

# Generating Music with Variational Autoencoders

Manuel Antoinette, 4a  
MNG Rämibühl - Maturitätsarbeit  
Supervised by Lukas Fässler

January 6, 2020

## **Abstract**

Generative modelling deals with analysis of complex data distributions of content of various kind. This includes images, texts and sounds. By analysing a dataset of such content with an appropriate artificial neural network model we can learn how the data is built up and synthesise new data that is indistinguishable from the original dataset. In recent years great resources have gone into research and development of such models, especially for image generation. My goal was to develop a model that is capable of generating new original music. I have documented the development and training of a stacked variational autoencoder that can successfully generate new 16-measure songs. The songs generated showed clear musical elements such as rhythm, melody and song structure.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Theory</b>	<b>3</b>
2.1	Autoencoders . . . . .	3
2.2	Variational Autoencoder . . . . .	6
2.3	Stacked Autencoders . . . . .	8
<b>3</b>	<b>Methods</b>	<b>9</b>
3.1	Goal and Problem definition . . . . .	9
3.2	The Data . . . . .	9
3.2.1	Obtaining and analysing MIDI files . . . . .	9
3.2.2	Processing the MIDI files . . . . .	10
3.3	The Model . . . . .	12
3.3.1	Choosing a Model . . . . .	12
3.3.2	Designing a VAE . . . . .	12
3.4	Coding Environment and Hardware . . . . .	13
<b>4</b>	<b>Results</b>	<b>15</b>
4.1	The Variational Autoencoder . . . . .	15
4.1.1	Implementation in Python . . . . .	17
4.1.2	The Training Process . . . . .	19
4.2	The Generated Music . . . . .	20
4.2.1	Melody and Rhythm . . . . .	21
4.2.2	Song Structure . . . . .	21
<b>5</b>	<b>Discussion</b>	<b>22</b>
5.1	The Generated Music . . . . .	22
5.2	The Latent Space . . . . .	22
5.3	Hyperparameters . . . . .	23
5.4	Outlook . . . . .	23
<b>6</b>	<b>Appendix</b>	<b>24</b>
6.1	USB-Stick . . . . .	24
6.2	Eigenständigkeitserklärung . . . . .	24

# 1 Introduction

In the last few years generative neural networks have gained a lot of attention due to amazing improvements in the field. Tremendous amount of work has been done designing sophisticated network architectures and training techniques. These networks have shown to produce highly realistic content, such as images, texts and sounds. Some of these architectures include Generative Adversarial Networks (GANs) and Variational Autoencoders (VAEs). The capabilities of these architectures to generate new meaningful images similar to a dataset were demonstrated in various papers [4, 5] (Figure 1). In a 2015 paper by Alec Radford, et al. [12] even the ability to perform vector arithmetic (in the latent space) was demonstrated (Figure 2). These generative models are extremely powerful tools for data with a complex unknown underlying structure.

One such form of data is music. While music theory allows us to study existing musical pieces in terms of their melodic structure and harmony, it would not be possible to develop an algorithm that produces new songs solely based on music theory. Generative models can analyse thousands of existing musical pieces and learn about their underlying structure and data distribution. Based on this knowledge the models can generate new original music. In this Matura paper I have documented and detailed the development of a generative model designed to generate new original songs and analysed its results.

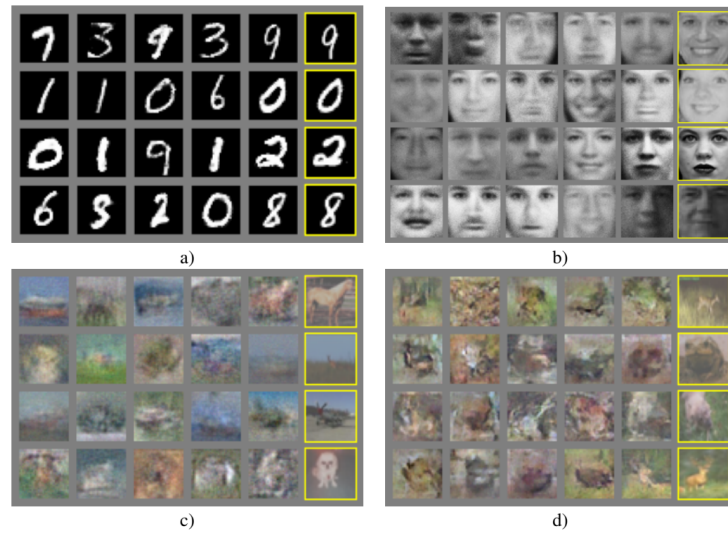


Figure 1: Examples of GANs used to generate new plausible examples for image datasets. The rightmost column shows the nearest training example. [4]

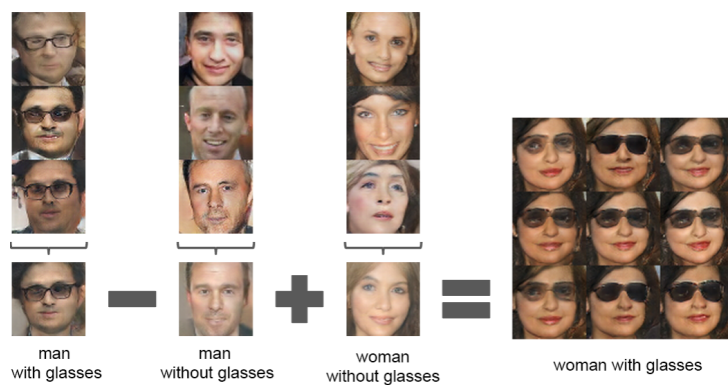


Figure 2: Example of vector arithmetic for GAN-generated faces. [12]

## 2 Theory

Generative modelling is a broad area of machine learning which deals with modelling of probability distributions defined over many datapoints. Each datapoint has thousands of dimensions (e.g. an image has thousands of pixels), and the generative model's job is to somehow capture the dependencies between each dimension (e.g. nearby pixels have similar colour). Learning the distribution of a dataset allows us to synthesize new datapoints according to the same distribution. These datapoints will be completely original, but still have the same characteristics as our dataset.

One of the most popular frameworks for generative modelling is the Variational Autoencoder. As the name implies a, VAE is very similar to an autoencoder, but with a few modifications to perform better at data generation.

### 2.1 Autoencoders

An autoencoder is a data compression algorithm. It compresses the data by *dimensionality reduction*. This forces the autoencoder to reduce the number of features that describe some data and depict which of these features are most important to reconstruct the original data as accurately as possible.

#### Structure

Its architecture is comprised of an *encoder* followed by a *decoder*. The encoder maps some n-dimensional data into a lower dimensional *latent space*. The decoder then tries to reconstruct the original data from that latent space. Depending on the initial data distribution and the size of the latent space, this compression can be lossy, meaning that part of the information is lost during the encoding process and cannot be recovered when decoding. The goal is to find the best encoder/decoder pair with minimal reconstruction error.

The encoder and the decoder are neural networks of mirrored architecture often with several hidden layers. For the autoencoder to act as a compression algorithm the number of neurons in each layer must decrease in the encoder and increase in the decoder.

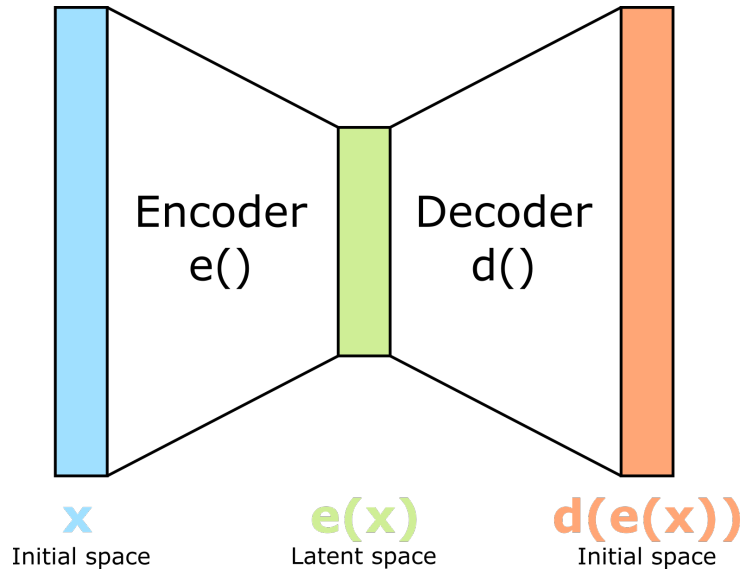


Figure 3: Illustration of the structure of an autoencoder.

### Training

To learn the best encoding-decoding scheme an iterative optimisation process is used. Each iteration we feed the autoencoder with some data, compare the encoded-decoded output with the initial data and backpropagate the error through the network to update the weights and biases. The adjustments necessary are determined by gradient descent.

The function we want to minimize (the loss function) is given by the mean squared error between the input data  $x$  and the encoded-decoded data  $x'$ :

$$\text{loss} = ||x - x' ||^2 = ||x - d(z)||^2 = ||x - d(e(x))||^2$$

where  $x'$  is given by the decoded latent vector  $d(z)$  and  $z$  is given by the encoded input data  $e(x)$ .

### Intuitive Understanding of Dimensionality Reduction

The process of encoding and decoding is a mapping of a vector from the initial space into the latent space and vice-versa. If we consider the encoder and decoder to both only have one layer without non-linearity<sup>1</sup>, they would simply translate to linear transformations that can be expressed as matrices. By doing so we would be looking for the best linear subspace to project

<sup>1</sup>with a linear activation function  $y = x$ .

a given dataset onto with as few information loss as possible. With a 3-dimensional initial space this can easily be visualised:

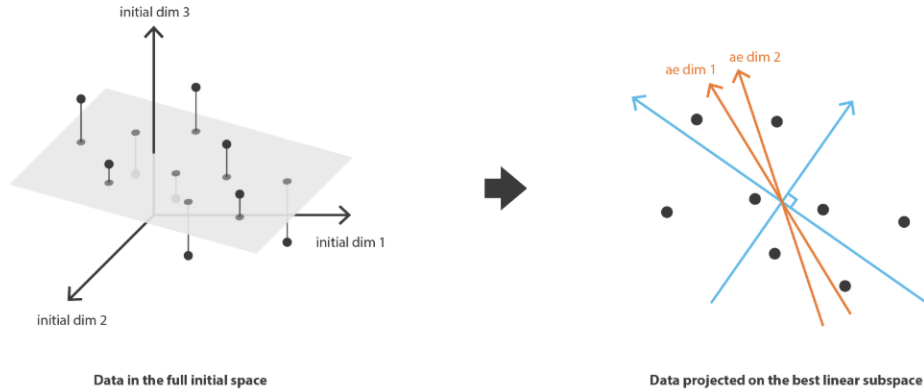


Figure 4: Projection of a 3D dataset onto best linear subspace. The two encoded dimensions (in orange) do not have to be fully independent. Taken and edited from [14].

If we now assume that both the encoder and decoder are more complex deep non-linear neural networks the autoencoder will be able to perform much higher dimensionality reduction and encode more complex data while keeping the reconstruction loss low. If we allow for enough degrees of freedom it would theoretically be possible to reduce any initial dimensionality to a single scalar and reconstruct it perfectly.

However, it is important to keep in mind what happens if we do so. By reducing the latent space dimension size to only one, we would lose most interpretable or exploitable structures in the latent space. Our goal is not to compress the data as far as possible but to reduce it to its most key features. This will later allow us to generate new, meaningful datapoints out of the latent space. Therefore, it is important to carefully control and adjust the number of dimensions in the latent space, depending on the final application of the autoencoder.

## Data Generation

So how do you generate new data after having trained an autoencoder? The idea is to sample some vector from the latent space and feed it into the decoder. As the decoder is trained to decode a latent vector to a datapoint in the initial space we expect it to generate something new and similar to the dataset. However, we would see that this is rarely the case as not all



vectors in the latent space can be decoded into something meaningful. The latent space usually is not regular enough, for us to randomly choose a latent vector.

If we again consider an autoencoder that can encode any initial dimensionality as a scalar and decode it without any reconstruction loss, we would observe severe overfitting. This means that the architecture has learned to encode every training sample in a scalar and then perfectly decode it, basically assigning a number to every training sample. All other numbers are then left without any meaning. In this case we have lost all information about the structure of the data in the latent space.

This irregularity is actually not very surprising, as nothing in the training process enforces the autoencoder to organize the encoded data in a certain way that allows for sampling of new latent vectors. The autoencoder is solely trained to encode and decode as accurately as possible, no matter how the latent space is organised. Thus, we want to adjust the architecture of the standard autoencoder in such a way that allows us to regularize the latent space.

## 2.2 Variational Autoencoder

### Structure

Just as a standard autoencoder, a VAEs architecture is composed of both an encoder and a decoder that is trained to minimize the reconstruction error between the encoded-decoded data and the initial data. However, instead of encoding an input as a single vector we encode it as a distribution over the latent space. We then sample a latent vector from this distribution which then gets decoded as usual. The distribution is usually chosen to be normal so that the encoder can be trained to return a mean and a standard deviation vector that describe the gaussians. The reason why the input is encoded as distribution instead of a single vector is that this allows us to very naturally regularize the latent space. This is done by forcing the distributions returned by the encoder to approach a standard normal distribution during training.

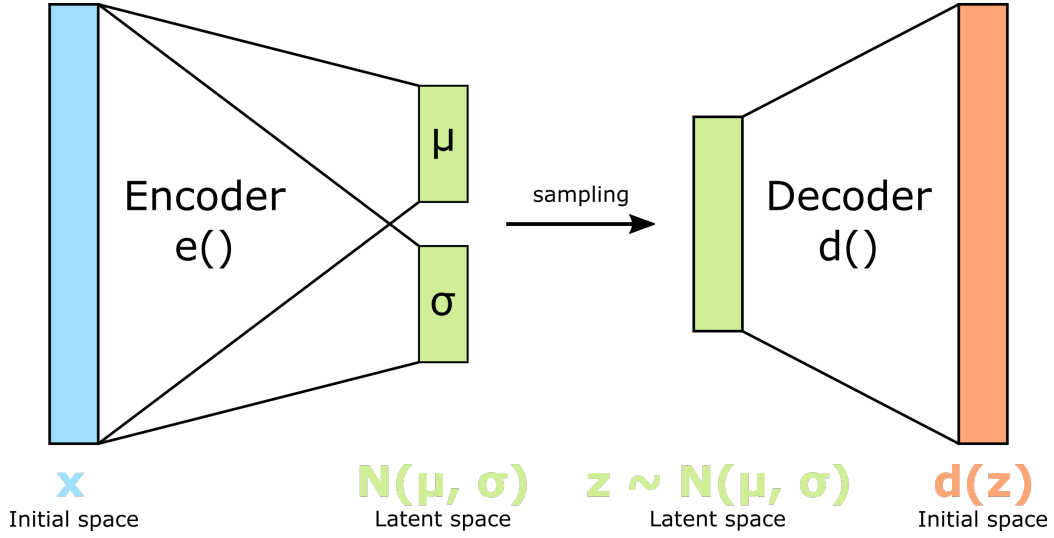


Figure 5: Illustration of the structure of a VAE.

### Training

The loss function that is minimised when training a VAE is composed of a reconstruction term and a regularisation term. The regularisation term is given by the *Kullback-Leiber divergence* between the returned distribution and a standard Gaussian. The Kullback-Leiber divergence is a measure for how one distribution is different from another one. Therefore, the regularisation term we want to minimize is:

$$KL[N(\mu_x, \sigma_x), N(0, 1)]$$

where  $KL[\cdot]$  denotes the Kullback-Leiber divergence,  $N(\text{mean}, \text{standard deviation})$  a normal distribution and  $\mu_x$  and  $\sigma_x$  are the mean and standard deviation vectors outputted by the encoder. The complete loss function of a VAE therefore is<sup>2</sup>:

$$\begin{aligned}
 \text{loss} &= \|x - x'\|^2 + KL[N(\mu_x, \sigma_x), N(0, 1)] \\
 &= \|x - d(e(x))\|^2 + KL[N(\mu_x, \sigma_x), N(0, 1)]
 \end{aligned}$$

<sup>2</sup>Here the sum of the reconstruction term and the regularisation term is used for the loss function. This does not have to be the case, the mean and the difference between the two terms is also often used.

### Data Generation

By encoding the data as distributions and enforcing them to approach a standard normal distribution, the latent space of a VAE will show much more regularity than the one of a standard autoencoder. The space of sensible latent vectors is much more continuous, meaning that two vectors that are close to each other in the latent space reveal similar characteristics when decoded, and complete, meaning that one is very likely to sample a latent vector encoding some meaningful data. The regularisation and the variation in the latent space are key factors for how easily and intuitively we can sample a latent vector which will generate a meaningful piece of data. If the regularisation is too big compared to the variation most of the latent vectors will generate something meaningful, but they will all look very similar (figure 6, right). If the variation is much bigger than the regularisation the opposite is true: The generated data will look very diverse, but much of it will be meaningless (figure 6, left). A well trained VAE will organise its latent space with a good balance between regularisation and variation (figure 6, middle). [14]

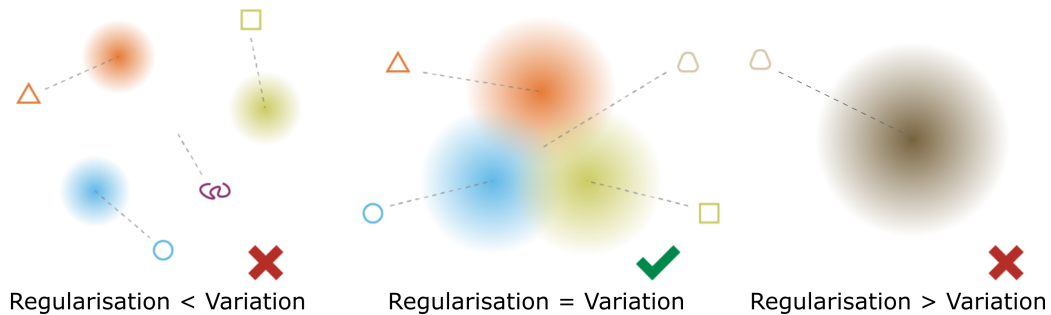


Figure 6: Illustrations of the latent space comparing its regularisation and variation. A good balance between the two is desired. Taken and edited from [14].

### 2.3 Stacked Autencoders

A stacked autoencoder stacks multiple encoders and decoders. By preceding the main autoencoder at the center of the architecture with another encoder, the data gets pre-processed and compressed before entering the main autoencoder. This is called a *feature extraction* and is often applied to problems where large and complex data is involved. This reduces the size and complexity of the data for the main autoencoder. [6]

## 3 Methods

### 3.1 Goal and Problem definition

Inspired by amazing results achieved previously by generative neural networks and my deep interest in music, I set out to build a neural network that can produce new, original songs. I decided to go for whole songs instead of procedurally generated music, as this allows for more structured music, for example with an intro, verse and chorus. As this technology is still subject of active research, I will focus on a proof of concept and its technical details, rather than a fully-fledged product.

### 3.2 The Data

A crucial part of generative modelling is having a dataset large enough and prepared in such a way, that the network can learn about the patterns and features in the data as thoroughly as possible. The type and structure of the data must therefore be chosen very carefully. In the case of music data representation there is a wide range of options. These mostly fall into two categories: Symbolic representations of music and mathematical descriptions of sound waves. Both have their advantages and disadvantages. Raw soundwaves represent the original music very accurately by perfectly retaining information not only about rhythm and melody, but also about non-mathematical elements such as tension, timbre, phrasing and articulation. However, by capturing the music with such accuracy we end up with extremely complex data containing many patterns and features that are not easily visible. Simply put, working with raw audio waves would require an enormously powerful model with complicated architecture. Symbolic representations of music on the other hand, vastly reduce the amount of information kept about a piece of music. Most representations break a song down into notes and store information about them such as pitch, duration and velocity. An example of symbolic musical notation are MIDI files. They organise the notes in a piano roll. This reveals many of the rhythmic and harmonic patterns that are present in a song. MIDI files therefore are a great choice for generative modelling. [2]

#### 3.2.1 Obtaining and analysing MIDI files

First, as much data as possible must be acquired. Fortunately, there are a lot MIDI files available online in various libraries and archives. I was able to find several sources, including MIDI files of classical music, jazz music

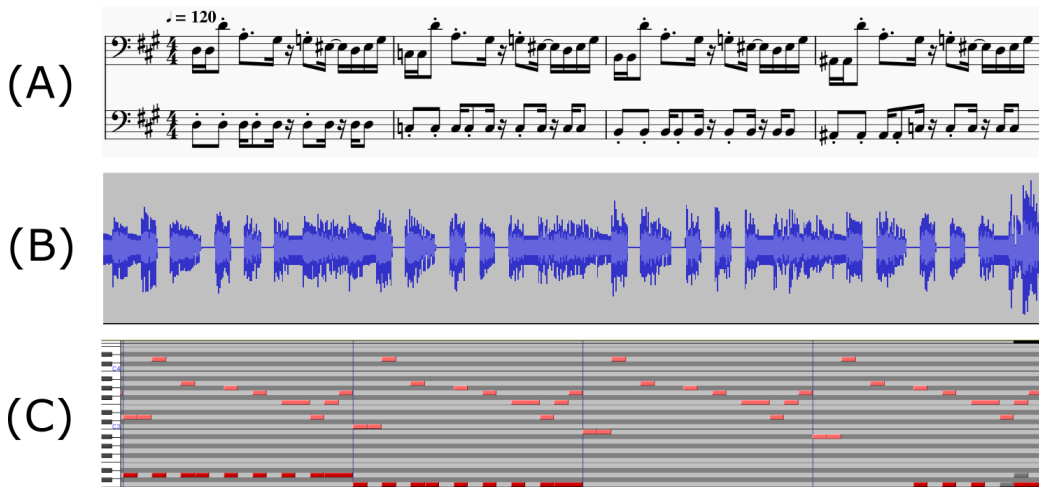


Figure 7: An excerpt of a musical piece from the dataset in different formats: (A) sheet music, (B) raw audio wave, (C) MIDI format displayed in a piano roll.

and video game music. Ultimately I opted to go for video game music, as it is very structured and rhythmical whereas classical and jazz music is much more freeform. In total I was able to obtain over 30'000 MIDI files of video game music. [10]

MIDI files are organised into multiple tracks, which are played by a specified instrument. Each track contains notes with information about their pitch, starting- and ending time, velocity, pitch bend, key pressure and more. Further there are various events such as tempo, key or time signature changes. To make the training data as simple and structured as possible, most of this information will be scrapped when processing the data.

### 3.2.2 Processing the MIDI files

In order to feed the data into a neural network the data must first be processed to fit the architecture. I decided to go for 16 measure songs with each measure being represented by a  $96 \times 96$  piano roll matrix. The measure length of 96 was chosen as it is perfectly divisible by all major time signatures. In the piano rolls I only retained information about pitch and starting time of each note (no duration), as shown in figure 8.

To process all my MIDI files, I wrote a custom Python processing script. I made use of the module `pretty_midi` [13], which allowed me to load and manipulate MIDI files with Python. In the script I went through each midi file iteratively: First, I loaded a MIDI file as a `pretty_midi.PrettyMIDI` Object. This allowed me to change the duration of each note to  $1/96$  of a measure

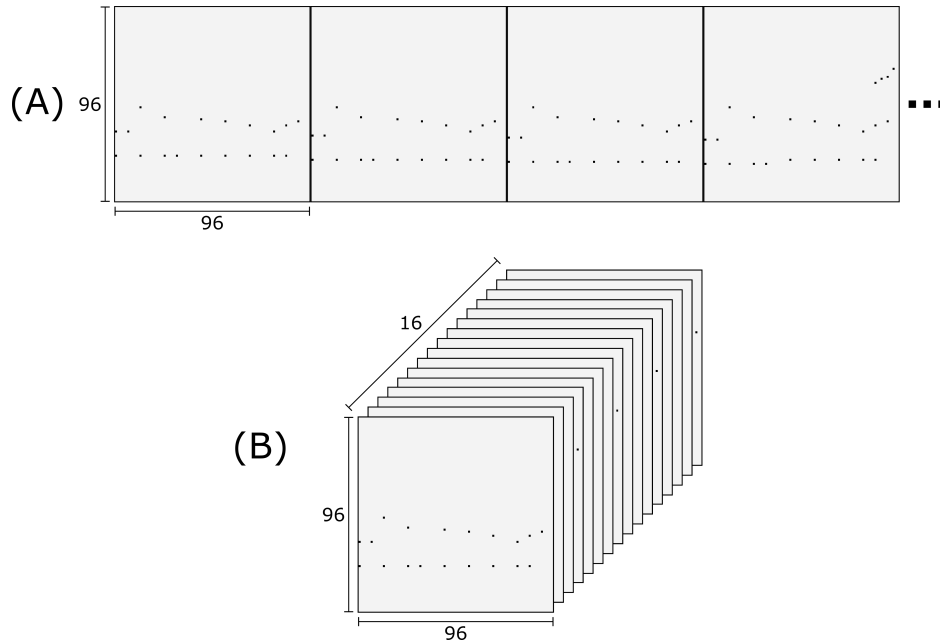


Figure 8: Illustration of the format and shape of the input arrays. (A) shows the piece from figure 7 in pianoroll format with pitch on the vertical axis and time on the horizontal axis. Each measure is  $96 \times 96$  elements large. (B) shows the song vector of size  $16 \times 96 \times 96$ . Note that the input vectors for the neural network have each measure flattened meaning that they are of size  $16 \times 9216$ .

length, which means that each note will be represented by just one single element in the piano roll matrix (as the piano roll matrix is 96 elements long and I do not want to store the duration of notes). Next, I extracted the piano roll as a numpy array using the `pretty_midi.PrettyMIDI.get_piano_roll()` method. As the MIDI files in my dataset were usually far longer than 16 measures, I could get multiple training samples out of a single MIDI file. If the number of measures in a MIDI file was not divisible by 16, I simply discarded the remaining measures. After some reshaping, I had numpy arrays of shape  $16 \times 96 \times 96$  (figure 8B). As I also wanted to discard information about the volume of each note, I normalised the value of each note in the piano roll matrix to 1. To feed them into a neural network I finally had to flatten each measure, giving me training samples of shape  $16 \times 9216$ . As I will be using a data generator when training my model, I saved each sample as a `.npy` file. In total I ended up with over 100'000 training samples.

## 3.3 The Model

### 3.3.1 Choosing a Model

As mentioned in the introductory part, there exist several types of generative neural networks. To achieve the best results possible, it is important to choose a suitable architecture for the task. In the initial phases of my project, I spent a lot of time learning about different architectures and testing them. As music is very time structured data it would make sense to use a Recurrent Neural Network (RNN). RNNs procedurally generate new data based on the recent context, making it possible to generate endless streams of music. But as I wanted to generate whole songs with fixed length and a song structure, RNNs did not fit my needs. This left me with two major categories of generative models: Generative Adversarial Networks and Autoencoders. GANs are usually built with Convolutional Neural Networks (CNNs), meaning that it learns about the data on a very spatial basis. Therefore, GANs are very popular with image generation, as pixels in an image stand in very close relation with their neighbouring pixels. Unfortunately, this is not true for piano rolls. The notes most related to each other often are far apart (e.g. a melody following a baseline is far higher in pitch). Finally, I looked into Autoencoders. They are good at encoding and generating data with complex underlying patterns (not necessarily spatial or temporal data). In contrast to GANs, autoencoders produce way sharper data. For example, images generated by GANs often are very washed out and blurry, whereas ones produced by an autoencoder show very sharp edges and large contrasts. This is beneficial, as the piano roll data is structured in a very binary manner, with either ones or zeros.

For the abovementioned reasons I decided to design an autoencoder. A variational autoencoder is best suited for generative purposes as it gives great control over how the network organises the latent space and because after the model is trained, it is very simple to sample a latent vector that produces meaningful data when decoded.

### 3.3.2 Designing a VAE

After having obtained an intuitive and mathematical understanding of VAEs, I called for inspiration from other projects and examples. In the book *Praxis-einstieg Machine Learning mit Scikit-Learn und TensorFlow* by Aurélien Géron [6] I learned about the concept of a stacked autoencoder. As my music data contains complicated patterns and the song vectors are very large, I deemed it a good Idea to perform a feature extraction before inputting them into the main VAE. Instead of extracting features out of the whole

song I decided to extract a feature vector for each measure. The differences between measures should therefore be much more emphasised for the main VAE, allowing it to learn about the song structure more easily (e.g. chords that change with each measure).

As for the number of layers I went for a standard approach with one hidden layer in both the preceding autoencoder and the main VAE. The number of neurons in each layer is chosen rather arbitrarily. I gained some insights in other projects and adopted similar layer sizes [7]. After some testing I finally decided on a good balance as will be shown in section 4.1.

### 3.4 Coding Environment and Hardware

All code for this project was written in Python version 3.7. I built my VAE with the Keras sequential API implemented in Tensorflow version 2.0 [3]. Further, various modules were used to process and visualise the data (`pretty_midi` [13], `numpy` [11], `matplotlib` [8]). My VAE was developed and trained on a Dell XPS 15 9570 equipped with an Intel core i7-8750H, 16GB of RAM and a NVIDIA GeForce GTX 1050Ti with Max-Q design. Tensorflow was set up to train on the dedicated GPU. Unfortunately, this hardware was not powerful enough to sustain extended periods of training. After the model had trained on a certain amount of training samples, a **Resource Exhaust Error** occurred, meaning that I had run out of memory.

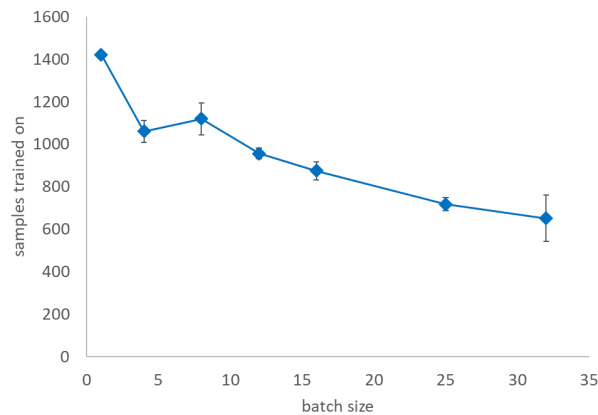


Figure 9: The Number of samples the model was able to train on before the **Resource Exhaust Error** occurred in dependance on the batch size. Smaller batch sizes allowed for better training.

The number of samples the model was able to train on before the error occurred was largely dependent on the batch size chosen. To maximise the



number of trained samples I analysed how the batch size affected training. As can be seen in figure 9, with a smaller batch size the model was able to train on more samples. Thus, I set the batch size to a minimum of one. Training with small batch sizes uses less memory but is computationally more intensive. Therefore, the training process takes a bit more time but is anyway still short at around 5 to 10 minutes. The advantage of a small batch size is that the model can generalize over the data much better than when using large batches. It has been observed in practice that when using a larger batch there is a significant degradation in the quality of the model, as measured by its ability to generalize. [9]

To train the model as well as possible in such short training time, we can increase the learning rate. An increased learning rate means that we make bigger adjustments to the model with each iteration in the training process. Therefore, we are more likely to end up in a state of the model where the loss value is low, which is what we want. Usually a learning rate around 0.001 is chosen, but with my set up I found ten times that, around 0.01, to be a good balance.

## 4 Results

In order to achieve my goal of generating songs I designed my own VAE and optimized it to generate good results with the resources I had. Even though I was constrained by the hardware available to me, the VAE was able to generate songs with clear musical elements, which I will briefly analyse in this section.

### 4.1 The Variational Autoencoder

My VAE is divided into two parts: An outer standard autoencoder (green in figure 10) and an inner main VAE (red in figure 10). The preceding encoder extracts the features of each measure and encodes them in a feature vector. It takes a  $16 \times 9216$  song vector as an input, processes each measure through a hidden layer with 2000 neurons and outputs 16 feature vectors of size 200. Note that every measure gets processed by the same model, meaning that the same operations are applied to each measure. The 16 feature vectors then get concatenated into one large song feature vector of size 3200. The main VAE takes this song feature vector as an input, feeds it through a hidden layer with 1600 neurons and finally outputs a mean and a standard deviation vector, both with 120 elements. These matrices describe a normal distribution in the 120-dimensional latent space. We then sample from this distribution to get a latent vector. This latent vector gets decoded by the main VAE to get the 3200-dimensional song feature vector. To feed it into the feature decoder it gets separated into 16 feature vectors. After the final decoding process, we again have a song vector of size  $16 \times 9216$  in the initial space.

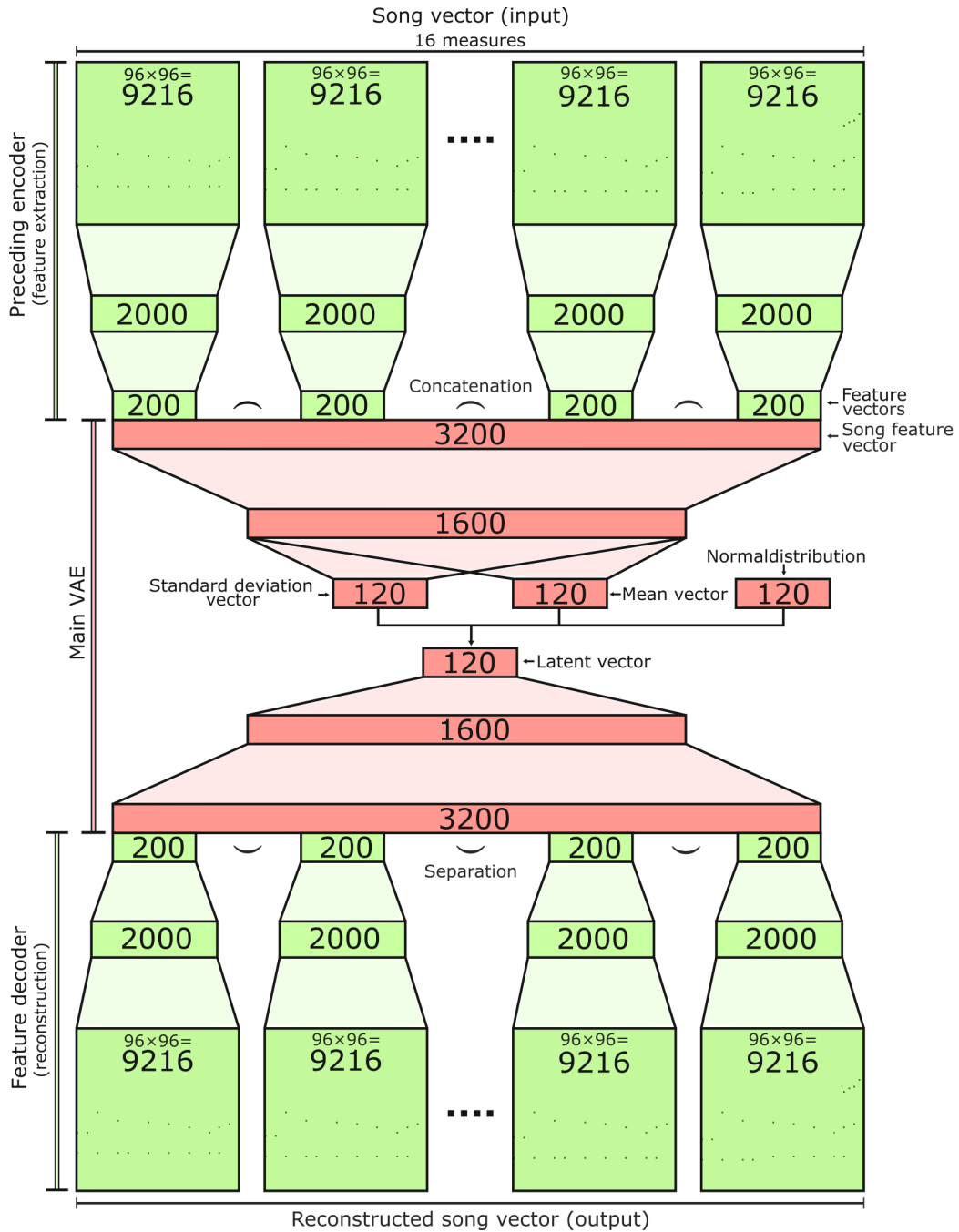
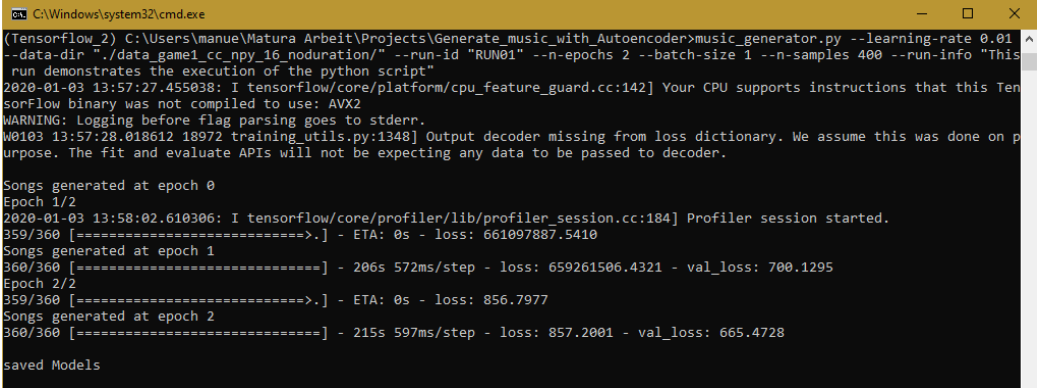


Figure 10: Illustration of the structure of the VAE I designed. Note that both the input- and output layers are illustrated two dimensionally in order to show the song vectors, but are in fact flattened to size  $16 \times 9216$ .

### 4.1.1 Implementation in Python

The whole code used to build and train the VAE is contained within a single Python file. I used an argumentparser that passed all the relevant parameters to the script for very easy usage within the Windows console, as shown in figure 11. This allowed me to automatically train multiple networks with various parameters using a Powershell script.



```

C:\Windows\system32\cmd.exe
(tensorflow_2) C:\Users\manue\Matrura Arbeit\Projects\Generate_music_with_Autoencoder>music_generator.py --learning-rate 0.01 --data-dir ".\data_game1_cc_npy_16_noduration/" --run-id "RUN01" --n-epochs 2 --batch-size 1 --n-samples 400 --run-info "This run demonstrates the execution of the python script"
2020-01-03 13:57:27.455038: I tensorflow/core/platform/cpu_feature_guard.cc:142] Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2
WARNING: Logging before flag parsing goes to stderr.
2020-01-03 13:57:28.018612 18972 training_utils.py:1348] Output decoder missing from loss dictionary. We assume this was done on purpose. The fit and evaluate APIs will not be expecting any data to be passed to decoder.

Songs generated at epoch 0
Epoch 1/2
2020-01-03 13:58:02.610306: I tensorflow/core/profiler/lib/profiler_session.cc:184] Profiler session started.
359/360 [=====>.] - ETA: 0s - loss: 661097887.5410
Songs generated at epoch 1
360/360 [=====] - 206s 572ms/step - loss: 659261506.4321 - val_loss: 700.1295
Epoch 2/2
359/360 [=====>.] - ETA: 0s - loss: 856.7977
Songs generated at epoch 2
360/360 [=====] - 215s 597ms/step - loss: 857.2001 - val_loss: 665.4728

saved Models

```

Figure 11: Execution of the Python script that builds and trains the VAE.

The python code starts off with defining a few helper functions necessary for the sampling of a latent vector, separating the song feature vector, and dealing with the conversion from numpy arrays to MIDI files. Next, the model gets built using the Keras Sequential API implemented in Tensorflow. In order to perform the feature extraction and reconstruction on each measure of the input song separately, I used time distributed dense layers for the preceding encoder and the feature decoder. For the main VAE simple dense layers were used. The whole VAE is divided into two `tf.keras.Model` objects: One for the whole encoding and one for the decoding process. This was necessary as we only need the decoding part for generating new songs after the model is trained. To get the complete model, the two objects are simply connected to instantiate the whole VAE. After defining the loss function the VAE gets compiled with the *Adam* optimizer.

For each training session a filestructure gets generated where all generated songs, the trained model, the Tensorboard graphs and information about the training run gets saved. The files generated by the run in figure 11 can be seen in figure 12. The information about the run is stored in a text file (`20200103135728_RUN01_info.txt` in figure 12) and contains information about all parameters. Additionally I implemented a feature that allowed me to write a small information text containing info about the purpose of the run and its expected results. This text is passed to the script

```

C:\Users\manue\Matura Arbeit\Projects\Generate_music_with_Autoencoder\runs>tree .\20200103135728_RUN01 /f
Aufistung der Ordnerpfade für Volume OS
Volumeseriennummer : 3052-4AF1
C:\USERS\MANUE\MATURA ARBEIT\PROJECTS\GENERATE_MUSIC_WITH_AUTOENCODER\RUNS\20200103135728_RUN01
  20200103135728_RUN01_csv.log
  20200103135728_RUN01_info.txt
  checkpoints
    decoder_pt.h5
    encoder_pt.h5
  generated
    img
      epoch0_0.png
      epoch1_0.png
      epoch2_0.png
    midi
      epoch0_0.mid
      epoch1_0.mid
      epoch2_0.mid
    npy
      epoch0_0.npy
      epoch1_0.npy
      epoch2_0.npy
  graphs
    train
      events.out.tfevents.1578056281.MANMAN-DELL.15232.719.v2
      events.out.tfevents.1578056283.MANMAN-DELL.profile-empty
    plugins
      profile
        2020-01-03_13-58-03
        local.trace
    validation
      events.out.tfevents.1578056482.MANMAN-DELL.15232.129770.v2

```

Figure 12: Files generated by the training run shown in figure 11.

by the flag `--run-info` in figure 11. This information callback was crucial during development and implementation of the model, as it allowed me to go back and thoroughly analyse previous training sessions and debug and improve my code accordingly. To monitor the loss value of the VAE in real time during training I implemented a Tensorboard callback. Tensorboard is a browser based application that allows monitoring and analysis of all kinds of neural networks. Tensorboard is deeply implemented within Tensorflow and Keras, therefore coding this callback was very simple. Now to the actual training of the model. Usually this can be done very easily with the `tf.keras.Model.fit()` method, but this requires the whole dataset to be loaded into RAM. As my dataset is very large, this was not possible. Therefore I had to use the `tf.keras.Model.fit_generator()` method instead. With this method a data generator (a class inherited from `tf.keras.utils.Sequence`) provides the trainer with a batch of samples each training iteration. This way only one single batch of data has to be loaded into RAM at once. Therefore I had to write a custom data generator class that loads the samples from a directory where each song array from the dataset is stored as a `.npy` file. Within this data generator class I was also able to implement that after each epoch a few songs get generated.

This way it is possible to listen to the VAE’s song generating capabilities while it is still training and see its development. After the model is finished with training or the computing resources have exhausted, new songs get generated one last time with the final model. The song vectors generated get saved as `.npy` files in `./generated/npy/`, get converted to and saved as MIDI files in `./generated/midi/`, and also saved as images of a piano roll in `./generated/img/` as shown in figure 12. The pianoroll images again allow for easy analysis of each training session after it is completed.

#### 4.1.2 The Training Process

In the training process we feed the model with a batch of samples, calculate the loss and finally adjust the weights and biases using backpropagation and gradient descent. To monitor the model’s ability to generate new sensible music a few songs are generated after each training epoch. To monitor the loss value during training I set up a Tensorboard callback. The Tensorboard graphs are shown in figure 13.

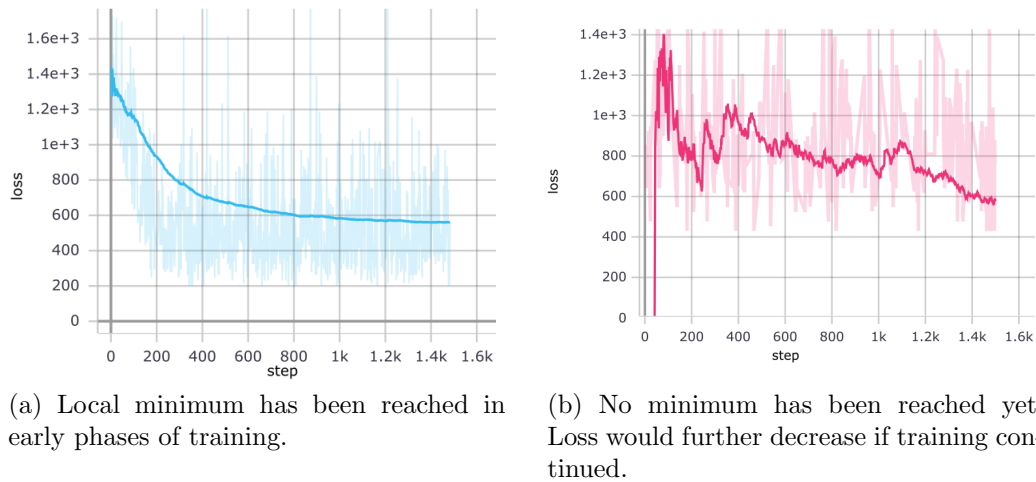
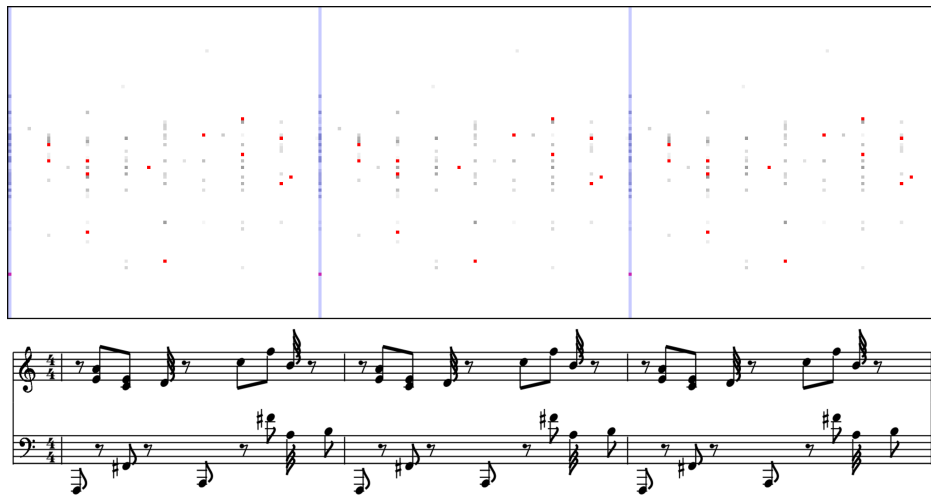


Figure 13: Graphs showing the development of the loss value during training. The darker graph lines are smoothed out to more clearly see the developments.

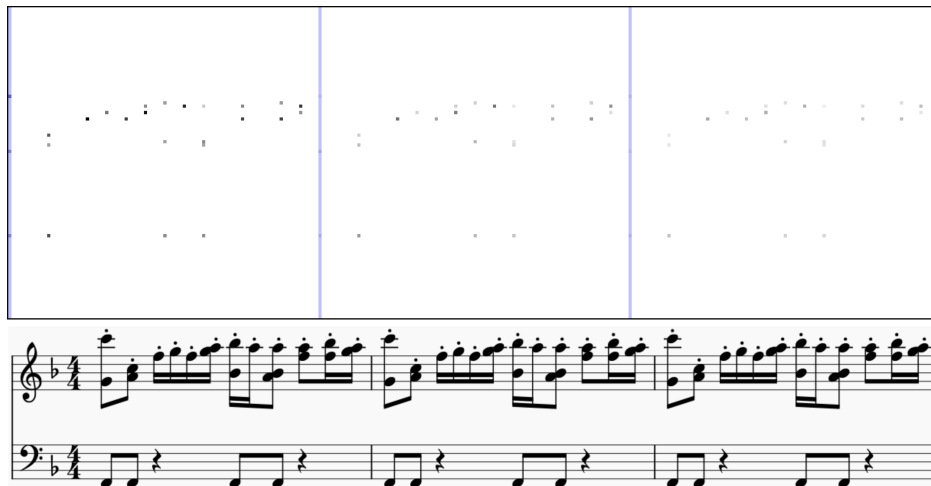
In some cases, the loss value converged quite early during training, as can be seen in figure 13a. This means that we have found a local minimum of the loss function. In other cases though, the loss value constantly decreased, as seen in figure 13b. In these cases, the VAE would have benefitted from a longer training session.

## 4.2 The Generated Music

After training my VAE I could successfully generate new songs. The songs generated showed various musical elements such as rhythm, melody and song structure. In the following sections I will show a few examples that demonstrate these musical elements.



(a) For a clearer understanding of the melody only notes with a value above a certain threshold in the song vector (marked in red) are denoted in the sheet music.



(b)

Figure 14: Two excerpts of generated songs. A visualisation of the generated song vector along with the corresponding sheet music is shown. The blue lines in the visualisation of the song vector mark the first beat of each new measure.

### 4.2.1 Melody and Rhythm

Many training runs ended with the VAE generating songs with the same music in all measures. This seems to be an easy way for the model to reduce the loss value. Nevertheless, the music generated often was very interesting with musical features which can also be observed in real music. Figure 14 shows excerpts of songs generated by the VAE. In some cases it was helpful to filter out the loudest notes (the notes with the highest value in the generated pianoroll matrix). In figure 14a the most prominent melody was filtered out by only considering the values in the pianoroll matrix above a certain threshold (highlighted in red). This was not necessary in figure 14b. We can see that both examples are quite musical as the melody is very rhythmic and the base line always nicely plays on the beat.

### 4.2.2 Song Structure

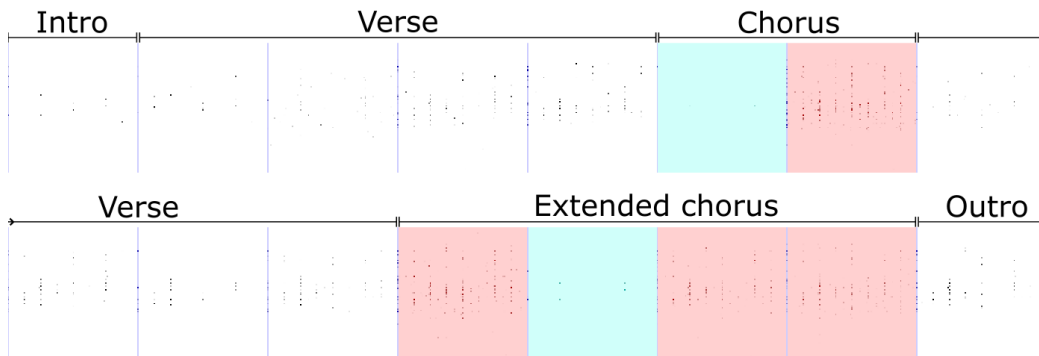


Figure 15: An example of a generated song demonstrating the song structuring capabilities of the model. The blue lines mark the first beat of each new measure. Measures tinted in the same color contain the same musical motifs. The verse structure is interpreted according to the motifs.

Figure 15 shows an example of a generated song that nicely demonstrates how the VAE learned to structure songs. The measures tinted in the same colour contain similar rhythms and melodies. These motifs play towards the middle and the end of the song, with the red tinted measures being the loudest and showing the most movement in the music. This led me to interpret these motifs as a chorus of the song. Their placement within the song allowed for an interpretation of a well-fitting song structure with an intro, two equally long verses, a standard and an extended chorus and finally an outro. Such a structure was applicable to many of the songs generated.



## 5 Discussion

### 5.1 The Generated Music

The main element which differentiated the generated music from random noise is rhythm. Most of the musicality of these songs comes from the leading melodies having very interesting rhythms. As the dataset is processed to treat all notes as single strikes – no holds, the rhythmic elements get emphasized more over melodic or harmonic elements. Therefore, it is sensible that the VAE learned about rhythmic elements the best.

Harmonic structure was rather hard to decipher, as most songs were very repetitive with each measure playing the same melodies. The reason for this is not quite clear. But somehow by doing so the network can easily reduce the loss value (as mentioned in section 4.2.1).

I was usually able to interpret the generated songs as a piano piece with two channels (left and right hand) and could write them down in sheet music with two voices accordingly (as seen in figure 14). The MIDI files in the dataset however, often had more than two voices with a much wider pitch range. The VAE somehow learned to only generate the leading melodies of a song. These leading melodies are also easiest for us humans to memorize. Everything else is primarily there to accompany the main melody. It therefore makes sense that the VAE too only learned to generate the leading melodies.

### 5.2 The Latent Space

The latent space of a VAE allows us to adjust the characteristics of a generated song depending on how we sample from it. How well and intuitively this works highly depends on the regularisation and the variation in the latent space. Unfortunately, in my results the latent space did not show much variation (see figure 6 in section 2.2 on the right). Most songs generated from the latent space with the same VAE sounded very similar with the biggest difference between them being the volume of the notes. The very limited training resources most certainly are the reason for this. Making out the patterns in the music and learning how they can be changed is one of the hardest tasks for the VAE. It would therefore require further training on many more samples. However, it is important to note that each training run produced unique songs, meaning that I could simply train a new VAE to generate a completely new song.

### 5.3 Hyperparameters

There are a lot of hyperparameters to vary in the model I developed. This includes all the layer sizes, the optimizer used, its learning rate etc. All these parameters could be varied and adjusted in such a way that the VAE generates the best results possible. With music however, it is hard to pin down what the best results are, as the quality of music is hard (maybe impossible) to quantify. Therefore there exists quite a large range of settings of hyperparameters where the model produces *good* results. Within this range it is left to the developer to set them to his likings.

### 5.4 Outlook

Even though I could only train the VAE for short amounts of time due to the limited computational power available to me, the VAE I developed was able to generate very impressive results. If the network could be trained on more powerful hardware it is very likely that the generated music will be indistinguishable from real music, as currently is the case with image generation. There already exist slider-based editing tools for generating images of faces [1]. If a model for music generation could be fully trained with its latent space encoding real interpretable features of music, a similar tool could be developed for music generation. This would not only open up a completely new creative process but also have practical applications like actively generating music in video games depending on the players situation in the game. Further, by using such generative models we can for the first time apply technology to produce and utilize real artistic content.

## 6 Appendix

### 6.1 USB-Stick

The included USB-stick contains the following files and folders:

- `music_generator.py`: The Python script for building and training the VAE, described in section 4.1.1
- `data_loader.py`: The Python script used for processing the MIDI files, described in section 3.2.2
- `20200103135728_RUN01/`: The folder shown in figure 12
- `results/`: Folder containing the three examples of the generated songs presented in section 4.2

### 6.2 Eigenständigkeitserklärung

Der/die Unterzeichnete bestätigt mit Unterschrift, dass die Arbeit selbständig verfasst und in schriftliche Form gebracht worden ist, dass sich die Mitwirkung anderer Personen auf Beratung und Korrekturlesen beschränkt hat und dass alle verwendeten Unterlagen und Gewährspersonen aufgeführt sind.

*Datum, Unterschrift:*

---

## References

- [1] Carykh. celebrityfaces. <https://github.com/carykh/celebrityFaces>, 2018. Accessed: 01.01.2020.
- [2] Roger B. Dannenberg. A brief survey of music representation issues, techniques, and systems. *Computer Music Journal*, 1993.
- [3] Martín Abadi et. al. TensorFlow: Large-scale machine learning on heterogeneous systems. <http://tensorflow.org/>, 2015. Accessed: 01.01.2020.
- [4] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems 27*. 2014.
- [5] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, and Daan Wierstra. DRAW: A recurrent neural network for image generation. *CoRR*, 2015.
- [6] Aurélien Géron. *Praxiseinstieg Machine Learning mit Scikit-Learn & TensorFlow*. O'Reilly. Translated by Kristian Rother.
- [7] HackerPoet. Neural composer. <https://github.com/HackerPoet/Composer>. Accessed: 01.01.2020.
- [8] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 2007. <https://matplotlib.org/>. Accessed: 01.01.2020.
- [9] Nitish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tang. On large-batch training for deep learning: Generalization gap and sharp minima. 2016.
- [10] Mike Newman. Video game music archive. <https://www.vgmusic.com/>. Accessed: 01.01.2020.
- [11] Travis Oliphant. NumPy: A guide to NumPy. <https://numpy.org/>, 2006. Accessed: 01.01.2020.
- [12] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks, 2016.

- [13] Colin Raffel and Daniel P. W. Ellis. Intuitive analysis, creation and manipulation of MIDI data with `pretty_midi`. <https://github.com/craffel/pretty-midi/>, 2014. Accessed: 01.01.2020.
- [14] Joseph Rocca. Understanding variational autoencoders (VAEs). <https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73>. Accessed: 01.01.2020.