these composer and instrumentation tokens were prepended to each sample, so the model would learn to use this information in making note predictions. At generation time, the model can then be conditioned to create samples in a chosen style by giving it a prompt.

Training data for MuseNet was collected from many different sources. ClassicalArchives and BitMidi donated their large collections of MIDI files for the project, as well as several online collections were used, including jazz, pop, African, Indian, and Arabic styles.

The transformer is trained on sequential data by being given a set of notes and asked to predict the upcoming note. Several different ways to encode the MIDI files into tokens suitable for this task were used. First, a chordwise approach that considered every combination of notes sounding at one time as an individual "chord" and assigned a token to each chord. Second, condensing the musical patterns by only focusing on the starts of notes and tried further compressing that using a byte pair encoding scheme (OpenAI, 2022).

An application using MuseNet is presented in Figure 26.

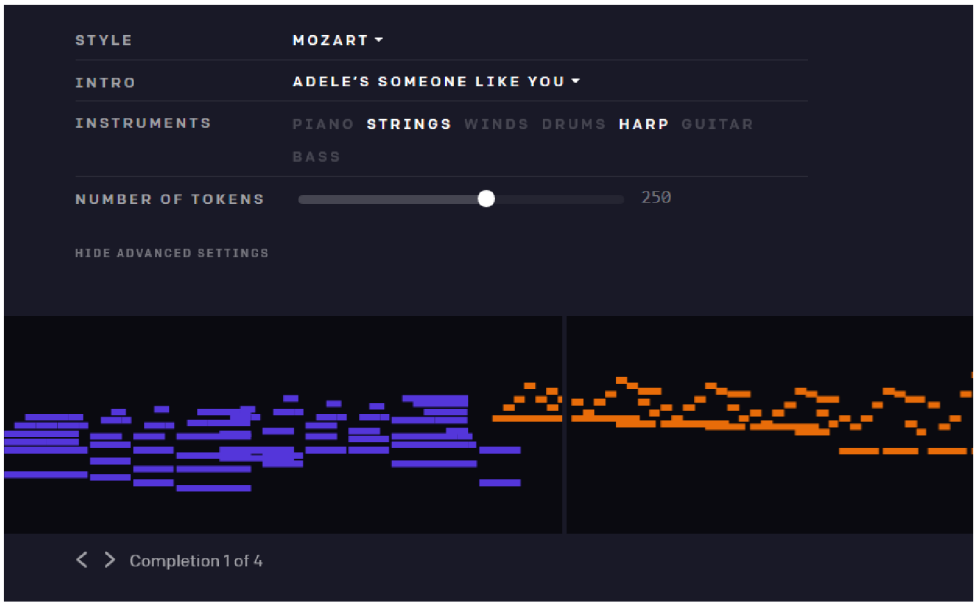

Figure 26: Application using OpenAI's MuseNet
Source: OpenAI (2022)

4.3.2 Jukebox

Jukebox was created in 2020, it is a variational autoencoder network that generates music, including rudimentary singing, as raw audio in a variety of genres and artist styles. Unlike MuseNet it works with music in its raw audio form, not in MIDI format.

The model had to learn to tackle high diversity as well as very long-range structure, and the raw audio domain is particularly unforgiving of errors in short-, medium-, or long-term timing. The model uses the variational autoencoder architecture with some specific configuration.

Jukebox's autoencoder model compresses audio to a discrete space, using a quantization-based approach called VQ-VAE. Hierarchical VQ-VAEs can generate short instrumental pieces from a few sets of instruments, however they suffer from hierarchy collapse due to use of successive encoders coupled with autoregressive decoders. A simplified variant called VQ-VAE-2 avoids these issues by using feedforward encoders and decoders only, and they show impressive results at generating high-fidelity images.

Like the VQ-VAE, VQ-VAE-2 has three levels of priors: a top-level prior that generates the most compressed codes, and two upsampling priors that generate less compressed codes conditioned on above.

Once all the priors are trained, codes can be generated from the top level, upsampled using the upsamplers, and decoded back to the raw audio space using the VQ-VAE decoder to sample novel songs (Figure 27) (OpenAI, 2022).

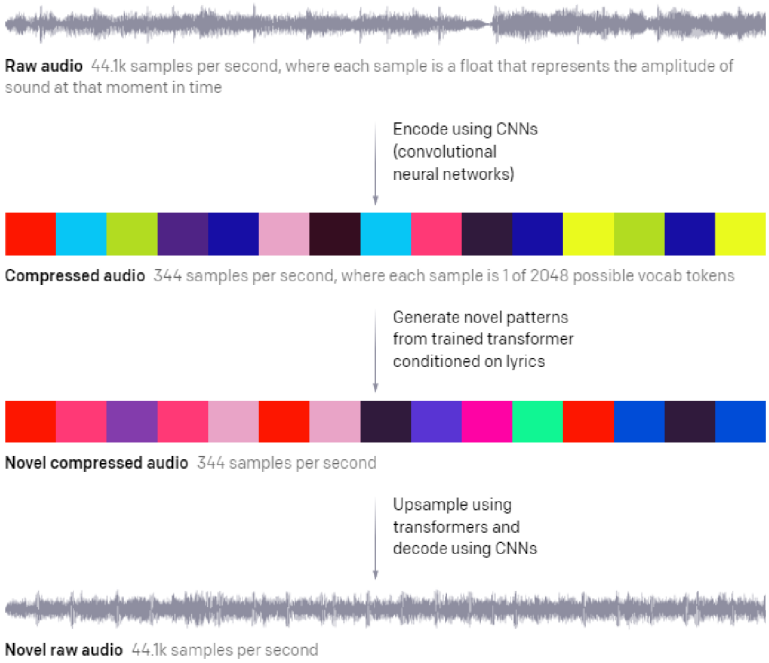

Figure 27: OpenAI's VAE raw audio processing
Source: OpenAI (2022)

4.4 Independent projects

Not all impressive projects are created by using funding from big corporations, a lot of research is done by independent teams of engineers and scientists, as well as smaller AI-oriented companies. Some of the more interesting projects by such teams will be discussed below.

4.4.1 WaveNet

WaveNet was created by researchers at London-based artificial intelligence firm DeepMind in 2016, and currently powers Google Assistant voices. It is a deep generative model of raw audio waveforms. WaveNet can generate speech which mimics any human voice, and which sounds more natural than the best existing Text-to-Speech systems, reducing the gap with human performance by over fifty percent. The same network can be used to synthesize other audio signals such as music and present some striking samples of automatically generated piano pieces.

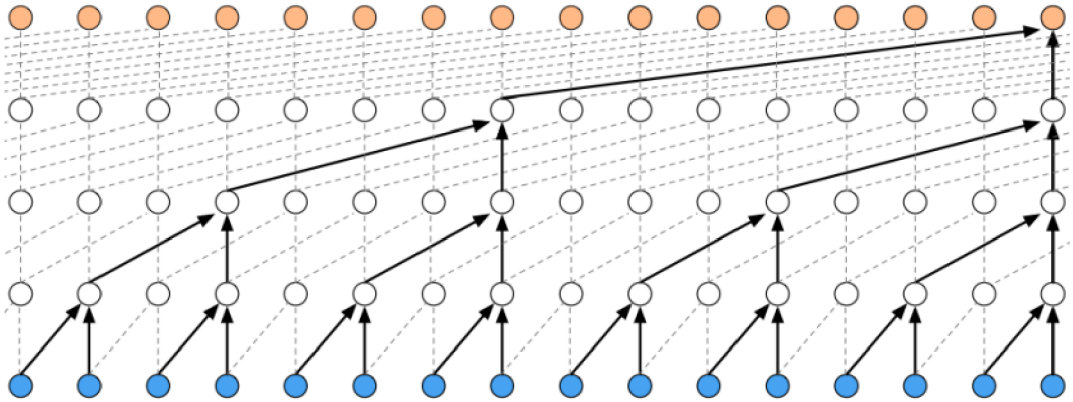

Figure 28: Visualization of a stack of dilated causal convolutional layers
Source: Lou (2016)

Figure 28 shows how a WaveNet is structured. It is a fully convolutional neural network, where the convolutional layers have various dilation factors that allow its receptive field to grow exponentially with depth and cover thousands of timesteps (Lou, 2016).

At training time, the input sequences are real waveforms recorded from human speakers. After training, the network can be sampled to generate synthetic utterances. At each step during sampling a value is drawn from the probability distribution computed by the network. This value is then fed back into the input and a new prediction for the next step is made. Building up samples one step at a time like this is computationally expensive, but it was found it essential for generating complex, realistic-sounding audio (Oord et al., 2016).

CNN architectures, such as WaveNet, have been shown to achieve just as good if not better performance as RNNs in sequence generation. Additionally, they are much faster to train due to performance optimizations with convolutional operations.

Since WaveNet can be used to model any audio signal, it is also able to generate relatively simple music compositions. Unlike the text-to-speech experiments, the model wasn't conditioned on an input sequence telling it what to play (such as a musical score); instead, it simply generates whatever it wants to. After training it on

a dataset of classical piano music, it produced fascinating samples (DeepMind, 2022).

4.4.2 MidiNet

MidiNet is a research project created by Li-Chia Yang, Szu-Yu Chou, Yi-Hsuan Yang in 2017, the result of this project was a CNN model capable of producing novel MIDI tracks. While there already were many deep learning-based music generation models, including WaveNet and MelodyRNN, most of them were using RNN and transformer architectures, as well as their variants. WaveNet was the only other major project that used CNN, and it showed that convolutional neural networks can also generate realistic musical waveforms in the audio domain. One important advantage of training CNN vs RNN is that the former is faster and more easily parallelizable

MidiNet was created precisely with that information in mind, it would use GAN architecture with a CNN-like generator and discriminator, that would be able to learn the distributions of melodies. In the case of MidiNet, the generator is to transform random noises into a 2-D score like representation, that “appears” to be from real MIDI. Meanwhile, the discriminator takes this 2-D score like representation and predicts whether this is real or not. Moreover, the authors of the paper proposed a novel conditional mechanism to exploit available prior knowledge, so that the model can generate melodies either from scratch, by following a chord sequence, or by conditioning on the melody of previous bars (e.g., a priming melody), among other possibilities.

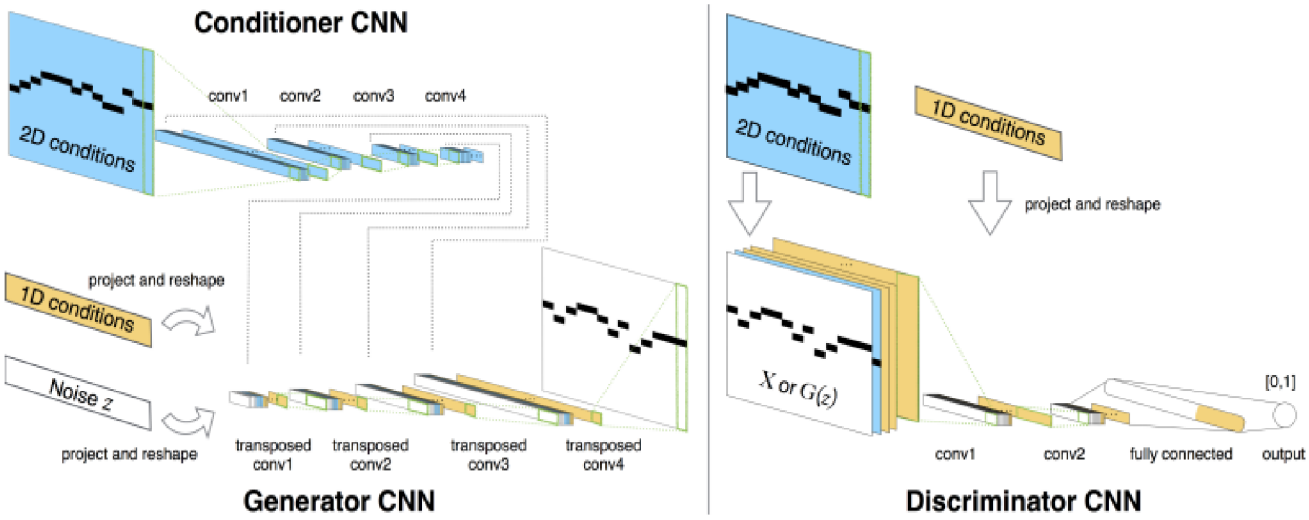

Figure 29: System diagram of the proposed MidiNet model for symbolic-domain music genertion
Source: Yang, Chou, Yang (2017)

The resulting model, named MidiNet, can be expanded to generate music with multiple MIDI channels. A user study was conducted to compare the melody of an eight-bar long generated by MidiNet and by Google's MelodyRNN models, each time using the same priming melody. The result shows that MidiNet performs comparably with MelodyRNN models in being realistic and pleasant to listen to, yet MidiNet's melodies are reported to be much more interesting (Yang, Chou, Yang, 2017).

5 Model proposition

This chapter will describe the structure and specifics of the proposed model, explain the decision behind choosing specific network architecture and training data format and predict some problems that the author might run into while developing the set model.

5.1 Architecture

Based on the architectures and models described above, the author has decided that the most straightforward approach would be to use a variational autoencoder model, based on the performance it has shown in the task of music generation, as well as fact that the chosen architecture would be able to avoid two big issues.

First, VAE architecture would solve the problem of creating human-like sounding melodies, since the reconstructive loss during training would penalize the network for producing any output sounding different from the input, effectively blocking any attempt of the network to produce anything that doesn't follow the patterns and musical rules present in the training data. On the other hand, that makes the choice of the dataset a very important decision since the quality of the used data and its preprocessing will directly impact the quality of the generated songs.

Second, unlike with RNNs and CNNs, where under specific circumstances the resulting model can only create a batch of similarly sounding melodies when for example network converges on outputting only a small subset of common sequences in the training data so that it minimizes the training loss, VAE model can avoid such problem by adding random noise to the encoded latent distribution's mean parameters during training, ensuring that each sample will sound novel.

5.2 Data format

Next question that needs to be addressed is what format of data VAE will work with. There are ultimately two approaches. As presented in section 4, existing models either work with symbolic (MIDI or note) representation of music or raw audio (Müller, 2015). Based on the fact, that nowadays most music production is done with the use of such tools as BandLabs, Cakewalk, Soundtrap and many others, where

each song is represented as a collection of separate MIDI track, with an astonishing number of ways to edit them, the author has decided to choose the output of the model to be represented as a MIDI file, subsequently it makes sense to choose the dataset accordingly, consisting from the preprocessed MIDI tracks.

5.3 Structure

Training VAE on a MIDI formatted dataset alone will not yield results that sound anything like an artist produced music. The best that can be expected is a monophonic melody with a single chosen instrument, such as a piano performance, a guitar solo, etc. And while there are plenty of songs consisting only of one instrument, the vast majority is polyphonic, combining and mixing musical sequences of multiple instruments and voices, all of which come together into one harmonically sounding song.

To make the model output a complex melody, involving multiple instruments, with specific interdependencies between them requires a more complex model structure. One solution to this issue would be to choose one instrument as a lead, and make all other instruments depend on its output. This can be achieved by training a separate VAE network for each instrument and linking their input with the output of one main instrument VAE network, this task can be made simpler by using training data represented in the MIDI format, where each instrument can be isolated as a separate track. By using this approach, the resulting model will be able to retain most important instrumental interdependencies and learn the existing music patterns presented in the dataset.

A practical solution to the method described above would look similar to Figure 30, where piano was chosen as a lead instrument, while the rest of the instruments depend on its output.

First, a new ANN called MelodyNN will be presented to the model structure, it will effectively predict the next step for the piano's track. The latent parameters from the piano previous time step will be fed to the MelodyNN, which will learn a mapping between piano sequences in successive time steps and output the next step for the

piano sequence. Then the new resulting sequence will be decoded back with the use of piano VAE, and subsequently fed further as a new input.

Secondly, multiple ConditionalNNs will be created, that take in the generated next-period piano latent parameters from the MelodyNN output, as well as the previous-period guitar, strings, drums, and bass latent parameters, and will learn a mapping to the next-period instrument latent parameters. Then it will decode it by the instrument-specific VAE's decoder to produce the next-period instrument output.

Finally, at each iteration of the described generative method, it is essential to add some random noise to the latent space of each instrument to increase the variation of the generated output, while maintaining the similarity between the previous sequences, ultimately improving the uniqueness of the generated music (Tham, 2021).

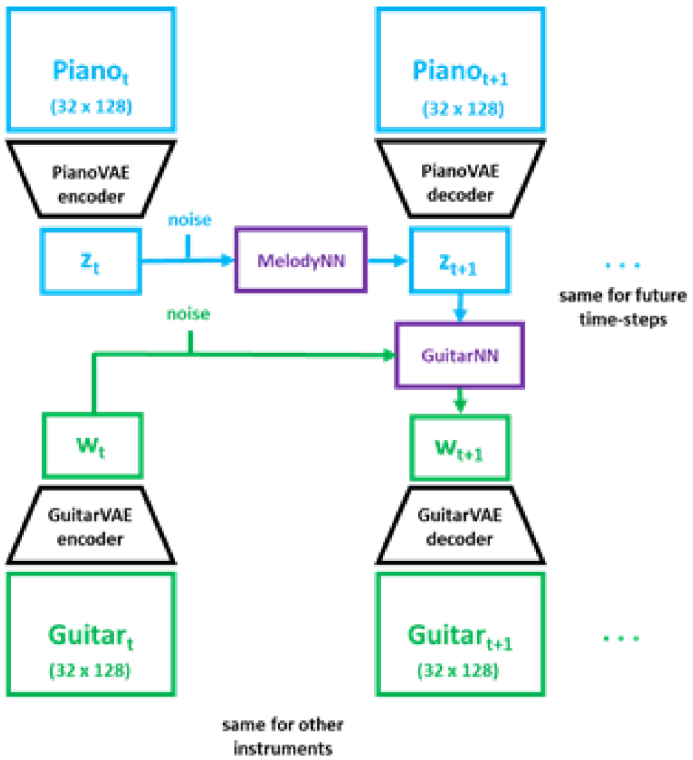

Figure 30: Architecture of VAE-NN used to generate music
Source: Tham (2021)

The method described above has already been presented in practice and has shown impressive results. The author has taken inspiration for the project model structure and implementation from the article on Towards Data Science by Issac Tham (Tham, 2021).

6 Implementation

This chapter will first describe the development environment including the used libraries to help train the VAE model and later generate new songs. The second part of the chapter will showcase the implementation of the model, described in the above chapter, in which the structure and the architecture of the network were specified. Lastly, the chapter will include commentated code snippets written to perform the set task.

6.1 Environment

When choosing a proper programming language for the set task Python seemed like an obvious choice, due to the author's prior experience with the language, as well as an abundance of both available information and established tools such as libraries and packages.

Python version 3.10 was used for the project. Implementation required a list of specific libraries, capable of working with artificial neural network models, song datasets, files of different formats, as well as some general-purpose packages for calculations. The following sections will describe the tools and libraries that were used in the work.

6.1.1 NumPy

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more (Harris et al., 2020).

6.1.2 Keras

Keras is a deep learning API written in Python, running on top of the machine learning platform TensorFlow. It contains fully configurable standalone modules

that connect to each other to form a model. As individual modules, there are types of layers of artificial neural networks, activation functions, and others. In addition, each module contains a significant number of parameters that can be modified as needed (Chollet et al., 2015).

6.1.3 TensorFlow

TensorFlow is an open-source library for numerical computation and large-scale machine learning, acquiring data, training models, serving predictions, and refining future results. TensorFlow bundles together Machine Learning and Deep Learning models and algorithms. In implementation, it is mainly used as a basis for Keras, who is responsible for low-level operations for optimized vector manipulation (Abadi et al., 2015).

6.1.4 PyTorch

PyTorch is a fully featured framework for building deep learning models, which is a type of machine learning that's commonly used in applications like image recognition and language processing. Written in Python, it's relatively easy for most machine learning developers to learn and use. PyTorch is distinctive for its excellent support for GPUs and its use of reverse-mode auto-differentiation, which enables computation graphs to be modified on the fly. It is the main package used to work with the artificial neural network model in this project. It allows a more efficient training of the model, with the use of GPU-provided support (Paszke et al., 2019).

6.1.5 Pypianoroll

Pypianoroll is an open-source Python library for working with piano rolls. It provides essential tools for handling multitrack piano rolls, including efficient I/O as well as manipulation, visualization, and evaluation tools. The author of this project uses this package to decompose existing melodies and then generate new multitrack MIDI format songs (Dong, Hsiao, Yang, 2018).

6.1.6 hdf5_getters

A set of get methods that work with the Million Song Dataset. Allows users to access various metadata from the dataset records, such as song genre, artist tags, etc. In this project, it is used to work with the HDF5 files, sort and filter them based on the specific information required (Tbertinmahieux, 2010).

6.2 Dataset

The chosen dataset came from the Lakh Pianoroll Dataset (LPD), it is a collection of 174,154 multitrack pianorolls derived from the Lakh MIDI Dataset and was curated by the Music and AI Lab at the Research Center for IT Innovation, Academia Sinica. The multitrack pianorolls in LPD are stored in a special format for efficient I/O and to save space.

LPD offers a choice of two different datasets: LPD-5 and LPD-17. The difference is that in LPD-5, the tracks are merged into five common categories: Drums, Piano, Guitar, Bass, and Strings according to the program numbers provided in the MIDI files. While In LPD-17, the tracks are merged into drums and sixteen instrument families according to the program numbers provided in the MIDI files and the specification of General MIDI Level 1. The seventeen tracks are Drums, Piano, Chromatic, Percussion, Organ, Guitar, Bass, Strings, Ensemble, Brass, Reed, Pipe, Synth Lead, Synth Pad, Synth Effects, Ethnic, Percussive, and Sound Effects (Dong et al., 2018) (Colin, 2016).

LPD-5 was chosen as the more adequate dataset version, since it already provides quite enough complexity for the chosen model, while allowing to generate complex and rich music and to demonstrate the ability of the generative models to arrange music across different instruments.

Besides that, a few JSON files from the Million Song Dataset containing genre and artist metadata were used to make the process of selecting the subset of desired training songs a little easier (Bertin-Mahieux et al., 2011).

6.3 Application

Project implementation consists of solving multiple tasks one by one. First, it is necessary to preprocess the data in the dataset, so that it is represented in the appropriate format for the VAE model to train on. Second, it is required to create dataset and dataloader classes, that would directly feed the training and testing data to the VAE. The third step involves specifying the structure and creating the VAE classes themselves and training them in the next step. Then the same process is repeated for the MelodyNN and ConditionalNNs. And finally, once all the networks in the model are created and trained, it is time to use them to generate new songs.

6.3.1 Training data preprocessing

The first step is to prepare the training data, the chosen Lakh Pianoroll Dataset version consists of tens of thousands of songs, that have different genres and artists. With the use of the hdf5_getters methods, it is possible to work with the song metadata and retrieve important information without manually going through each file and looking for the information.

As the author's personal preference, it was decided to create a subset of the existing dataset, which would include songs either in the genre of blues or rock, making the resulting generated songs lean in the direction of such genres (Figure 31).

```
def get_all_blues_rock_titles(basedir, ext='.h5'):
  ids_to_add = []
  for root, dirs, files in os.wSalk(basedir):
    files = glob.glob(os.path.join(root, '*' + ext))
    for f in files:
      h5 = hdf5_getters.open_h5_file_read(f)
      for genre in hdf5_getters.get_artist_terms(h5):
        if genre.decode("utf-8") == 'blues' or genre.decode("utf-8")
== 'rock':
          with open(genre_ids) as f:
            s = mmap.mmap(f.fileno(), 0, access=mmap.ACCESS_READ)
            if s.find(hdf5_getters.get_track_id(h5)) !=-1:
              with open (training_ids, 'a') as file_object:
                file_object.write("\n")

file_object.write(hdf5_getters.get_track_id(h5).decode("utf-8"))
          break
```

```
      h5.close()
  return
```

Figure 31: Blues and rock songs subset creation
Source: Personal screenshot

The resulting subset consists of roughly 12000 songs, and once the song IDs are written into a new file, the last remaining step is to simply choose an even smaller subset (around 1000 songs) that will be used in each iteration (epoch) of training.

Once the random songs are chosen it is time to parse each song into a separate instrument track, effectively representing each file as a list of notes found in the file (Figure 32). It is then possible to create the training input sequences by taking subsets of the list representation for each song and arranging the corresponding training output sequences by simply taking the next note of each subset. With this training input and output, the model will be trained to predict the next note, which will then allow it to pass in any sequence of notes and get a prediction of the next note.

```
parts = {'piano_part': None, 'guitar_part': None, 'bass_part': None,
'strings_part': None, 'drums_part': None}
  song_length = None
  empty_array = None
  has_empty_parts = False
  for track in multitrack.tracks:
    if track.name == 'Drums':
      parts['drums_part'] = track.pianoroll
    if track.name == 'Piano':
      parts['piano_part'] = track.pianoroll
    if track.name == 'Guitar':
      parts['guitar_part'] = track.pianoroll
    if track.name == 'Bass':
      parts['bass_part'] = track.pianoroll
    if track.name == 'Strings':
      parts['strings_part'] = track.pianoroll
    if track.pianoroll.shape[0] > 0:
      empty_array = np.zeros_like(track.pianoroll)

  for k, v in parts.items():
    if v.shape[0] == 0:
      parts[k] = empty_array.copy()
      has_empty_parts = True

  combined_pianoroll = torch.tensor(
```

```
    [parts['piano_part'], parts['guitar_part'], parts['bass_part'],
parts['strings_part'], parts['drums_part']])
```

Figure 32: Decoding songs into instrumental tracks
Source: Personal screenshot

6.3.2 Dataloader

Once the suitable dataset is collected it is still required to assemble data in a format that is appropriate for model training which means creating an object called 'Dataloaders'. The data format requires code that can read training data into memory, convert the data to PyTorch tensors, and serve the data up in batches. Each type of ANN requires a separate data loader.

CombinedDataloader shown in Figure 33, is used for feeding training data to instruments VAE (piano, guitar, etc.). Additional dataloaders were created for both ConditinalNNs and MelodyNN.

```
class CombinedDataloader(Dataset):
  def __init__(self, pianorolls, instrument_id):
    self.data = pianorolls
    self.length = int(pianorolls.size(1) / 32)
    self.instrument_id = instrument_id

  def __getitem__(self, index):
    sequence = self.data[self.instrument_id, (index * 32):((index + 1)
* 32), :]
    return sequence

  def __len__(self):
    return self.length
```

Figure 33: CombinedDataloader
Source: Personal screenshot

6.3.3 VAE

As was specified above each instrument will be represented with the use of VAE architecture. The internal structure of VAE consists of an encoder (Figure 34) and a decoder (Figure 35), since VAE is a symmetrical architecture both parts are a mirror image of each other.

```
# Define the recognition model (encoder or e) part
self.e_conv_1 = nn.Conv2d(in_channels=1, out_channels=64,
```

```
kernel_size=(4, 4), stride=(4, 4))
self.e_conv_2 = nn.Conv2d(in_channels=64, out_channels=128,
kernel_size=(4, 4), stride=(4, 4))
self.e_conv_3 = nn.Conv2d(in_channels=128, out_channels=256,
kernel_size=(2, 8), stride=(2, 8))
self.e_fc_phi = nn.Linear(256, K + 1)
```

Figure 34: Encoder implementation
Source: Personal screenshot

```
# Define the generative model (decoder or d) part
self.d_fc_upsample = nn.Linear(K, 256)
self.d_deconv_1 = nn.ConvTranspose2d(in_channels=256,
out_channels=128, kernel_size=(2, 8), stride=(2, 8))
self.d_deconv_2 = nn.ConvTranspose2d(in_channels=128, out_channels=64,
kernel_size=(4, 4), stride=(4, 4))
self.d_deconv_3 = nn.ConvTranspose2d(in_channels=64, out_channels=1,
kernel_size=(4, 4), stride=(4, 4))
```

Figure 35: Decoder implementation
Source: Personal screenshot

6.3.4 VAE Training

After the VAE classes are created it is time to start training them on the dataset. Next code snapshot snows the training loop for each of the 5 instruments. Since the music samples are relatively sparse in music space, it was decided to train each instrument in the 16-dimensional latent space. Then Instrument is fed a combined dataset, and once the training is done the resulting model is saved (Figure 36).

```
for K in [16]:
  instruments = ['piano', 'guitar', 'bass', 'strings', 'drums']
  for i in range(5):
    print(K, instrument)
    dataset = CombinedDataset(combined_pianorolls, instrument_id=i)
    piano_loader = DataLoader(dataset, batch_size=32, drop_last=True)

    vae = ConvVAE(K=K)
    elbo_vals = train_vae(vae, piano_loader, epochs=25)
    model_name = 'VAE_{}_{}'.format(instruments[i], K)
    save_path = os.path.join(root_dir, model_path, model_name)
    torch.save(vae.state_dict(), save_path)
```

Figure 36: VAE training
Source: Personal screenshot

6.3.5 MelodyNN and ConditionalNN

Once the training is done, the result is 5 VAEs for each instrument. At the current moment, it only gives the ability to generate separate tracks for each instrument, and while it is possible to run all of them in a MIDI editor at the same time, the resulting melody sounds incoherent and random. It is because generated tracks do not depend on each other in any way. This problem can be solved by introducing a new batch of ANNs that will "glue" each single instrument track into one harmonious sequence.

First is MelodyNN (Figure 37), its internal structure represents a Multi-Layer Perceptron that learns a mapping from the previous piano sequence's latent distribution to the next piano sequence's latent distribution. Its output is then decoded by piano VAE to become generated next piano output.

```
class MelodyNN(nn.Module):
  def __init__(self, K):
    super(MelodyNN, self).__init__()
    self.fc1 = nn.Linear(K, 128)
    self.fc2 = nn.Linear(128, 128)
    self.fc3 = nn.Linear(128, K)
    self.dropout = nn.Dropout(0.2)

  def forward(self, x):
    x = F.relu(self.fc1(x))
    x = self.dropout(x)
    x = F.relu(self.fc2(x))
    out = self.fc3(x)
    return out
```

Figure 37: MelodyNN structure
Source: Personal screenshot

Second is ConditionalNN (Figure 38), another MLP that takes in the generated next-period piano latent parameters as well as the previous-period non-piano instrument latent parameters and learns a mapping to the next-period guitar latent parameters. The output is then decoded by the instrument-specific VAE's decoder to produce the next-period instrument output. 4 ConditionalNNs are trained, one for each non-piano instrument, which allows the next 5-instrument sequence to be generated.

```
class ConditionalNN(nn.Module):
  def __init__(self, K):
    super(ConditionalNN, self).__init__()

    self.fc1 = nn.Linear(2 * K, 128)
    self.fc2 = nn.Linear(128, K)

  def forward(self, prev_harmony, melody):
    x = torch.cat((prev_harmony, melody), axis=1)
    x = F.relu(self.fc1(x))
    out = self.fc2(x)
    return out
```

Figure 38: ConditionalNN structure
Source: Personal screenshot

6.3.6 MelodyNN and ConditionalNN Training

The use of PyTorch and its optimization tools, as well as having trained instrument VAEs allows to start training MelodyNN and ConditionalNNs. The biggest difference in training these ANNs is the fact that instead of feeding them time steps of a specific instrument and expecting them to predict the next one, they will be fed with the sample from the latent distribution of a given instrument VAE. In the case of MelodyNN it will be piano VAE (Figure 39).

```
melody_nn = MelodyNN(K=K).to(device)
optimizer = torch.optim.Adam(melody_nn.parameters(), lr=lr)
scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer,
lr_lambda=lambda epoch: lr_lambda ** epoch)
criterion = nn.MSELoss()
train_losses, test_losses = training_loop_MelodyNN(piano_vae,
melody_nn, optimizer, scheduler, criterion,
                          melody_train_loader, melody_test_loader,
n_epochs=n_epochs)
```

Figure 39: MelodyNN training
Source: Personal screenshot

While ConditionalNN will be trained on both the sample from the latent distribution of the instrument-specific VAE's and the sample from the MelodyNN output. (Figure 40).

```
for instrument in ['guitar', 'bass', 'strings', 'drums']:
  print(instrument)

  cond_train_dataset = ConditionalDataset(pianorolls_list,
dataset_length=32 * 8000, seq_length=32,
```

```
                        instrument=instrument)
  cond_train_loader = DataLoader(cond_train_dataset, batch_size=32,
drop_last=True)
  cond_test_dataset = ConditionalDataset(pianorolls_list[0:500],
dataset_length=32 * 1000, seq_length=32,
                        instrument=instrument)
  cond_test_loader = DataLoader(cond_test_dataset, batch_size=32,
drop_last=True)

  load_model()
  conditional_nn = ConditionalNN(K=K).to(device)
  optimizer = torch.optim.Adam(conditional_nn.parameters(), lr=lr)
  scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer,
lr_lambda=lambda epoch: lr_lambda ** epoch)
  criterion = nn.MSELoss()
  train_losses, test_losses = training_loop_VAENN(piano_vae,
harmony_vae, conditional_nn, optimizer, scheduler,
                          criterion, cond_train_loader,
cond_test_loader, n_epochs=n_epochs)
```

Figure 40: ConditionalNNs training
Source: Personal screenshot

6.3.7 Generating music

With all the ANNs created and trained it is finally time to generate new songs. The method for generating music is described in Figure 41. Each instrument track will be represented as a matrix, containing information about the placement and intensity of every note.

```
def generate_music_vae(sample, vae_models, nn_models, noise_sd=0,
threshold=0.3, binarize=True):
  piano_vae, guitar_vae, bass_vae, strings_vae, drums_vae = vae_models
  melody_nn, guitar_nn, bass_nn, strings_nn, drums_nn = nn_models

  piano, guitar, bass, strings, drums = sample[0, :, :], sample[1, :,
:], sample[2, :, :], sample[3, :, :], sample[4,
:, :]
```

Figure 41: Instrumental tracks declaration
Source: Personal screenshot

Once every instrument has its track, each instrument VAE is called convert to all parts from image space to latent space, allowing the MelodyNN and CoditionalNNs to work with latent space (Figure 42).

```
  piano_latent = piano_vae.infer(piano.unsqueeze(0).to(device))[:, :-
1]
  guitar_latent = guitar_vae.infer(guitar.unsqueeze(0).to(device))[:,
```

```
:-1]
  bass_latent = bass_vae.infer(bass.unsqueeze(0).to(device))[:, :-1]
  strings_latent =
strings_vae.infer(strings.unsqueeze(0).to(device))[:, :-1]
  drums_latent = drums_vae.infer(drums.unsqueeze(0).to(device))[:, :-
1]
```

Figure 42: Conversion to latent space
Source: Personal screenshot

The next time step of the piano will be produced by feeding a previous piano VAE step to the MelodyNN and adding some random noise to it (Figure 43).

```
  piano_next_latent = melody_nn(piano_latent)
  random_noise = torch.randn_like(piano_next_latent) * noise_sd
  piano_next_latent = piano_next_latent + random_noise
```

Figure 43: Generating next piano step
Source: Personal screenshot

Once the model produces a new piano step, ConditionalNNs will be used to predict the next steps for each instrument, random noise will be added to each subsequent prediction (Figure 44).

```
guitar_next_latent = guitar_nn(guitar_latent, piano_next_latent) +
torch.randn_like(piano_next_latent) * noise_sd
  bass_next_latent = bass_nn(bass_latent, piano_next_latent) +
torch.randn_like(piano_next_latent) * noise_sd
  strings_next_latent = strings_nn(strings_latent, piano_next_latent)
+ torch.randn_like(piano_next_latent) * noise_sd
  drums_next_latent = drums_nn(drums_latent, piano_next_latent) +
torch.randn_like(piano_next_latent) * noise_sd
```

Figure 44: Generating next instrument step and adding random noise
Source: Personal screenshot

Finally, the resulting steps are decoded by the instrument-specific VAE (Figure 45).

```
piano_next =
piano_vae.generate(piano_next_latent.unsqueeze(0)).view(1, 32, 128)
  guitar_next =
guitar_vae.generate(guitar_next_latent.unsqueeze(0)).view(1, 32, 128)
  bass_next = bass_vae.generate(bass_next_latent.unsqueeze(0)).view(1,
32, 128)
  strings_next =
strings_vae.generate(strings_next_latent.unsqueeze(0)).view(1, 32,
128)
  drums_next =
drums_vae.generate(drums_next_latent.unsqueeze(0)).view(1, 32, 128)
```

```
  creation = torch.cat((piano_next, guitar_next, bass_next,
strings_next, drums_next), dim=0)
  creation[creation < threshold] = 0

  return creation
```

Figure 45: Decoding resulting sequence
Source: Personal screenshot

To create a complete song the generation method is called iteratively, predicting a new step, and gradually adding it to the result of the previous iteration (Figure 46).

The generation method allows to specify the number of prediction steps, as well as the amount of random noise, that will be added to each newly generated latent, making the entire song more variation filled.

```
for i in range(1, prediction_steps + 1):
  sample = generate_music_vae(sample, vae_models, nn_models,
noise_sd=1, threshold=0.3, binarize=True)
  generated_track[:, 32 * i:32 * (i + 1), :] = sample

generated_track_out = generated_track * 127
piano_track = pypianoroll.StandardTrack(name='Piano', program=0,
is_drum=False,
                      pianoroll=generated_track_out[0, :,
:].detach().cpu().numpy())
guitar_track = pypianoroll.StandardTrack(name='Guitar', program=24,
is_drum=False,
                       pianoroll=generated_track_out[1, :,
:].detach().cpu().numpy())
bass_track = pypianoroll.StandardTrack(name='Bass', program=32,
is_drum=False,
                      pianoroll=generated_track_out[2, :,
:].cpu().detach().numpy())
strings_track = pypianoroll.StandardTrack(name='Strings', program=48,
is_drum=False,
                       pianoroll=generated_track_out[3, :,
:].cpu().detach().numpy())
drums_track = pypianoroll.StandardTrack(name='Drums', is_drum=True,
                      pianoroll=generated_track_out[4, :,
:].cpu().detach().numpy())
generated_multitrack = pypianoroll.Multitrack(name='Generated',
resolution=2,
                        tracks=[piano_track, guitar_track, bass_track,
strings_track,
                                drums_track])
```

Figure 46: Iterative call for sequence generation
Source: Personal screenshot

The resulting song is represented as a MIDI format file, with 5 tracks for each corresponding instrument. (Figure 47).

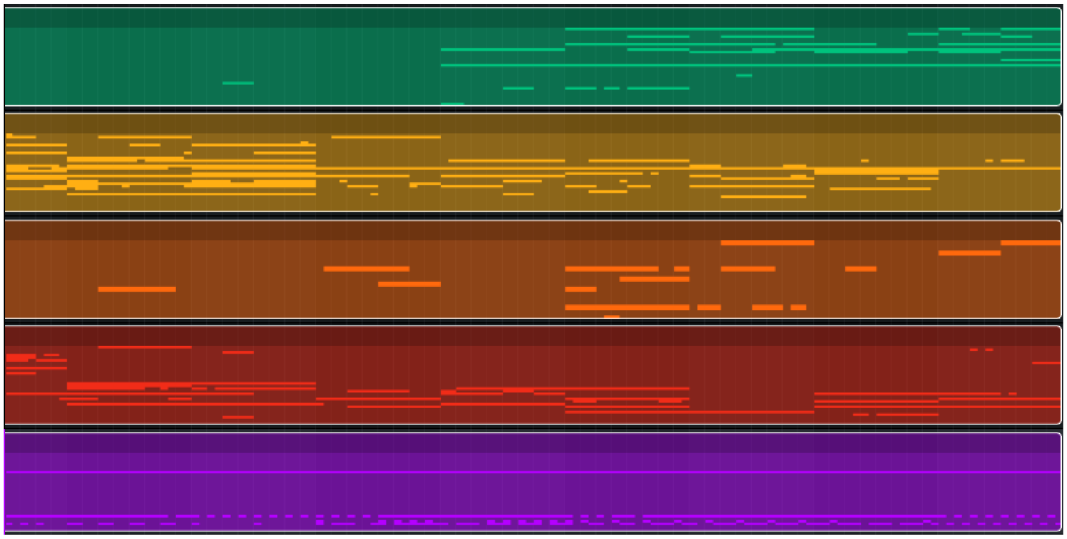

Figure 47: Resulting MIDI song represented in BandLabs
Source: Personal screenshot

7 Results

This chapter aims to present the results of the practical part of this project, describe what the created model has been able to achieve, what are its biggest flaws and advantages, and compare it to the models used today.

7.1 Results overview

The VAE-based architecture that was devised has been able to perform its main set task, it was able to “compose” a novel piece of music, that sounds close to a human-created melody. While not without its drawbacks, the chosen architecture and structure of the model seem to show promise. Each song shows good variety, usually presenting multiple changes in paste and intensity, on the other hand occasionally certain portions of the song can sound out of place, often when the model receives unexpected notes as a prediction input. Sometimes it tries to "use" it and make a transition to a different instrumental key, but such moments occur at a rather random time. Such cases might be caused by the unpredictability of random noise that’s being added for each predicted step. One more observed behavior is that the model can retain some long-term memory of the composition parts, especially with the drums, using the same or similar sounding batches of notes throughout the song, almost like it is trying to recreate a musical pattern consisting of a verse and a chorus. One of the bigger issues involves the occasional abandoning of the entire instrument, where the model only predicts five to ten steps for a certain instrument, most often such behavior is seen with the guitar. A possible case for it might be the specifics of the dataset, where some songs had a minimum amount of certain instrument notes.

The chosen output data format allows the resulting song to be easily imported into a MIDI editing software, such as BandLabs for example, where it is possible to manipulate and edit the existing instrument tracks by adding some post effects or changing the instrument itself, from piano to an 8-string guitar, or from a drum kit to a church choir.

Results are available on: https://on.soundcloud.com/8BgCR

7.2 Model comparison

To compare the resulted model to the state-of-the-art models, the author used Magenta's Multitrack MusicVAE as well as OpenAI's Jukebox pre-trained models.

7.2.1 Multitrack MusicVAE

Depending on the configuration MusicVAE can perform a variety of tasks. The model allows users to generate new melodies, extend a given melody or a drum pattern, "humanize" the melody by giving it human-like timing and velocity to drum parts, it can combine features of the given inputs to create musical transitions between phrases and finally it can turn any sequence into an accompanying drum performance (Magenta, 2022).

For the sake of this comparison, only the generative ability of the Multirack MuiscVAE model was looked at. In this case MusicVAE framework is applied to single measures of multi-instrument General MIDI tracks. It is capable of encoding and decoding single measures of up to 8 tracks, optionally conditioned on an underlying chord. Encoding transforms a single measure into a vector in a latent space, and decoding transforms a latent vector back into a measure. Both encoding and decoding are performed hierarchically, with one level operating on tracks and another operating on the notes in each track.

Multitrack MusicVAE implementation offers multiple options for controlling the generation process of each song. First, like many other Magenta models it uses a sampling temperature option – it determines the creativity and energy of the model. The higher the temperature, the more chaotic and intense the result will be. One more notable option is chord-conditioning, effectively it forces the model to build the resulting melody around the one or multiple chords that user provides. Finally, Magenta team worked on training Multitrack MusicVAE on dozens of different styles, and in turn it allowed for style bending, effectively giving an option to change the melody style in the middle of the composition.

Magenta's model can generate melodies consisting of up to 8 different instruments out of 128 available in General MIDI format, it is trillions of possible instrument combinations. And with the addition of chords to the generation process, the resulting melodies, subjectively, sound very impressive.

While the representation used by the Multitrack MusicVAE is intended to be quite general, there are still a few restrictions. Each measure must contain 8 or fewer tracks (in the case of the implemented model it is 5) and must have a 4/4-time signature.

As was observed earlier, General MIDI allows Multitrack MusicVAE to produce an astonishing amount of variety, but at the same time model's most fundamental restriction is imposed by General MIDI format itself: the limitation to 128 instrument presets plus the drums. Real music contains instrument sounds selected from an essentially infinite set, and individual pieces of music often contain custom sounds not used anywhere else (Magenta, 2022).

The model presented in this thesis faces similar limitations to that of Multitrack MusicVAE. Although the created model does not contain the same variety of instruments present in the Multitrack MusicVAE's training dataset, if trained on such data the model would also be limited to 128 instrument presets, since its implementation was built on an idea of working with the MIDI formatted files. The author's model also does not possess the same number of customizable options.

Multitrack MusicVAE examples available at: https://on.soundcloud.com/Z52hM

7.2.2 Jukebox

Jukebox, unlike the implemented model and MusicVAE works with audio in its raw format, and it allows it to avoid the same MIDI-related limitations. It combines a complex solution to the problems of both speech synthesis and music generation, allowing it to come a little closer to a truly human-like music generation. Its generative model can synthesize the lyrics by using text-to-speech and generate

melody around it at the same time, while able to maintain the musical consistency throughout the entire song.

In a similar manner to the Magenta's team, OpenAI has provided a variety of options for manipulating the generation process, such as the ability to condition the model on artist, genre, and lyrics.

While the generated songs show local musical coherence, follow traditional chord patterns, and can even feature impressive solos, it is rare to hear familiar larger musical structures such as choruses that repeat. The downsampling and upsampling process introduce discernable noise. The models that are used by Jukebox are so complex, that they become slow to sample from, because of the autoregressive nature of sampling. It takes approximately 9 hours to fully render one minute of audio through its models (OpenAI, 2022).

Comparatively, the model provided in this thesis takes approximately five to fifteen seconds to generate a three-minute song. But it has to be said that Jukebox is a one-of-a-kind project, capable of dealing with an impossibly complex array of problems at the same time.

Jukebox examples available at: https://jukebox.openai.com/?song=789449191

8 Conclusion

This thesis sought to investigate the feasibility of training an artificial neural network to compose music. This goal was accomplished by first studying the structure, and individual components of artificial neural networks, as well as training techniques. Later, a study of state-of-art generative models was conducted, complete with examples.

As part of a practical study, a polyphonic music generator, capable of working with music represented in MIDI format was proposed. The proposed model combined variational autoencoder and multilayer perceptron architectures. The set design would allow the model to develop complex instrumental interdependencies and learn musical patterns, subsequently making the generated music sound more novel and harmonical.

Based on the proposed scheme, an artificial neural network model was built and trained. The process of training required the preprocessing of training data, which consisted of choosing the proper dataset, filtering out the desired subset of songs, and converting it into the format appropriate for training. Apart from the training subset, dataloader classes were required, they were used to feed the training data to the networks and facilitate the training process. The design of the architecture and subsequent definition of artificial neural networks was the next step. The final step was training itself, it was run in multiple stages, first training the VAE networks on separate instruments, and later unifying them through the training of conditional neural networks as well as a lead instrument neural network.

Finally, the results of the created generator model were presented and discussed. The implemented model was compared to existing cutting-edge projects like Multitrack MusicVAE and Jukebox. While being inferior to the named projects, the presented model was able to perform above the author's expectations, it was capable of generating unique and interesting-sounding melodies, that can later be used as an inspiration or concept generator for the process of music production.

In future work, the results of this thesis and the presented artificial neural network model can be used to further explore the field of music generation, as well as a basis for a practical solution, that would combine both MIDI-represented melody and raw formatted voiced lyrics, effectively creating a fully voiced song, while still retaining the benefits of MIDI format.

9 References

[1] GÉRON, Aurélien. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. Boston: O'Reilly, 2017. ISBN 978-1-4920-41948.

[2] FOSTER, David. Generative deep learning: *Teaching machines to paint, write, compose, and play*. Sebastopol: O'Reilly, 2019. ISBN 978-1-0981-3418-1.

[3] BRIOT, Jean-Pierre, Gaëtan HADJERES and François-David PACHET. *Deep learning techniques for music generation*. Cham: Springer, 2020. ISBN 978-3-319-70162-2.

[4] DUBREUIL, Alexandre. *Hands-On Music Generation with Magenta: Explore the role of deep learning in music generation and assisted music composition*. Birmingham: Packt, 2020. ISBN 978-1-8388-2441-9.

[5] MCCULLOCH, Warren and Walter PITS. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics 5*. 1943, (5), 115–133. Available at: doi:https://doi.org/10.1007/BF02478259

[7] THAKUR, Amey and Archit KONDE. Fundamentals of neural networks. *International Journal for Research in Applied Science and Engineering Technology*. 2021, 407-426.

[8] MAKLIN, Cory. LSTM recurrent neural network Keras example. *Towards Data Science* [online]. Towards Data Science, 2019, 14 June 2019 [cit. 2022-10-09]. Available at: https://towardsdatascience.com/machine-learning-recurrent-neural-networks-and-long-short-term-memory-lstm-python-keras-example-86001ceaaebc

[9] ELSARAITI, Meftah and Adel MERABET. A Comparative Analysis of the ARIMA and LSTM Predictive Models and Their Effectiveness for Predicting Wind Speed. *Energies* [online]. 2021, **14**(20), 6782. ISSN 1996-1073. Available at: doi:10.3390/en14206782[10] Dorian, J. (2021, July 20). *Character-level deep language model with GRU/LSTM units using tensorflow*. Nabla Squared. Retrieved November 9, 2022, Available at: https://www.nablasquared.com/character-level-deep-language-model-with-gru-lstm-units-using-tensorflow/

[11] LÓPEZ, Fernando. From a LSTM cell to a multilayer LSTM network with pytorch. *Towards Data Science* [online]. Towards Data Science, 2020, July 27 2020 [cit. 2022-10-09]. Available at: https://towardsdatascience.com/from-a-lstm-cell-to-a-multilayer-lstm-network-with-pytorch-2899eb5696f3

[12] SHI, Yan. Understanding LSTM and its diagrams. *ML Review* [online]. ML Review, 2016, March 17 2016 [cit. 2022-10-09]. Available at: https://blog.mlreview.com/understanding-lstm-and-its-diagrams-37e2f46f1714

[13] NICHOLSON, Chris. A beginner's guide to generative adversarial networks (gans). *Pathmind* [online]. Pathmind, 2020, April 15 2020 [cit. 2022-10-09]. Available at: https://wiki.pathmind.com/generative-adversarial-network-gan

[14] TABIAN, Iuliana, Hailing FU and Zahra Sharif KHODAEI. A Convolutional Neural Network for Impact Detection and Characterization of Complex Composite Structures. *Sensors* [online]. 2019, 2019 (22), 4933. ISSN 1424-8220. Available at: https://doi.org/10.48550/arXiv.1703.10847

[15] DERTAT, Arden. Applied deep learning - part 3: Autoencoders. *Towards Data Science* [online]. Towards Data Science, 2017, 3 October 2017 [cit. 2022-10-09]. Available at: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

[16] Anon. Role of KL-divergence in variational autoencoders. *GeeksforGeeks* [online]. GeeksforGeeks, 2022, 27 January 2022 [cit. 2022-10-09]. Available at: https://www.geeksforgeeks.org/role-of-kl-divergence-in-variational-autoencoders/

[17] RINTERESTED. Multivariate Gaussian distribution. Multivariate gaussian. *NOTES ON STATISTICS, PROBABILITY and MATHEMATICS* [online]. NOTES ON STATISTICS, PROBABILITY and MATHEMATICS, 2022, 20 January 2022 [cit. 2022-10-24]. Available at: https://rinterested.github.io/statistics/multivariate_gaussian.html

[18] ZITAO, Shen. 3 mins of machine learning: Multivariate Gaussian Classifer. *Zitao's Web* [online]. Zitao's Web, 2022, 14 March 2020 [cit. 2022-10-24]. Available at: https://zitaoshen.rbind.io/project/machine_learning/3-mins-of-machine-learning-multivariate-gaussian-classifer/

[20] DEVOXX. Music Generation with Magenta: Using Machine Learning in by Arts Alexandre Dubreuil. *YouTube* [video]. YouTube, 2022, 7 November 2019 [cit. 2022-10-28]. Available at: https://www.youtube.com/watch?v=O4uBa0KMeNY

[21] *Magenta* [online]. Google, 2022 [cit. 2022-11-01]. Available at: https://magenta.tensorflow.org/

[21] LOU. *Music Generation Using Neural Networks* [online]. 2016 [cit. 2022-11-16]. Available at: https://www.semanticscholar.org/paper/Music-Generation-Using-Neural-Networks-Lou/af3c69967cc6756c3d74ffae5d3ad39fe0637755. Academic research. Stanford.

[22] *OpenAI* [online]. Google, 2022 [cit. 2022-11-01]. Available at: https://openai.com/

[23] YANG, Li-Chia, Szu-Yu CHOU and Yi-Hsuan YANG. MidiNet: A Convolutional Generative Adversarial Network for Symbolic-domain Music Generation. *Preprint arXiv:1703.10847* [online]. ArXiv, 2017, 31 March 2017, 2017 [cit. 2022-11-04]. Available at: https://doi.org/10.48550/arXiv.1703.10847

[24] THAM, Issac. Generating music using Deep Learning. *Towards Data Science* [online]. Towards Data Science, 2021, 9 November 2021 [cit. 2022-11-04]. Available at: https://towardsdatascience.com/generating-music-using-deep-learning-cb5843a9d55e

[25] MÜLLER, Meinard. *Fundamentals of Music Processing: Audio, Analysis, Algorithms, Applications* [online]. Springer, 2015 [cit. 2022-11-04]. ISBN 978-3-3192-1944-8. Available at: https://link.springer.com/book/10.1007/978-3-319-21945-5

[26] *DeepMind* [online]. Google, 2022 [cit. 2022-11-04]. Available at: https://www.deepmind.com/

[27] *Gabor Melli's Research Knowledge Base* [online]. Google, 2022 [cit. 2022-11-04]. Available at: https://www.gabormelli.com/RKB

[28] OORD, Aaron van den, Sander DIELEMAN, Heiga ZEN, Karen SIMONYAN, Oriol VINYALS, Nal KALCHBRENNER, Andrew SENIOR and Koray KAVUKCUOGLU. WaveNet: A Generative Model for Raw Audio. *Preprint arXiv:1609.03499*. [online]. ArXiv, 2016, 12 September 2016, 2016 [cit. 2022-11-04]. Available at: https://doi.org/10.48550/arXiv.1609.03499

[30] HARRIS, Charles, Jarrod MILLMAN, Stéfan VAN DER WALT, Ralf GOMMERS, Pauli VIRTANEN et al. Array programming with NumPy. *Nature* [online]. 2020, 16 September 2020, 2020(585), 357–362 [cit. 2022-11-04]. Available at: https://doi.org/10.1038/s41586-020-2649-2

[31] CHOLLET, François. Keras. *GitHub* [online]. GitHub, 2015, 4 November 2015 [cit. 2022-11-04]. Available at: https://github.com/keras-team/keras

[32] ABADI, Martín, Ashish AGARWAL, Paul BARHAM, Eugene BREVDO Zhifeng CHEN, Craig CITRO, Greg S. CORRADO et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *ArXiv* [online]. 2016, 14 Mar 2016, 2016 [cit. 2022-11-04]. Available at: https://doi.org/10.48550/arXiv.1603.04467

[33] PASZKE, Adam, Sam GROSS, Francisco MASSA, Adam LERER and Gregory CHANAN. An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems 32* [online]. Curran Associates, 2019, 2019 (32), 8024–8035 [cit. 2022-11-04]. Available at: https://doi.org/10.48550/arXiv.1603.04467

[34] DONG, Hao-Wen, Wen-Yi HSIAO and Yi-Hsuan YANG. *Pypianoroll: Open Source Python Package for Handling Multitrack Pianorolls* [online]. Taipei, 2018 [cit. 2022-11-04]. Available at: https://salu133445.github.io/pypianoroll/pdf/pypianoroll-ismir2018-lbd-paper.pdf. Academic Research. Academia Sinica.

[35] TBERTINMAHIEUX. MSongsDB/hdf5_getters.py. *GitHub* [online]. GitHub, 2010, 12 December 2010 [cit. 2022-11-04]. Available at: https://github.com/tbertinmahieux/MSongsDB/blob/master/PythonSrc/hdf5_getters.py

[36] DONG, Hao-Wen, Wen-Yi HSIAO, Li-Chia YANG and Yi-Hsuan YANG. MuseGAN: Multi-track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment. *Proceedings of the AAAI Conference on Artificial Intelligence* [online]. 2018, 2018, 2018(32) [cit. 2022-11-04]. Available at: https://doi.org/10.1609/aaai.v32i1.11312

[37] COLIN, Raffel. *Learning-Based Methods for Comparing Sequences, with Applications to Audio-to-MIDI Alignment and Matching* [online]. Taipei, 2016 [cit. 2022-11-16]. Available at: https://academiccommons.columbia.edu/doi/10.7916/D8N58MHV. PhD Thesis. Columbia University.

[38] BERTIN-MAHIEUX, Thierry, Daniel P.W. ELLIS, Brian WHITMAN and Paul LAMERE. The Million Song Dataset. *Proceedings of the 12th International Society for Music Information Retrieval Conference* [online]. 2011, 2011, [cit. 2022-11-04]. Available at: https://www.researchgate.net/publication/220723656_The_Million_Song_Dataset

10 Attachments

Project folder structure:

vae-music-generator

```
├── data
│   ├── generated-midi ..... generated MIDI songs
│   ├── lakh-dataset ..... dataset used for the project
│   ├── processed-pianoroll ..... preprocessed training data
│   └── saved-nn-models ..... saved trained NNs
└── implementation
    ├── data-preprocess ..... data preprocessing methods
    ├── networks ..... declared NN classes
    ├── training ..... training methods
    └── music_generation.py ..... music generation methods
```

Zadání bakalářské práce

Autor:	**Aleksey Yanushko**
Studium:	I1900276
Studijní program:	B1802 Aplikovaná informatika
Studijní obor:	Aplikovaná informatika
Název bakalářské práce:	**Generování hudby pomocí umělých neuronových sítí**
Název bakalářské práce AJ:	Music generation using artificial neural networks

Cíl, metody, literatura, předpoklady:

Thesis goal: Explore and test artificial neural network approaches and techniques in music generation

1. Introduction
2. Exploring current possibilities for generating music and melodies using artificial neural networks
3. Application proposition for music generation
4. Implementation and testing of the proposed application
5. Conclusion and evaluation of achieved results

Foster, 2019 - Generative Deep Learning: Teaching Machines to Paint, Write, Compose, and Play. O'Reilly Media. ISBN:978-1492041948
DuBreuil, 2020 - Hands-On Music Generation with Magenta: Explore the role of deep learning in music generation and assisted music composition. Packt Publishing Ltd. ISBN:9781838824419
Briot, 2020 - Deep Learning Techniques for Music Generation (Computational Synthesis and Creative Systems). Springer. ISBN:9783319701622

Zadávající pracoviště:	Katedra informatiky a kvantitativních metod, Fakulta informatiky a managementu
Vedoucí práce:	Ing. Milan Košťák
Datum zadání závěrečné práce:	15.10.2021