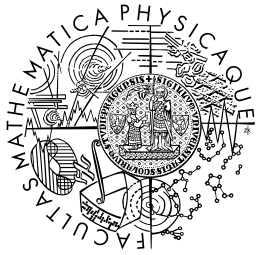


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jonáš Havelka

Generating Music Symbols Using Neural Networks

Institute of Formal and Applied Linguistics

Supervisor of the bachelor thesis: doc. RNDr. Pavel Pecina, Ph.D.

Study programme: Computer Science

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I want to thank my supervisor, doc. RNDr. Pavel Pecina, Ph.D., and Mgr. Jiří Mayer for their great help. I also thank my almost-fiancé Bětka Neubauerová for proofreading, support, and providing food.[†] ~~Finally I should thank Grammarly for corrections of my terrible english.~~ Finally, I thank Grammarly for correcting my terrible English.

[†]The template says I should express my thanks to food providers. And I thank Mirek Kratochvíl (and, transitively, Martin Mareš, Arnošt Komárek, and Michal Kulich) for the template.

Title: Generating Music Symbols Using Neural Networks

Author: Jonáš Havelka

Institute: Institute of Formal and Applied Linguistics

Supervisor: doc. RNDr. Pavel Pecina, Ph.D., Institute of Formal and Applied Linguistics

Abstract: We create more training data for the optical music recognition (OMR) task by generating artificial images of the music symbols.

We follow up Mashcima and the model J. Mayer trained on it. We take the Rebelo dataset (dataset of music symbol images), adjust it with some computer vision methods, and train generative neural networks (above all, variational and adversarial autoencoders) on it.

By replacing some original images in Mashcima input with ones generated by those networks, we get more general performance from the model: For slightly worsening on the original dataset (CVC-MUSCIMA), we get much better results on the PrIMuS dataset. Also, we create very realistic synthetic images of music symbols.

Keywords: optical music recognition; synthetic data generation

Contents

Chapter 1

Introduction

In machine learning, if we want to train a better model, we need plenty of training data. We can achieve this in many ways. Firstly we can take real-world data. Nevertheless, this may be very costly, especially when it requires many people to work for many hours. Another way is data augmentation. However, good data augmentation is often rocket science. Sometimes we find another dataset and can use it directly or modify it for our purpose. We do this in our work a little. Finally, we can create synthetic data. Either we can use our knowledge of the rules that made the data. For example, Mashcima [1, 2] uses rules of music notation to create train data for OMR (optical music recognition) from individual real-world symbols. Alternatively, we notice that nowadays, we can teach computers to create images that look real only by imitating real-world images. Our work uses this to form an infinite supply of synthetic training data.

We focus on optical music recognition (OMR). The OMR task is about inputting images of music scores (let us say, individual staffs) and outputting computer-readable data describing music written in them. As output encoding, we use the Mashcima encoding [1, 2], so we work only with monophonic ("one note/rest at each time") music. There are already models that do OMR. For their training, we need many training data. However, nowadays, there are not many (digitally) annotated music notation images. Directly synthesizing full images of music notation is a problematic way. Thus we can divide it into two steps – generating symbol images and joining them together. Mashcima does the second step. This thesis does the first one. With generative neural networks (above all, variational and adversarial autoencoders), we generate symbol images from learning on the Rebelo dataset [3] adjusted with some computer vision methods.

Besides realistic-looking images, we get training data for the model [1, 2]. With this data, we can train it more generally, reducing the symbol error rate on the PrIMuS dataset [4] from 0.560 (or 0.613 originally shown in the thesis [1]) to 0.498.

Chapter 2

Prerequisites

For our work, we need many already developed knowledge and resources, such as architectures of generative neural networks (Section 2.1), some music theory (Section 2.2), datasets of music symbols (Section 2.3), already built music engraving software Mashcima (Section 2.4), and image operations (Section 2.5).

2.1 Generative neural networks

A *neural network* (NN) is a math function (preferably a differentiable one) composed of many trivial math operations (such as addition, multiplication, and some *activation functions*) with plenty of parameters (weights). Usually, it is arranged into *layers*, and each layer takes the previous layer's output as its input.

To get the NN output, we calculate all operations (this is called *forward propagation*). For "learning", we use some *loss* (or cost) function. It compares the desired output of NN with the actual output and return number, denoting "how good is the actual output". Then we derivate this output of the loss function with respect to NN parameters and adjust parameters for decreasing loss. This procedure is called *backward propagation*.

In this work, we mainly need two layers: a *fully connected layer* (also called dense), which takes inputs as a vector, multiplies it by a matrix (elements of the matrix are the parameters), adds a vector called *bias* (elements of the bias are also parameters), and applies (element-wise) activation function. A *Convolutional layer* does the same but separately for each piece (for example, 3×3 pixels of the image) of input (with the same matrix).

We only put those layers, activation functions, and loss functions together. TensorFlow [5] handles all implementation of this.

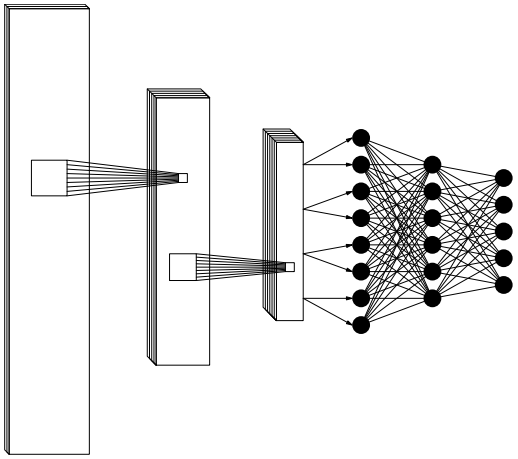
Generative neural networks work in the way that for generating some output, we provide some point from so-called *latent space* (n-dimensional real space), and generative NN produces generated data for this point (deterministically). The latent space also has some probabilistic distribution (called *latent prior*) over it, which helps encode that some data is more common and some less. (It also restricts the used part of latent space, so we do not have to deal with huge numbers.)

2.1.1 Downsampling and upsampling

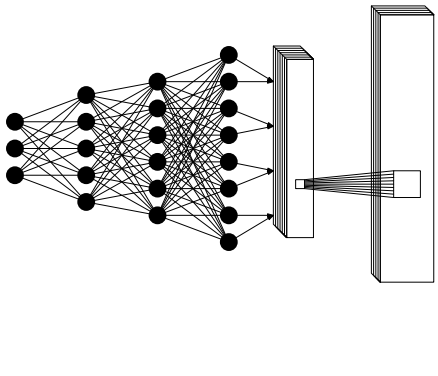
By *downsampling*, we mean *encoding* data from a high-dimensional space (for example, an image) to a low-dimensional space (a latent space). On the contrary, by *upsampling*,

we mean *generating* data from a low-dimensional to a high-dimensional space.

In our work, downsampling is done by applying firstly convolutional layers (with batch normalization) and then fully connected layers, both with a rectified linear unit (relu) activation function. Upsampling is the same "in reverse", which means, firstly, we apply fully connected layers, then transposed[1] convolutional layers (with batch normalization).



(a) Downsampling: After input, there can be some convolutional layers C[1], C[2], ... They are parametrized by the number of channels, kernel size, and strides. Then there is the flattening layer (it converts *width* × *height* × *channels* to one-dimensional *width*·*height*·*channels*). After that, we can have dense layers D[1], D[2], ... parametrized by the number of outputs.



(b) Upsampling: It is more complicated than downsampling. Of course, it can start easily with dense layers D[1], D[2], ... But in the last dense layer (in downsampling the one after flattening), we must count the number of outputs, so after reshaping it has the correct number of channels, width, and height (to reaching desired output shape after convolutions). Thus we use the first number of channels (C[1]) in dense layers. Then we reshape the output of this layer and continue with convolutional layers C[2], C[3], ...

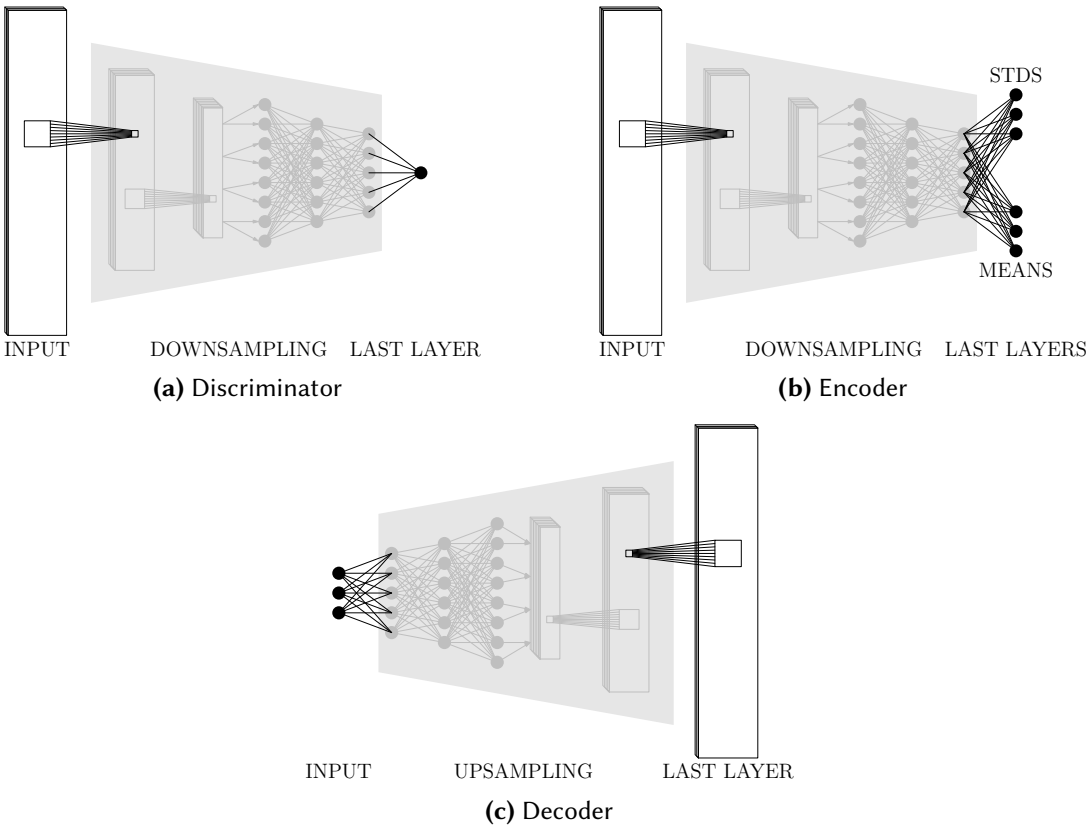
Downsampling and upsampling are our building blocks. If we supply downsampling with the fully connected layer to one number (and sigmoid activation), we get *discriminator* – NN telling "how much, input data meets some criterion" (1.0 means entirely, 0.0 means absolutely not).

Upsampling with a proper last layer (in our case, convolutional or fully connected layer with sigmoid activation because we want to output images with pixels' values from 0.0 to 1.0) forms a *decoder* (often called a generator, especially in GAN). The decoder takes a point from latent space and "decodes" it to data (for example, an image).

Finally, downsampling forms an *encoder* – NN taking data and "encoding" it to a point in the latent space. Even the encoder needs a special last layer because we need to encode data to a probability distribution over the latent space, not to a single point in it. So we generate two numbers (or vectors if the latent space is multidimensional) – one represents a mean of normal distribution, and the other represents a standard deviation. See Subsection 2.1.3.

[1] A convolutional layer has a tensor from kernel to one "pixel", and a transposed convolutional layer has a tensor from one "pixel" to a whole kernel.

We equip a discriminator with binary cross-entropy loss. Decoder has this loss too, but only for small data because it diverges with binary cross-entropy on bigger data. For bigger data, we use mean square error (MSE). The reader can notice that an encoder has no loss as we calculate the loss on latent space in other ways, see Subsections 2.1.3 and 2.1.4.



(a) Discriminator **(b)** Encoder **(c)** Decoder

Figure 2.2 Building blocks for generative networks

2.1.2 Generative adversarial network (GAN)

A *generative adversarial network* (GAN) [6] is the easiest universal way to teach computers to mimic real-world data. It consists of a decoder and a discriminator. The decoder tries to generate realistic data, and the discriminator tries to tell which data is real or fake. In this system, "the decoder acts as a student, and the discriminator acts as a teacher".

In particular, we take real data and the same number of random samples from the latent space. From those samples, we generate (by decoder) fake data. Next, we train the discriminator (to label real data 1.0 and fake data 0.0) in parallel with training the decoder by pushing the system decoder–discriminator to label 1.0. A gradient flow through the discriminator causes the decoder to "know" how to generate more realistic (according to discriminator) data.

If we want to generate category-conditioned data, we add as many new dimensions as we have categories to the latent space. In these new dimensions, we always have 1.0 in the desired category and 0.0 elsewhere. In addition, we must provide a category of data (we need labeled data) to the discriminator (after convolutional layers, before fully connected layers).

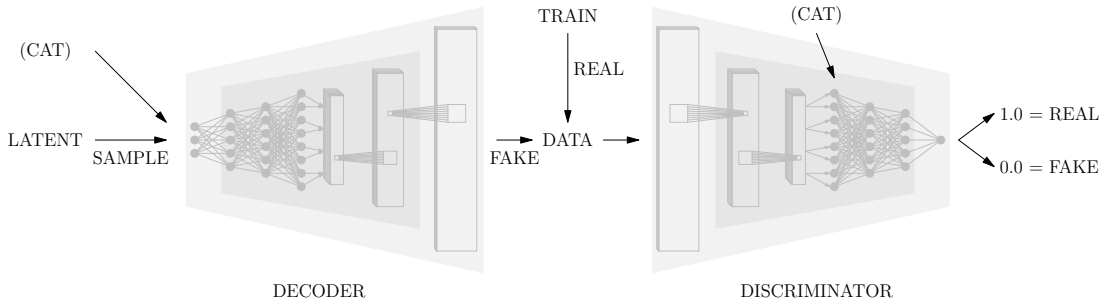


Figure 2.3 GAN

2.1.3 Variational autoencoder (VAE)

A *variational autoencoder* (VAE) [7, 8] is based on the idea that we want to encode data to the latent space and then decode it back. So it consists of an encoder and a decoder. However, if we encode every data to only a single point, we end up with a decoder that can decode only these single points. So we encode data to some distribution over the latent space and want to fill the latent space with those distributions.

We use normal distributions for those distributions because, for training, we need to count gradient flow from samples of those distributions to their parameters. And we know that samples from a normal distribution with a particular mean and a particular standard deviation are samples from the standard normal distribution (mean = 0, standard deviation = 1) multiplied by that standard deviation and shifted by that mean. On this, we already know how to count gradients. (This is called the 'reparametrization trick'.)

Contrary to GAN, we train VAE as one system encoder–decoder. We input real images and train VAE to generate the same ones. In addition, we add Kullback–Leibler (KL) divergence (between encoded distributions and latent prior) to loss. This way, we force the encoder to use the whole latent space.

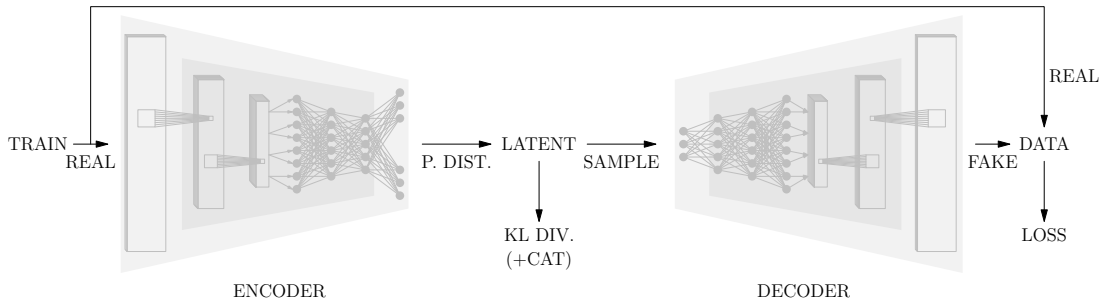


Figure 2.4 VAE

If we want to generate data conditioned on category, we again add categorical distribution to the latent space. In this part of the latent space, we do not use KL divergence. Instead, we use MSE between the encoder's standard deviation and 0.0 and between the encoder's mean and the one-hot[2].

[2]If the category is n, then all values are zero except n-th, which is one.

2.1.4 Adversarial autoencoder (AAE)

An *adversarial autoencoder* (AAE) [9] works the same way as VAE, except it has a discriminator instead of KL divergence. This discriminator takes samples from encoder distributions and samples from latent prior and tries to discriminate them (as it discriminates real and fake data in GAN).

This leads to a state where the encoder must "deceive" the discriminator instead of only fitting latent prior. So the output of AAE is sharper than the output of VAE. Also, we can use more distributions (for example, distributions from which we can only sample but do not have a description) because we only must be able to sample, not count KL divergence. Nevertheless, in the end, we did not try other latent priors in our work.

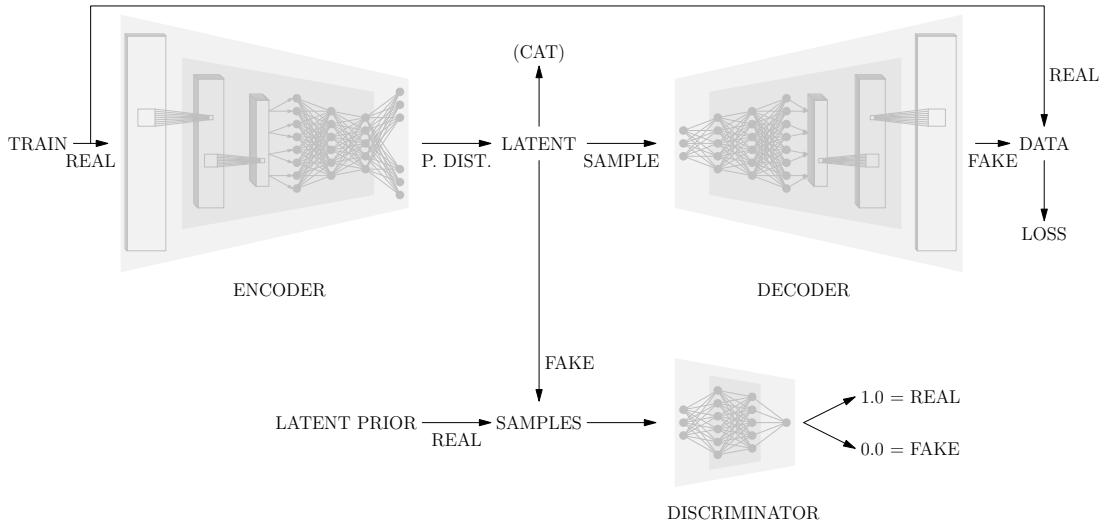


Figure 2.5 AAE

2.2 Music notation

Those generative networks, in general, work for images and even for any data, but in this work, we use them on music symbols. Music notation is too complex, so we do not generate images of full notation. Instead, we generate only particular symbols (and then use Mashcima to synthesize them together; see Section 2.4).

Music notation has many symbols:

- **Notes:** *whole, half, quarter, eighth.* Notes have heads, stems, and flags or beams. The notehead's "center" determines the note's pitch (excluding change of pitch by key, accidentals, and clefs), and by this point, the note is situated in the staff – the first *attachment point.*

  The note can have one stem (or sometimes – in polyphony – two stems), and the stem is situated down or up depending on the pitch and other factors. So we should detect it and rotate notes so all stems aim in the same direction. Also, the other end of the stem is another attachment point because there, the beams bind.

- **Clefs:** *G-clef, F-clef, C-clef.* All clefs indicate where the specific note is on the staff by some attachment point. The G-clef has this point at the start of the spiral, approximately at the image's horizontal center and the image's vertical first third (from the bottom). F-clef has it between two dots or approximately at the rightmost point. C-clef is the most straightforward because it is vertically symmetric, and the horizontal position is not very important, so we can say that it has an attachment point at the center of the image.

- **Accidentals and key signature:** *flat, sharp, natural.* Accidentals change the pitch of notes on the staff line (or the staff space) where the closed part of the accident lies.

- **Rests:** *quarter, eighth, sixteenth, and thirty-second rest.* We can place these symbols "wherever we want". So we can center them in the center of their images.

- **Irrelevant small symbols:** *sixteenth and thirty-second note, half and whole rest, double flat, double sharp, half accidentals, time signatures, duration dot, accent, staccato, tenuto, fermata, trill, breve, longa, breve and longa rest, crescendo, decrescendo, coda, repeats, texts (lento, piano, forte, Ped., and others).* Many symbols are not too crucial for reading music notation, or we can create it another way, or they are rare, or they have one shape, so we do not work with them in this work (although with the proper data, our algorithm can learn to generate them).

- **Lines:** *staff (staff lines), ledger line, slur, beam, barline, triola, ottava alta, and bassa.* Lines are unsuitable for generating them by learning to mimic real images because they are often linked to some context (like the position of the first and the last note under it), and we can generate them more suitable ways.

2.3 Music datasets

There are a lot of types of information that music datasets can have. We can have images of whole staffs annotated with computer-readable descriptions of what symbols are in those images. This we can find in the **PrIMuS** dataset [4] – a dataset containing rendered real-music incipits[3], each annotated with a sequence of symbols. Mashcima (Section 2.4) uses those sequences as a database of real-world music, and one of the evaluations of Mashcima (which we want to beat) is on printed PrIMuS.

We can have scans of hand-written music, such as in **CVC-MUSCIMA** [10]. It has 1000 music sheets written by 50 different musicians. Over it, **MUSCIMA++** [11] is built. MUSCIMA++ contains a very detailed annotation of the part of CVC-MUSCIMA. MUSCIMA++ distinguish parts of a symbol such as a notehead, and for every such part in annotated part of CVC-MUSCIMA, MUSCIMA++ provides a mask, which pixels belong to this part. Moreover, it contains relations between those parts/symbols and denotes at which points these relations occur.

[3] "An incipit is a sequence of notes, typically the first ones, used for identifying a melody or musical work." [4]

This provides us annotated (symbol name and attachment points) set of symbol images. In this way, it serves Mashcima (Section 2.4), and we tested (by guesstimate) the automatic finding of attachment points (Section 3.1) on it a little.

The **Rebelo** dataset [3] provides symbols divided into groups (even hierarchically) as well. Actually, it consists of two datasets. The first is the one grouped hierarchically and consists of symbol images squeezed into a constant-sized square. However, we only need the second dataset with original images of symbols for our experiments. Those images are divided into hand-written and printed, and into the following categories: Accent, *AltoCleff* (C-clef), BarLines, *BassClef* (F-clef), Beams, Breve, Dots, *Flat*, *Naturals*, *Notes* (quarter notes), *NotesFlags* (eighth and smaller notes), *NotesOpen* (half notes), Relations, *Rests1* (quarter rests), Rests2 (eighth and smaller rests), SemiBreve, *Sharps*, TimeSignatureL, TimeSignatureN, *TrebleClef* (G-clef). This dataset we use for training image generation.

Last but not least, there is the **HOMUS** dataset [12]. It contains hand-written symbol strokes. Besides, it has included a library that can render those symbols. Unfortunately, time does not allow us to experiment with generation on these symbols, but we again tested (by eye) the automatic finding of attachment points (Section 3.1), so at least it could work with it.

2.4 Mashcima

Mashcima [1, 2] is a music engraving system created by Mayer, Mgr. Jiří Mayer. It takes music encoded to string and images (with denoted attachment points) from MUSCIMA++ (Section 2.3) and outputs music engraved to a music notation. We aim to replace those images from MUSCIMA++ and discover if it improves the performance of the model trained on top of Mashcima. See Tables 2.1 and 2.2.

Layer	Shape	Note
Input	`w` × 64 × 1	
Convolution	`w` × 64 × 16	Kernel 5x5
Max pooling	`w`/2 × 32 × 16	Stride 2,2
Convolution	`w`/2 × 32 × 32	Kernel 5x5
Max pooling	`w`/4 × 16 × 32	Stride 2,2
Convolution	`w`/4 × 16 × 64	Kernel 5x5
Max pooling	`w`/4 × 8 × 64	Stride 1,2
Convolution	`w`/4 × 8 × 128	Kernel 3x3
Max pooling	`w`/4 × 4 × 128	Stride 1,2
Convolution	`w`/4 × 4 × 128	Kernel 3x3
Max pooling	`w`/4 × 2 × 128	Stride 1,2
Convolution	`w`/4 × 2 × 256	Kernel 3x3
Max pooling	`w`/4 × 1 × 256	Stride 1,2
Reshape	`w`/4 × 256	
BLSTM	`w`/4 × 256 + `w`/4 × 256	Droupout
Concatenate	`w`/4 × 512	
Fully connected	`w`/4 × `num_classes + 1`	No activation function
CTC	≤ `w`/4	

Letter `w` stands for input image width. BLSTM means bidirectional recurrent network with LSTM cells. The dropout on the BLSTM layer is important because the model does not converge without it. At the end, `num_classes + 1` means the number of output classes (vocabulary size) plus the blank symbol required for connectionist temporal classification (CTC). All activation functions are ReLU. Convolutional layers do not have an activation function, only pooling layers do. All the details can be seen in the source code.

Table 2.1 Architecture of the model [1, 2]. Taken from the thesis [1].

Parameter	Value
Learning rate	0.001
β_1	0.9
β_2	0.999
ε	10^{-8}

Table 2.2 The parameters of Adam optimizer. Taken from the thesis [1]. The model is trained on random music notation or/and on incipits of music compositions from the PrIMuS dataset [4]. Batches of 10 incipits and training set of 63 000 incipits (respectively twice them) are used.

2.5 Image operations

Besides trivial operations like padding and automatic cropping images, we need many more complex image operations. Open Source Computer Vision library [13] implements those operations. We use: *connected components* find, area *moments* (for finding the area center), morphological *erosion* (or *dilatation*), *distance transformation* (often called distance map), and *Hough circles* transformation.

By a *binary image*, we mean the image having only black and white pixels. White pixels often represent the "significant part of the image", or pixels for which some criterion holds (`True` output of element-wise condition).

A **connected component** (in short, a component) of a binary image is any maximal area of white pixels that fulfills that any pixel of it is reachable from any pixel of it by simply moving along the area in the basic four directions (up, down, right, left).

The **moment** m_{ij} is the sum (across all pixels of the area) of the x coordinate to the i-th power multiplied by the y coordinate to the j-th power multiplied by the value of the pixel. We use it for a component of a binary image, so the value is always 1. We take the moment with $i = j = 0$, which gives us the number of pixels, and one with $i = 1$ and $j = 0$ (or $j = 1$ and $i = 0$, respectively). As m_{10}/m_{00} and m_{01}/m_{00} are the x and the y coordinate of the centroid of area, the best candidate for something we intuitively call "center of area".

Morphological erosion (or simply erosion) takes a *kernel* – some shape, in our case, always square. We put this kernel in every pixel of a binary image[4], this pixel (on the transformed binary image) is white iff every pixel under the kernel (on the original image) is white. Dilatation is "*any* pixel under the kernel is white". Thus, the erosion is "erosion" (reduction) of the white area (enlargement of the black area), whereas the dilatation is "dilatation" (enlargement) of the white area (reduction of the black one).

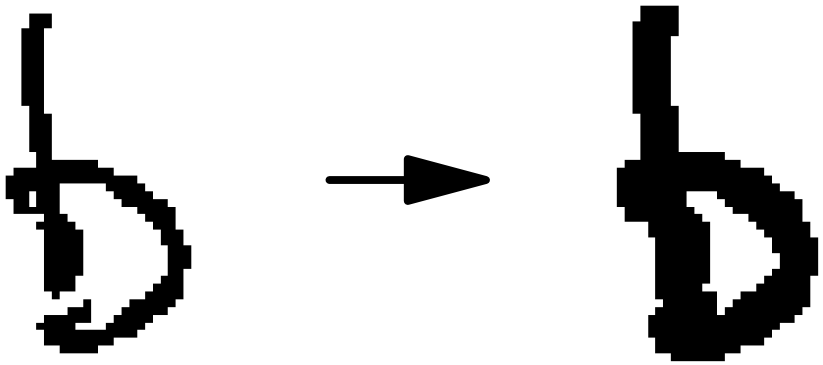


Figure 2.6 Erosion (with 2×2 kernel) applied on the image of a flat from the HOMUS dataset [12, image 96-53, thickness 3]. Notice the big area's closing and the small one's removal.

Distance transformation (also distance map) of a binary image converts white parts to distances from the nearest (in the chosen metric) black pixel. We can use it for finding big, filled areas.

Hough circles transformation is a method for finding circles and shapes reminding circles. It is not important how it works (OpenCV already implements it), we only

[4]The erosion is generalized for grayscale and even for colored images too, but we only need it for binary images.

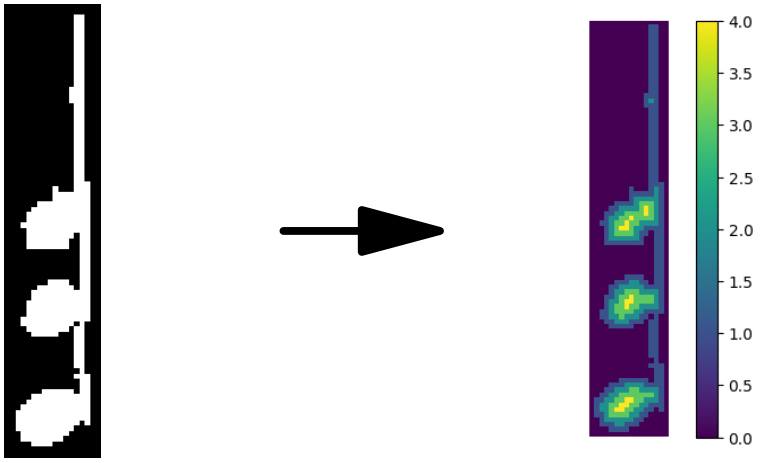


Figure 2.7 Distance transformation applied on the (inverted) image of a quarter-note from the Rebelo dataset [3]. Notice the three connected components of the two lightest colors (the two greatest distances).

need to know that it returns the centers and the radii of supposed circles in the image. We set it in the way it finds circles with adequate radius and everything which reminds circle (because people draw very imprecisely).

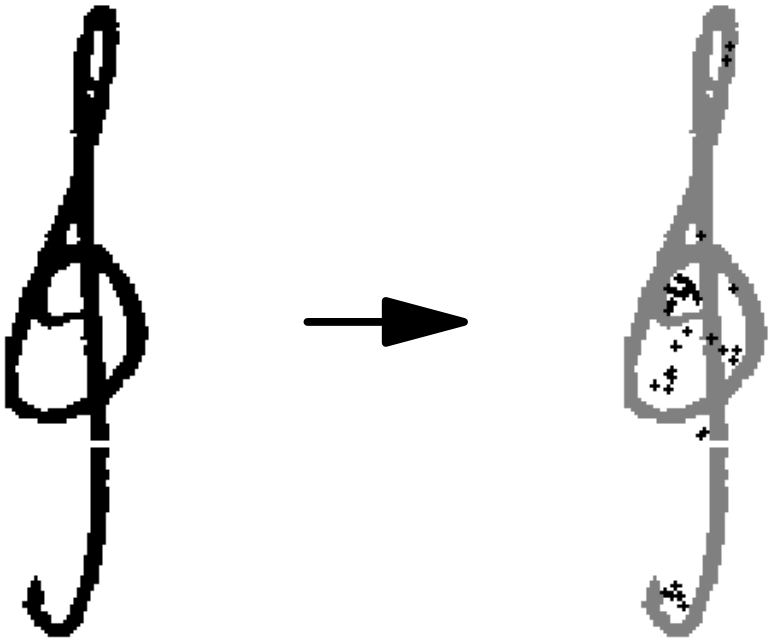


Figure 2.8 Centers (radii are omitted) of Hough circles transformation applied on the image of a G-clef from the MUSCIMA++ dataset [11, 10, image w38 p18 id656]. (Used parameters: `minDist` = 1, `param1` = 100, `param2` = 1, `minRadius` = 5, `maxRadius` = 10)

Chapter 3

Methodology

We want to train generative neural networks (from Section 2.1) to generate music symbols (and potentially other symbols). We examine this in Section 3.2.

For simply generating symbols, it suffices to use the Rebelo dataset [3] (Section 2.3) as it is. However, for better performance, it is good to pre-process images to have the same features (for example, the noteheads) in the same positions. This is also necessary for using those symbols in Mashcima (Section 2.4). We describe this pre-processing in Section 3.1.

Finally, when we get images from our models, we can post-process them so they do not have large "empty" background and are binarized. Moreover, we need to prepare them for Mashcima. Sections 3.3 and 3.4 are about these post-processing operations.

3.1 Automatic finding attachment point of symbols

As written in Chapter 2, for using Mashcima, we need to know where the symbols' attachment points are. We could add the coordinates of those points as other data to generate. However, we still need to find those points for training this.

So we chose a unifying position of attachment points across every symbol class. (Except for the other end of the note stem because we can find it easily as the highest point in the image. Thus every symbol has at most one attachment point.) We chose the center of the image as the unified position. This way, we can process every image separately, and if we need to put the symbol in a bigger image[1], we pad it symmetrically, and the center stays in the center.

To find those points, we use the already mentioned image operations. Symbols without attachment points, we leave as they are. Other symbols process as follows:

3.1.1 Accidentals

Accidentals have the point in the center of "their single component which they surround". It sounds easy, but there is a problem – people hardly ever draw accidentals this way. They often do not close the accidental, and (especially for flat) they draw other closed components elsewhere.

So we apply the image erosion until the image has exactly one component not touching the image's border. If it does not happen, we skip this image. Then we find the center of this component.

[1] Our generative NNs have a fixed size of the input.

Processing of **flats** differs from processing of sharps and naturals in one little thing: flats are more likely not closed and with other components. So when processing flats, we start with erosion and then repeat finding and counting components and the subsequent erosion. (And only if it fails, then we look at the components of the original image.)

Whereas **naturals** and **sharps** are often closed, and moreover, eroding can more often create new (incorrect) closed components. So we start with checking if the original image has exactly one component not touching the border. If it does not, we proceed as previously.

3.1.2 Clefs

Processing of **C-clef** is trivial because it has the attachment point in the center of the image. G-clef and F-clef are much more difficult.

F-clef should have two dots, and we can find the attachment point between them. However, sometimes, people do not write them. The most pathological case is F-clef which has exactly one dot. We remove those images manually.

Thus we must distinguish two cases – with two dots and without them. We cannot do this by looking at the rightmost column of pixels because people often do not write dots one over a second one. So we decided to detect them by projecting the image to the x axis and searching for the first (second can be between dots) "hole" in the fourth quarter (in the left of it, "hole" is more likely a defect in the curve part of the clef.)

If we find the "hole", we project everything on the right of it to the y axis and average coordinates of the painted part of the projection. If we do not find it, we take the last two (for smoothing) columns of pixels and do the same.

For improvement in detecting dots, we erode the image once. After it, dots rarely touch the curve part of the key. 'Once' is essential because by applying erosion twice, we create "holes" in the curve part too, and get false positives.

So we find the y coordinate of the attachment point. It is sufficient for drawing F-clef on staff. However, for better performance of generative NN, we need to choose a unified x coordinate of the attachment point, whether the clef has the dots or not. We choose it as the first "empty column" to the right from the curve part of the clef. Because if it has the dots, then we already found this column. If it has not, then this x coordinate is simply the image's width (assuming we crop the image).

G-clef has many "weird" hand-written forms. Thus, it is almost impossible to rigorously tell where it has the attachment point. However, a good heuristic is 'in the center of the most circular part.'

We realized this by using the Hough circles filter. We set it in the way it finds many and many "circles". For filtering most false positives, we set that all centers above one-half of the image belong to the upper loop of the clef. All centers below the fourth fifth belong to the possible circle at the bottom end of the clef. From the remaining centers, we take the median (separately in two coordinates) as the desired point.

3.1.3 Notes

As already written above, we do not deal with the end of the stem in this subsection. Thus we only search for noteheads (and their centers).

Quarter notes are quite easy. We can take a distance map of the black part of the image (find the distance of pixels to the nearest white pixel). Then take pixels with maximum distance and ones with that distance minus one (for better detection of notes with multiple noteheads).

When those pixels form one component, we declare this component's center as the center of the notehead. Otherwise, the note likely has two or more noteheads, so we throw it away.

For unification, if the notehead is in the image's upper half (stem points down), we mirror the image around the x axis. If the stem is on the left half (its upper end is at the left from the center of the notehead), then we mirror the image around the y axis.

One can say we handle **half notes** the same way as quarter ones. Unfortunately, we cannot use distant transform. Instead, we use a method similar to finding attachment points in accidentals. However, this time, we erode the image only once to close incomplete noteheads. Other erosions can hide more noteheads.

After this erosion, we find components not touching the image's border. If there is only one, we have our notehead. Otherwise, we throw away the image. Then we normalize the stem orientation as we normalize it in quarter notes.

The Rebelo dataset does not contain **whole notes**, but we can help with a little trick – we have half notes with the found notehead. So we can take this notehead and remove the stem. We do it by going from the center of the notehead, and we delete all above the first cross of a black area.

We can handle **eighth notes** (sixteen notes, ...) like quarter notes. However, there is one problem – we draw note flags always to the right. Thus an eighth note with the stem pointing up differs from an eighth note with the stem pointing down. We solved this by creating two categories – eighth notes with the stem pointing up and ones with the stem pointing down. We differentiate between them by the position of the notehead in the original image.

Moreover, we manually removed the sixteenth and the thirty-second notes from the eighth notes (Rebelo has only "notesFlags") because there is a high chance of "contamination". (We did not manually fix other categories because there was no significant bug.)

3.2 Implementation of generative NNs and their settings

We implement generative NN (from Section 2.1). In category-conditioned versions of VAE and AAE, we made a change: to the decoder, we input a one-hot label instead of the "categorical part" of the encoder's output. (We have got a small training set. Thus, it is difficult to train the encoder as a good classifier.) Furthermore, the conditioned AAE is implemented the same. So it slightly differs from the paper [9] from which we took the AAE. In that paper, the encoder does not output the category. Nevertheless, in our way, it is more prepared for semi-supervised learning (part of the data is labeled, another part not).

However, we do not use the category-conditioned versions for most cases. With a few exceptions, we take each category separately and train one independent non-categorical model for this category.

For this work, we chose that all convolutional layers (even transposed) have strides

equal to 3 and kernel 5×5. Moreover, for the encoder and the GAN discriminator, we use convolutional layers with channels 4, 16, and 64 and no hidden dense layer (this means one dense layer as the last layer). The decoder has the same in reverse, so no "hidden" dense layers (one from input to reshape), reshaping to 64 channels, convolutional layers with 16 and 4 channels, and one last to one channel (the image). The discriminator in AAE has one hidden dense layer with 128 units (and another dense layer as the last layer).

For a more straightforward computation of shapes in the decoder, we choose that transposed convolutional layers must not have any padding. Thus the images must have size divisible by the "magnification" of those layers. So we pad the images to multiplies of this magnification (here $3 \cdot 3 \cdot 3 = 27$).

We use 150 epochs and batches of 5 (respectively 50 for category-conditioned) images.

3.3 Binarization

Music notation images are often binarized for dealing with shadows and other noises. Thus we want to train models for optical music recognition on binarized images – we need binarized images of symbols.

Generative NNs output grayscale images even we train them on binarized images. Thus we need some method of making binarized images from grayscales. We tried an adaptive threshold (from OpenCV [13]). Nevertheless, it creates much noise because generative NN often creates slightly brighter areas in the background. Even using noise reduction (we tried blurring the image with the median filter) did not help because reducing noise breaks the desired part of the image.

Finally, we did not do more experiments with adaptive threshold because it turned out that a simple constant threshold [2] suffices. We use constant 0.2 (from the range 0.0–1.0) since our best generative NNs have more trouble with the lightness of an image than with the sharpness of it.

3.4 Generating output for Mashcima

Together with J. Mayer, we modified Mashcima (Section 2.4) to take only part of the original images of symbols and load the provided new images. We also add functionality for the two versions of eighth notes.

It remains to automatically crop our images (crop the image to the smallest rectangle containing all non-background pixels), as Mashcima uses image sizes for correct spacing between symbols. Therefore, we also have to output coordinates of the attachment point (a text file containing x-coordinate 'space' y-coordinate) because it is not at the center of the image anymore.

Moreover, we need to supply half, quarter, and eighth notes with the other end of the stem. We can find it as the highest non-background point. However, this way, we can take a noise pixel. So we first erode[3] the image, then we take the highest point.

[2] All pixels with a value above the chosen constant number (the threshold) are considered white others are black.

[3] We work with inverted images: the background is black, and the note is white. So the erosion acts on the note and noise.

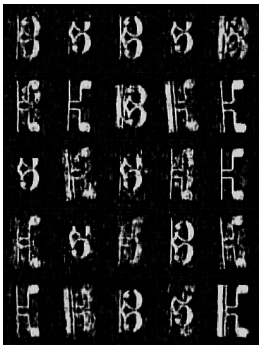
Chapter 4

Experiments

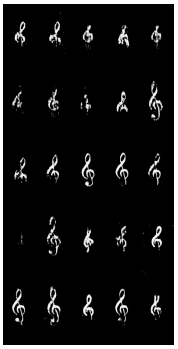
4.1 Realistic generated images

Initially, we measured the performance of the generative neural networks by eye. Figures 4.1 and 4.2 show examples of synthetic images from the categories we aim to replace in Mashcima (Section 2.4). We can see that increasing of dimensions[1] of the latent space increases variability (and AAE is slightly more variable than VAE).

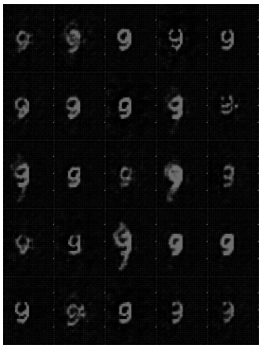
In Figure 4.2, we can observe the preservation of style (smaller symbols are at the left in all of those images, and upper symbols are slightly more "italic") typical for training one NN for all categories. And we can also notice that the category-conditioned generative NN (Figure 4.2) deals better with a tiny training set, see Subfigures (c).



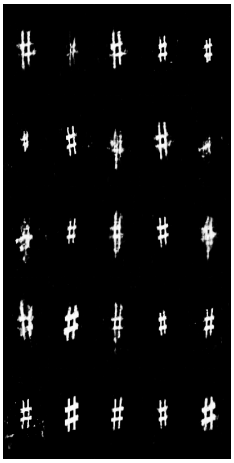
(a) C-clef



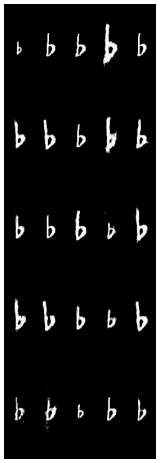
(b) G-clef



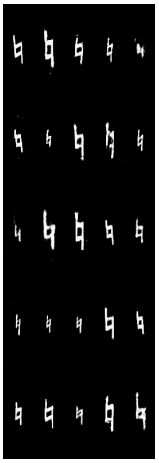
(c) F-clef



(d) Sharp

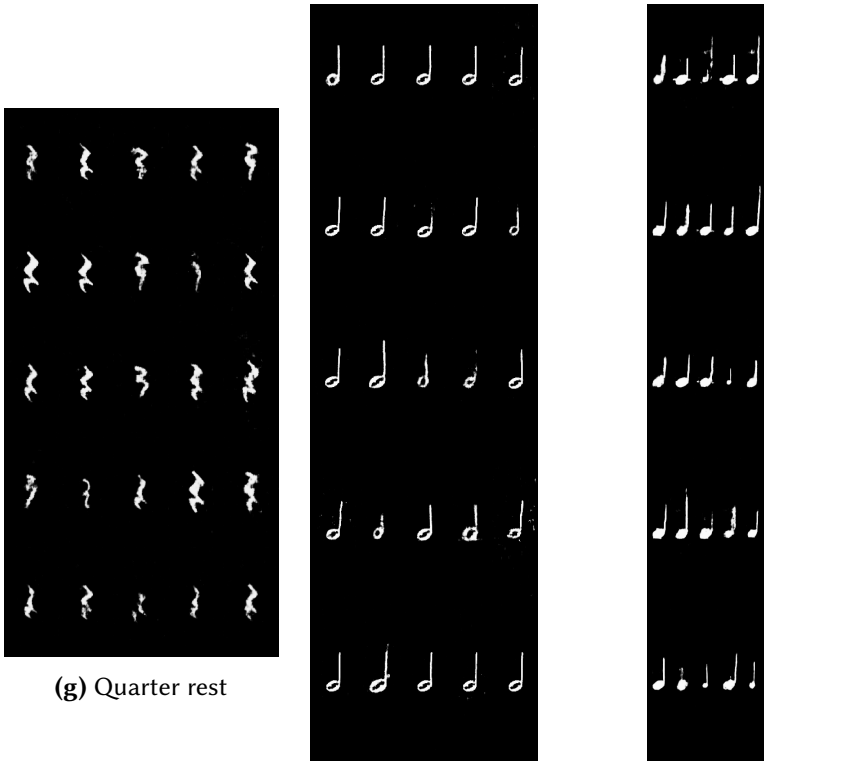


(e) Flat



(f) Natural

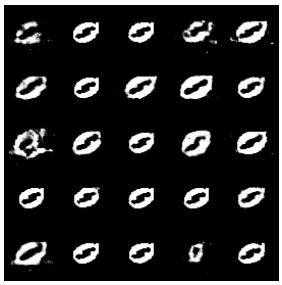
[1]We write '2D (4D, 100D, ...) latent space' for the latent space where points consist of 2 (4, 100, ...) real numbers, each independently sampled from the standard normal distribution.



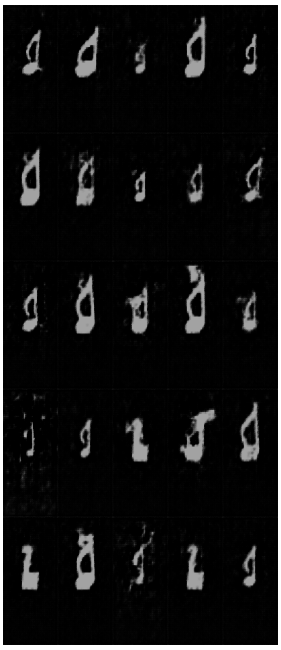
(g) Quarter rest

(h) Half-note

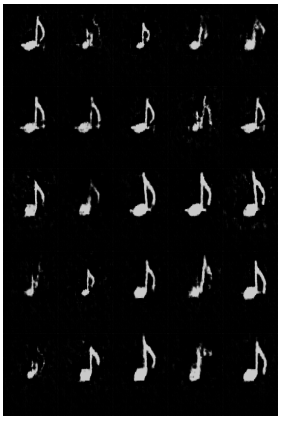
(i) Quarter note



(j) Whole note

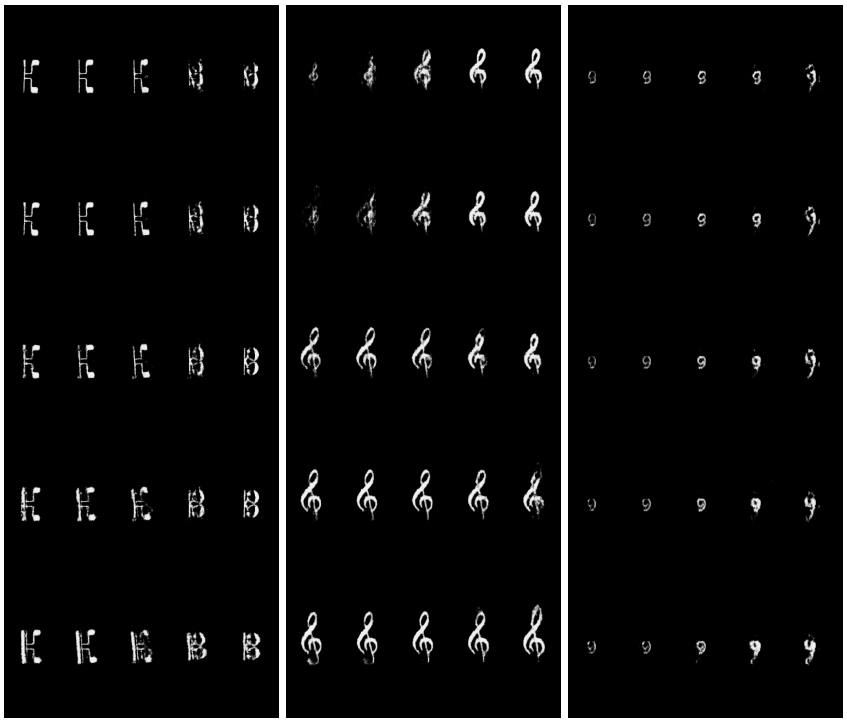


(k) Eight note down

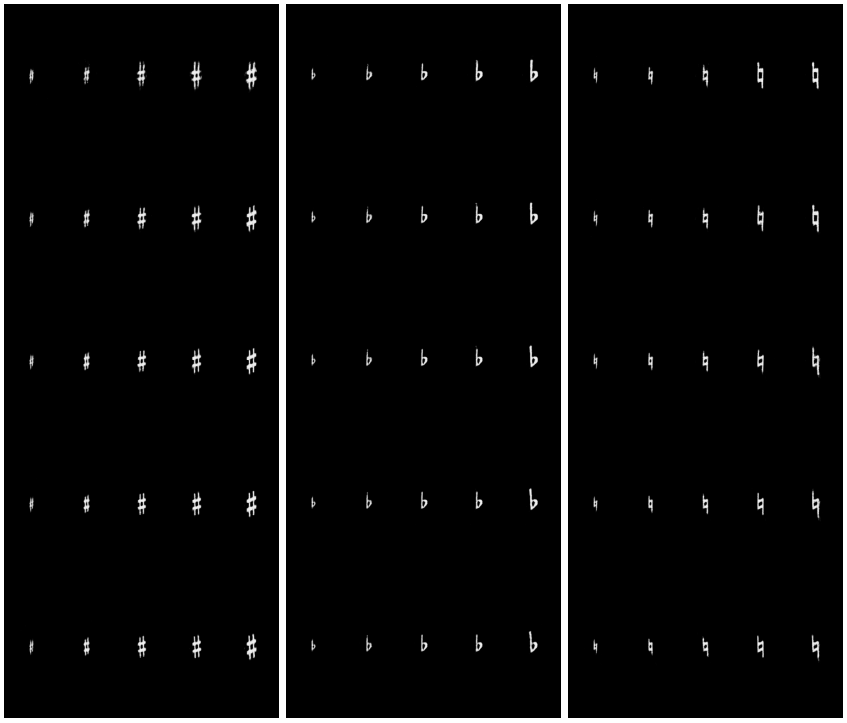


(l) Eight note up

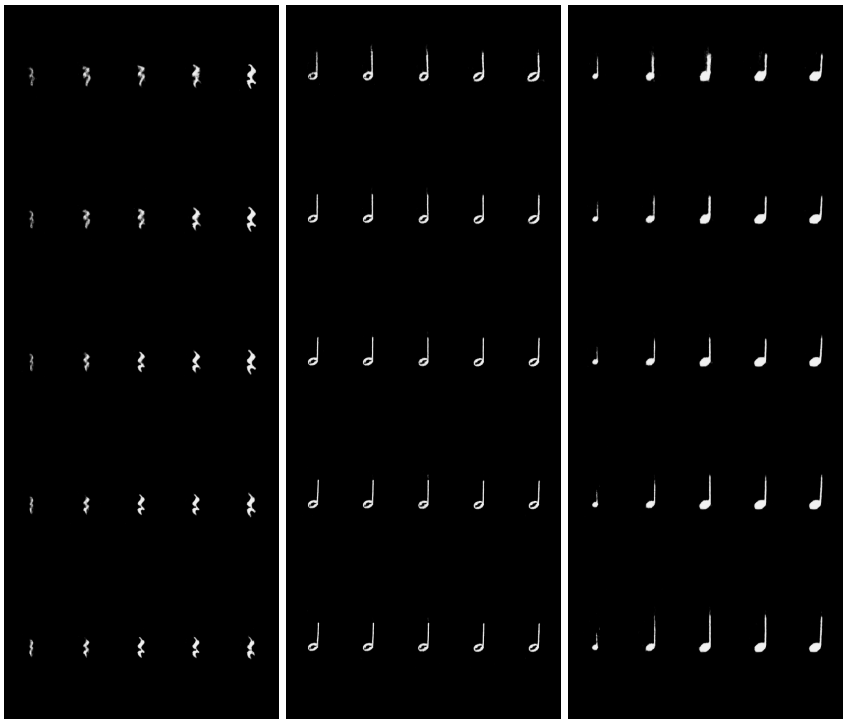
Figure 4.1 Images generated by AAE with the 4D latent space by random sampling from the 4D standard normal distribution



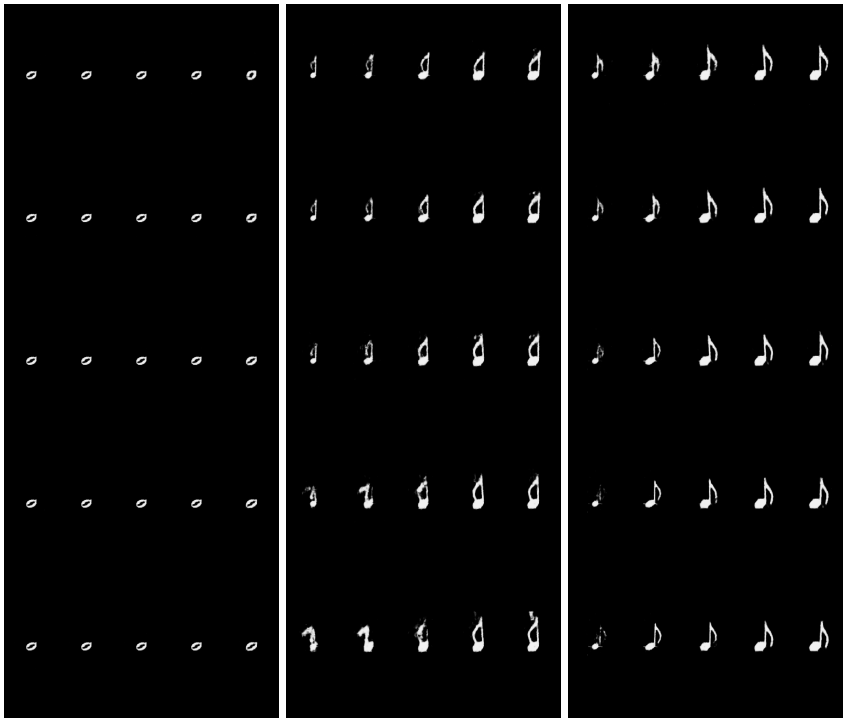
(a) C-clef **(b)** G-clef **(c)** F-clef



(d) Sharp **(e)** Flat **(f)** Natural



(g) Quarter rest **(h)** Half-note **(i)** Quarter note



(j) Whole note **(k)** Eight note down **(l)** Eight note up

Figure 4.2 Images generated by category-conditioned VAE with the 2D latent space by sampling uniform "grid" from the square $[-1, 1] \times [-1, 1]$

4.2 Baseline

Except for measuring images by eye, we need an exact method for measuring the "performance" of those images in the optical music recognition field. For this, we use Mashcima (see Section 2.4), and the way, Mashcima was tested – we train the model (see Section 2.4) on our data and see its performance.

4.2.1 Symbol error rate

A *symbol error rate* (SER, also known as normalized edit distance and normalized Levenshtein distance) is used to measure the performance of the optical music recognition model and, generally, any model outputting a sequence.

We define it as the smallest number of edits (insertions, deletions, and substitutions) needed to convert one sequence (the outputted one) to a second sequence (the golden one). Moreover, we normalize it by dividing it by the number of symbols in the golden sequence so it is comparable across various lengths of golden sequences.

4.2.2 The original experiments

For testing Mashcima, the model (Section 2.4) is used. The model is trained in four different experiments:

1. It takes 63000 annotations from the PrIMuS dataset [4] (see Section 2.3) and uses Mashcima to create images of them. Then it trains the model on this.
2. It creates 63000 random synthetic incipits, runs Mashcima on them, and uses this for training.
3. It takes half of the data from the first experiment and half of the data from the second one. (Still 63000.)
4. Finally, it uses all the data from both experiments.

The fourth experiment gives the best model. However, we use the third experiment because it is only slightly worse and takes half the time.

Original experiments use 20 epochs (respective 10 epochs in the fourth experiment) and choose the model from the epoch with the best validation error. We use 12 epochs (twice the optimal epoch for the third experiment) because validation error forms a "nice parabola", and the twentieth epoch is already on the "other side". Moreover, we have more complex data, so the sixth epoch (the optimal one for the third experiment) is too early.

The model is evaluated on three evaluation data:

- primarily, on some annotated images from CVC-MUSCIMA [10] (Section 2.3) (by writers Mashcima does not use for generating training data);
- also on some incipits from printed PrIMuS [4] (Section 2.3) unseen by the model;
- finally, on some real non-CVC-MUSCIMA scans of hand-written music notation [14].

We ran these evaluations on the trained models (from his repository [15]) and got values in Table 4.1

	CVC-MUSCIMA	Printed PrIMuS	Hand-written
Experiment 1:	0.338	0.560	0.593
Experiment 2:	0.285	0.784	0.605
Experiment 3:	0.262	0.691	0.469
Experiment 4:	0.254	0.613	0.509

Table 4.1 SER of the original experiments on three evaluation data (the minor differences to the thesis [1] can be caused by different versions of used Python libraries or changes in the datasets)

4.3 Mixing original and generated images is the best

We propose three experiments with mixing MUSCIMA++ images with our synthetic images:

- Take only generated images in the same quantity as the symbols in MUSCIMA++;
- Take half of the images from MUSCIMA++ and replenish them with our generated images to their original quantity;
- Take full MUSCIMA++ and add generated images in the same quantity;
- (Take only images from MUSCIMA++. This coincides with the original experiment 3.)

We generated images by AAE with the 4D latent space and got results in Table 4.2. We see that the "half-half" option gives the best results beating Experiment 3 on PrIMuS and having only slightly worse performance on CVC-MUSCIMA and the real hand-written images.

MUSCIMA++	generated	CVC-MUSCIMA	Printed PrIMuS	Hand-written
100%	0%	0.262	0.691	0.469
0%	100%	0.413	0.748	0.590
50%	50%	0.278	0.517	0.510
100%	100%	0.281	0.562	0.593

Table 4.2 SER of the experiment 3 with different fractions of MUSCIMA++ and synthetically generated images

4.4 Generating synthetic images helps

To prove that generative neural networks are not just a waste of time, we compare generated images with the training ones. We already have the reference performance of the model on images generated by AAE with 4D latent space. Thus it remains considering the Rebelo images [3] (Section sec:datasets), after the automatic finding of attachment points. We do it in two experiments:

- At first, we follow up the best option from the previous experiment – we take half of the original (MUSCIMA++) images and fill them with the same quantity

of the new ones. However, the Rebelo dataset does not have enough images. So we repeat them to fill the necessary quantity.[2]

- In the second experiment, we take up with the last version of the experiment from Section 4.3 – full MUSCIMA++ and "full" new images. In generated images, "full" means the same quantity as in the MUSCIMA++. But now, with the Rebelo dataset images, we simply take all images.

The first experiment is "we take the same ratios", and the second one is "we use all images we have". Table 4.3 shows that in both experiments, synthetic images cause the model to work on PrIMuS much better. (And differences on the rest of evaluating data are small.) So it definitely makes sense to examine generative neural networks in this context.

MUSCIMA++	New	CVC-MUSCIMA	Printed PrIMuS	Hand-written
50%	Generated 50%	0.278	0.517	0.510
50%	Rebelo repeated	0.274	0.570	0.491
100%	Generated 100%	0.281	0.562	0.593
100%	Rebelo all	0.287	0.771	0.574

Table 4.3 SER of half original and half generated/Rebelo images, and full original and the same quantity generated/full Rebelo images

4.5 Bigger latent space is better

We also examine the effect of the latent space size on the success of the model (Section 2.4). We choose three sizes: 2D (human-pleasant[3]), 4D (reasonable size), and 100D (extreme size). However, AAE with the 100D latent space cannot learn clefs, so we also use VAE for comparison.

In Table 4.4, we can notice better results on PrIMuS with bigger latent space. Performance on CVC-MUSCIMA slightly worsens, however, it is most likely caused by more difficult train data (so images from MUSCIMA++ have smaller significance). In addition, VAE with the 100D works best (of these five experiments) on non-CVC-MUSCIMA hand-written images.

4.6 AAE is better than VAE

We can use Table 4.4 once more – to compare AAE and VAE. It is clear that AAE outperforms VAE in all directions (except for the 100D latent space).

[2]This way, we preserve the ratio between taking the MUSCIMA++ images and the Rebelo dataset images. Nevertheless, a particular image from the Rebelo dataset is more frequent than a particular one from MUSCIMA++.

[3]2D latent space is human-pleasant because it is too small for learning noise and can be visualized as a grid.

	CVC-MUSCIMA	Printed PrIMuS	Hand-written
AAE, 2D	0.273	0.576	0.507
AAE, 4D	0.278	0.517	0.510
AAE, 100D	———	———	———
VAE, 2D	0.281	0.586	0.528
VAE, 4D	0.284	0.725	0.544
VAE, 100D	0.288	0.550	0.501

Table 4.4 SER of various sizes of the latent space and various types of generative NN

4.7 Category-conditioned is better than training each category separately

Training an individual model for every category seems like a good idea, but the opposite is likely true. We again used AAE with the 4D latent space and compared its category-conditioned version with one model for each category. And categorical version seems to perform better than the non-categorical one – we again get better results on PrIMuS and slightly worse results on the rest. See Table 4.5.

	CVC-MUSCIMA	Printed PrIMuS	Hand-written
non-categorical	0.278	0.517	0.510
categorical	0.280	0.498	0.511

Table 4.5 SER of an independent model for every category and model for all categories, both AAE with the 4D latent space

4.8 Beating the original experiments on PrIMuS

As we can already notice in the preceding text and tables, with generated images (half of them, half original), we got lower SER (the non-categorical CAAE 0.517; the categorical 0.498) than with original CVC-MUSCIMA images (stated in the thesis [1] – experiment 4: 0.613; the same experiment – experiment 3: 0.691; the really best original experiment – experiment 1: 0.560).

Tables 4.6 and 4.7 shows that we improve in the absolute numbers in quarter notes (**q**). However, we can notice a worsening in them in the relative numbers. Much greater is improvement in half notes (**h**), sharps (#), and naturals (**N**). And we also find our tragical performance on clefs.

Token	Errors	Proportional
q	349	11.2%
h	329	10.6%
#	208	6.7%
=s=	208	6.7%
e	175	5.6%
=t=	166	5.3%
=s	139	4.5%
s=	134	4.3%
*	124	4.0%
)	117	3.8%
e=	116	3.7%
=e	88	2.8%
=t	83	2.7%
s	81	2.6%
.	76	2.4%
w	72	2.3%
=e=	61	2.0%
t=	58	1.9%
(	50	1.6%
b	45	1.4%
time.C/	43	1.4%
er	39	1.3%
time.4	36	1.2%
|	33	1.1%
N	33	1.1%
time.3	32	1.0%
qr	30	1.0%
time.8	29	0.9%
clef.C	28	0.9%
sr	24	0.8%
time.C	24	0.8%
time.2	17	0.5%
br	11	0.4%
lr	10	0.3%
**	9	0.3%
hr	9	0.3%
clef.F	8	0.3%
wr	6	0.2%
t	6	0.2%
time.6	3	0.1%
time.7	2	0.1%
clef.G	2	0.1%
time.0	1	0.0%
time.9	1	0.0%
time.1	1	0.0%

(a) Experiment 3

Token	Errors	Proportional
q	290	12.3%
e	197	8.3%
=s=	195	8.3%
=t=	172	7.3%
=e=	138	5.8%
=s	124	5.3%
s=	115	4.9%
h	111	4.7%
=t	87	3.7%
e=	82	3.5%
#	75	3.2%
t=	74	3.1%
=e	70	3.0%
.	68	2.9%
*	54	2.3%
b	44	1.9%
s	44	1.9%
)	41	1.7%
w	41	1.7%
time.4	35	1.5%
clef.C	29	1.2%
time.8	27	1.1%
time.3	25	1.1%
time.C/	23	1.0%
time.C	23	1.0%
qr	21	0.9%
|	17	0.7%
time.2	16	0.7%
er	15	0.6%
(	15	0.6%
clef.F	15	0.6%
wr	13	0.6%
N	13	0.6%
clef.G	11	0.5%
time.6	8	0.3%
br	8	0.3%
**	6	0.3%
sr	6	0.3%
hr	6	0.3%
t	4	0.2%
lr	1	0.0%
time.7	1	0.0%

(b) Categorical AAE, 4D latent space, 0.5 and 0.5

Table 4.6 Errors in PrIMuS evaluation by tokens proportionally to the sum of errors

Token	Errors	Proportional
br	11	∞%
**	9	∞%
t	6	∞%
time.7	2	∞%
time.0	1	∞%
time.9	1	∞%
time.1	1	∞%
lr	10	1000.0%
t=	58	966.7%
=t=	166	922.2%
=t	83	830.0%
)	117	650.0%
wr	6	600.0%
sr	24	480.0%
(	50	277.8%
#	208	253.7%
h	329	222.3%
*	124	221.4%
w	72	200.0%
time.C/	43	195.5%
N	33	194.1%
s	81	192.9%
time.8	29	181.2%
clef.F	8	160.0%
s=	134	154.0%
=s=	208	118.9%
time.3	32	118.5%
.	76	115.2%
=s	139	113.9%
time.2	17	113.3%
time.4	36	100.0%
e	175	97.2%
qr	30	96.8%
q	349	94.1%
clef.C	28	93.3%
time.C	24	85.7%
=e=	61	76.2%
er	39	73.6%
e=	116	70.7%
=e	88	70.4%
hr	9	60.0%
time.6	3	50.0%
b	45	45.5%
|	33	10.2%
clef.G	2	3.1%

(a) Experiment 3

Token	Errors	Proportional
br	8	∞%
**	6	∞%
t	4	∞%
time.7	1	∞%
wr	13	1300.0%
t=	74	1233.3%
=t=	172	955.6%
=t	87	870.0%
clef.F	15	300.0%
)	41	227.8%
=e=	138	172.5%
time.8	27	168.8%
time.6	8	133.3%
s=	115	132.2%
sr	6	120.0%
w	41	113.9%
=s=	195	111.4%
e	197	109.4%
time.2	16	106.7%
s	44	104.8%
time.C/	23	104.5%
.	68	103.0%
=s	124	101.6%
lr	1	100.0%
time.4	35	97.2%
clef.C	29	96.7%
*	54	96.4%
time.3	25	92.6%
#	75	91.5%
(	15	83.3%
time.C	23	82.1%
q	290	78.2%
N	13	76.5%
h	111	75.0%
qr	21	67.7%
=e	70	56.0%
e=	82	50.0%
b	44	44.4%
hr	6	40.0%
er	15	28.3%
clef.G	11	16.9%
|	17	5.3%

(b) Categorical AAE, 4D latent space, 0.5 and 0.5

Table 4.7 Errors in PrIMuS evaluation by tokens proportionally to the token quantity

Chapter 5

Conclusion

We used generative neural networks on music symbol images and got the outstanding results. Also, we developed a quite good method for automatic finding attachment points of chosen music symbols. We tested our work on the OMR model (from Section 2.4) and showed that a bigger variation of symbol images (obtainable from generative NN) makes sense: Although we did not improve performance on the original dataset, the results on the PrIMuS dataset seem promising. However, we have done only a few experiments, and to confirm the results, we need to do more statistical experiments. Furthermore, of course, there is still a long way to a "good OMR models".

5.1 Future work

Apart from more statistical experiments, there are a lot of things to do next. We fixed the architecture of the generative NNs. Thus we do not have any notion about the impact of the architecture on the performance. We can change *strides*, *kernel*, *channels*, *units*, and *the number of layers*. We can add some other layers as *dropout* and *max pooling*. We can also replace the type of generative NN architecture (with, for example, *VAE/GAN* [16]). With our implementation of the category-conditioned versions, we are close to semi-supervised learning of symbol images (only a part of the data is annotated).

We had a problem with such a small symbol image dataset, so we could try to replace the dataset with the larger one (such as MUSCIMA++ [11, 10] or HOMUS [12]). People also have made significant progress in neural network mechanisms such as transformers. Thus we should do experiments with other models as well.

Bibliography

[1] Jiří Mayer. "Optical Music Recognition using Deep Neural Networks". bachelor thesis. Prague: Charles University, Faculty of Mathematics, Physics, Institute of Formal, and Applied Linguistics, 2020.

[2] Jiří Mayer and Pavel Pecina. "Synthesizing Training Data for Handwritten Music Recognition". In: *16th International Conference on Document Analysis and Recognition – ICDAR 2021. Lausanne, September 8-10*. Ed. by Josep Lladós, Daniel Lopresti, and Seiichi Uchida. Cham: Springer International Publishing, 2021, pp. 626–641. ISBN: 978-3-030-86334-0.

[3] A. Rebelo, G. Capela, and Jaime S. Cardoso. "Optical recognition of music symbols". In: *International Journal on Document Analysis and Recognition (IJDAR)* 13.1 (2009), pp. 19–31. DOI: 10.1007/s10032-009-0100-1. URL: https://doi.org/10.1007%2Fs10032-009-0100-1.

[4] Jorge Calvo-Zaragoza and David Rizo. "End-to-End Neural Optical Music Recognition of Monophonic Scores". In: *Applied Sciences* 8.4 (2018). ISSN: 2076-3417. DOI: 10.3390/app8040606. URL: https://www.mdpi.com/2076-3417/8/4/606.

[5] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[6] Ian J. Goodfellow et al. "Generative Adversarial Nets". In: *NIPS*. 2014.

[7] Dor Bank, Noam Koenigstein, and Raja Giryes. "Autoencoders". In: *ArXiv* abs/2003.05991 (2020).

[8] Diederik P. Kingma and Max Welling. "Auto-Encoding Variational Bayes". In: *CoRR* abs/1312.6114 (2013).

[9] Alireza Makhzani et al. "Adversarial Autoencoders". In: *ArXiv* abs/1511.05644 (2015).

[10] Alicia Fornés et al. "CVC-MUSCIMA: a ground truth of handwritten music score images for writer identification and staff removal". English. In: *International Journal on Document Analysis and Recognition (IJDAR)* 15.3 (2012), pp. 243–251. ISSN: 1433-2833. DOI: 10.1007/s10032-011-0168-2. URL: http://dx.doi.org/10.1007/s10032-011-0168-2.

[11] Jan Hajič jr. and Pavel Pecina. "The MUSCIMA++ Dataset for Handwritten Optical Music Recognition". In: *14th International Conference on Document Analysis and Recognition, ICDAR 2017, Kyoto, Japan, November 13 - 15, 2017*. Dept. of Computer Science and Intelligent Systems, Graduate School of Engineering, Osaka Prefecture University. New York, USA: IEEE Computer Society, 2017, pp. 39–46. ISBN: 978-1-5386-3586-5.

[12] Jorge Calvo-Zaragoza and Jose Oncina. "Recognition of Pen-Based Music Notation: The HOMUS Dataset". In: *2014 22nd International Conference on Pattern Recognition*. 2014, pp. 3038–3043. DOI: 10.1109/ICPR.2014.524.

[13] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[14] Author unknown, scan and annotations: Jiří Mayer. *Annotated images of Cavatine*. Images: https://github.com/Jirka-Mayer/BachelorThesis/tree/master/real-images, annotations: https://github.com/Jirka-Mayer/BachelorThesis/blob/master/app/real_annotations.py. Online; accessed 4 July 2023. 2020.

[15] Jiří Mayer. *Trained models from [1] and [2]*. https://github.com/Jirka-Mayer/BachelorThesis/releases/tag/v1.0.0. Online; accessed 15 July 2023. 2020.

[16] Anders Boesen Lindbo Larsen et al. "Autoencoding beyond Pixels Using a Learned Similarity Metric". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML'16. New York, NY, USA: JMLR.org, 2016, pp. 1558–1566.

Appendix A

Using NoteCopyist

After installing `requirements.txt`, we can directly run `python main.py`. The command `python main.py --help` explains the arguments. However, a good start is, for example:

```
python main.py --dataset crebelo --cat onecat --layers
    --conv_layers 64 16 4 --stride 3 --kernel 5 --multiply_of 27
    --batch 5 --network aae --epoch 150 --latent 4
```

The parameter `dataset` can be *mnist* for the MNIST dataset, *rebelo1* for the square Rebelo dataset, *rebelo2* for the original-sized Rebelo dataset, *crebelo* for rebelo2 automatically centered to the attachment points, and *other* for custom dataset (divided to directories by category) in directory `downloaded/other/`.

The parameter `cat` (a category) can be *basic* for unlabeled training, *onecat* also for unlabeled training, but for every category separately, or *cat* for category-conditioned training.

It trains the desired generative NN and creates a deep directory tree in the directory `out`. The deepest directories are `images`, where previews for each epoch and category are stored, and `parts` where the models are saved.[‡]

Then we can generate the images (and text files with the positions of the uncropped images' centers) by running:

```
python generate_images.py
    out/{dataset}/{cat}/{NN description}/parts/e{epoch}
    {output_dir} {number} --network {aae/vae/gan}
```

Finally, we need to add the other end of the stem to every image of a stemmed note. The following command serves this purpose (it adds a text file with the coordinates of stems' ends to every image in directories provided as arguments):

```
python add_other_end_of_stem.py
    {output_dir}/half-note {output_dir}/quarter-note
    {output_dir}/eighth-note-up {output_dir}/eighth-note-down
```

The other end of the stems works only for the normalized (crebelo) images.

[‡]The models can be loaded in Python by `generators.***.***.load_all(filename)`.