Generative Adversarial Networks for Multi-Instrument Music Synthesis

Master's Thesis
submitted by

Tobias Susetzky
susetzky@fim.uni-passau.de

Supervised and examined by

Univ.-Prof. Dr. Tomas Sauer

Chair of
Digital Image Processing
University of Passau, Germany

Co-examined by

Univ.-Prof. Dr. Michael Granitzer

Chair of Data Science
University of Passau, Germany

Abstract

Generative Adversarial Networks (GANs) recently succeeded in various tasks of humans' creative domain. In this master's thesis, based on a GAN a score-to-audio model is designed, built and analyzed. The model termed *orGAN* interprets sheet music by controlling on- and offsets, volume, instrumental interaction etc., taking the role of a human performer. It is capable of synthesizing scores of arbitrary length for arbitrary combinations of 13 types of instruments at once with a very high-quality sampling rate of 48kHz. For this, scores represented as pianorolls are transformed into a spectrogram. This way, generative model architectures from computer vision become applicable. After investigating the methodological concepts in depth, this master's thesis adapts a well-known image-to-image translation neural network architecture: *orGAN* is a conditional PatchGAN with fully-convolutional U-Net generator and an auxiliary instrument classifier. Experiments confirm the findings of pix2pix [27] regarding the patch size and show the superiority of the Multi-Scale Structural Similarity Index (MS-SSIM) over the L1 loss in a composite objective function. The instrument classifier is found to foster transfer learning from single- to multi-instrument play. Extensive human evaluation shows a superiority of *orGAN* over related work and state-of-the art synthesizers in naturalness, timbre and emotional expressiveness of the generated audio. Samples are available at `https://students.fim.uni-passau.de/~susetzky/organ/`.

Zusammenfassung

In dieser Masterarbeit wird ein auf einem Generative Adversarial Network (GAN) basierendes Modell zum Überführen von Musiknoten in Audio entwickelt, implementiert und analysiert. Das Modell mit der Bezeichnung *orGAN* nimmt die Rolle eines menschlichen Interpreten ein, indem es musikalische Parameter wie Lautstärke, Anschlag oder die Abstimmung verschiedener Instrumente selbstständig kontrolliert. Es bietet die Möglichkeit, für Partituren unterschiedlicher Länge hochfrequente 48kHz-Aufnahmen mit beliebigen Kombinationen von 13 Instrumententypen auf einmal zu erzeugen. Dafür werden sogenannte "Pianorolls" als Repräsentation der Partitur in ein Spektrogramm überführt. Nach einer tiefgehenden Beschreibung der zugrundeliegenden Konzepte wird die Anpassung einer GAN-Architektur aus der Bildverarbeitung beschrieben: *orGAN* verwendet eine PatchGAN-Achitektur mit Instrument Classifier im Discriminator, wobei der Generator eine Pianoroll als Bedingung erhält und eine U-Net-Struktur aus Convolutional Layers besitzt. Experimente bestätigen die Ergebnisse des pix2pix-Modells [27] bezüglich der dabei verwendeten Patch-Größe und zeigen, dass die Verwendung des Multi-Scale Structural Similarity Index (MS-SSIM) anstelle der L1-Metrik in der Loss-Funktion von Vorteil ist. Es zeigt sich auch, dass der Instrument-Classifier ein Transfer Learning von Einzelaufnahmen hin zu Audio mit mehreren Instrumenten fördert. Eine ausführliche Evaluation zeigt, dass *orGAN* verwandten Modellen und zwei herkömmlichen Synthesizern in Natürlichkeit, Klangfarbe und Emotionalität überlegen ist.

Acknowledgments

First, I want to express my gratitude to Professor Tomas Sauer at the University of Passau who supervised this work sharing my fascination for the topic as well as always providing helpful and precise advise.

I would also like to thank Bochen Li, Xinzhao Liu and their colleagues at the University of Rochester for providing me the URMP-Dataset, on which this work is based.

Further, I want to give special thanks to my fellow students and precious friends! In particular: Laura for her moral support and her musical expertise, Claudio for co-developing `audival`, a dedicated platform for human music evaluation, in the record-breaking time of six hours, Barbara for creatively naming this work "The *or-GAN*-Project" as well as eliminanting bugs in writing and all those who participated in the human evaluation process.

Last but by absolutely no means least, let me thank my parents for their patience and unconditional support not only while writing this thesis, but during my whole studies!

Contents

Chapter 1

Introduction

One of the most intense forms of expressing human intelligence and creativity is a musical performance: when playing an instrument, the musician not only has to translate the notes into a deliberately timed sequence of actions on the instrument. He or she also has to control the loudness, tempo, rhythm and pauses. It also has to be decided for every note individually whether, for instance, a tone should have a smooth, soft and slow on-/offset or should rather be played powerfully and without fading out. Via controlling all of these parameters simultaneously, the performer has a tremendous power on deciding what the actual intonation of a given score will be like. This can lead to sounds highly varying depending on age, gender and physical properties of the musician and its environment despite being all based on the very same music sheet. Further, there are strong dependencies on the musical genre and epoch as well as the assumptions professional musicians make about the intentions of the composer or the conductor. Above all, a musician can and will always express emotion via all of the mentioned parameters.

In short: humans do not simply *map* scores to audio in a deterministic manner but instead incorporate a broad variety of factors based on which they *interpret* musical scores. Additionally, the resulting sound is also determined by properties of the instrument where even little differences in material, manufacturing and environmental temperature can have large effects. In an ensemble or an orchestra, also interactions and interdependencies between performers will affect the resulting music.

All of the aforementioned makes the generation of music on a level which is comparable to the one of recordings of human performances a highly complex task. Despite, or maybe just because of this, this challenge has been continuously addressed for decades: there is a variety well-established commercial as well as open source synthesizers translating scores to audio of very high quality. Nevertheless, most of them require extensive and time-consuming manual fine tuning and customization through digital filters mimicking for instance the echo of different rooms. Even dedicated experts are required that apply those synthesizers and carefully mix

different sound channels. However, the results obtained this way rarely achieve the naturalness and emotional expressiveness of human performances. A cause might be that those synthesizers largely act deterministically or just carefully incorporate randomness which is then truely random instead of human-like. Therefore, such algorithms always play correctly sticking exactly to the given score while, in essence, often little "mistakes" or impurities are the key factors making a performance sound *human*.

On the other hand, over the last years machine learning techniques and so-called artificial intelligence have emerged dramatically and step by step have been successfully applied to a broad range of complex tasks of human domain. Starting from computer vision problems over tasks of extracting semantics from written or spoken word and predictive applications, artificial intelligence has already reached and in some cases even to exceeded human performance in cognitive tasks. Besides this, recently *generative* models have also emerged and spread widely headed by so-called Generative Adversarial Networks [20]. The latter are a combination of two artificial intelligence models competing against each other where one tries to produce realistic data while the other one strives for distinguishing those fakes from real world samples. Besides the underlying mathematical motivation this has the intuitive interpretation of an artist whose performance is continuously improved by feedback from a critic mimicking the learning process via feedback in the human brain.

In this master's thesis, a generative adversarial network architecture which has recently succeeded in conditional image synthesis is adopted to perform music synthesis approaching a level of human quality. The developed model, which has been named *orGAN*, will be able to transform scores into waveform audio via spectrograms by *interpreting* the sheet music without any manual fine-tuning. *orGAN* is *one single model* capable of performing audio synthesis for arbitrary combinations of *multiple instruments*. This includes playing arbitrary many instances of one type of instrument at once, for instance a violin orchestra. Another remarkable characteristic is that *orGAN* covers almost the full range of instruments of a classical orchestra including strings, woodwinds and brass. All of those instruments are commonly assumed to be more difficult to synthesize than others such as a piano or a drum as strings, for instance, have a very complex timbre and offer the musician a tremendous amount of tuneable parameters to express his or her interpretation of the sheet music. To be able to do this multi-instrument synthesis, this work investigates the use of a tensor representation of multi-track scores by stacking the well-knwon pianoroll [14] representations of all tracks so that the position in the stack indicates the type of instruments, i.e. the timbre to synthesize. This technique has already been used for score generation [13], but to the author's best knowledge it has not been used in a multi-instrument music synthesis GAN yet. Unlike other related machine learning models, *orGAN* will be able to produce audio with a high-quality sampling rate of 48.000 Hz.

After starting with a compact survey on related work (Chapter 2), the dataset on which this work is based, the *University of Rochester Multi-Modal Music Performance Dataset* [34], is described along with its usage as well as the pre- and post-processing for the *orGAN* model (Chapter 3). The latter includes a brief introduction to the foundations of signal processing relevant for this work (Section 3.2, Section 3.3) followed by a description and a pros-and-cons analysis of the used score representation (Section 3.4).

In Chapter 4, the theoretical concepts necessary to motivate, build and understand the generative adversarial network architecture used in *orGAN* are build up and investigated from scratch: starting with a formal model of a supervised machine learning task (Section 4.1), the Section 4.2 dives into detail about the process through which an artificial intelligence actually *learns* to perform a certain task. This covers basic considerations of optimization as well as a step-by-step description of the evolution of the so-called *Descent Algorithms* that perform the optimization through a gradient-based procedure. Accordingly, an algorithm to efficiently compute gradients for large chains of functions called *Back Propagation* is outlined (Section 4.2.3). Furthermore, a certain type of machine learning algorithms, that are in essence powerful function approximators is introduced (Section 4.3): those are so-called *Neural Networks* whose terminology arises from a biological motivation. Special forms of them, which are relevant for this thesis, will also be covered: in particular *Generative Adversarial Networks* (Section 4.4) used to actually generate new data, *Autoencoder* architectures (Section 4.5) for learning high-dimensional data representations or semantic input-to-output transforms as well as *Convolutional Neural Networks* capable of efficiently processing image-like data and respecting latent spatial information (Section 4.6). Once all those theoretical concepts have been established, Section 4.7 combines and applies them to form the architecture of the *orGAN* neural network.

This thesis also experiments with different configurations of the neural network model in Chapter 5: the discriminator of *orGAN*, i.e. the component which should learn to distinguish real from synthesized (fake) music, outputs not just a scalar random variable but instead a Markovian field of random variables, each one indicating the probability for *one patch* of the synthesized spectrogram to be real. In Section 5.2, different choices for the size of this Markovian field and correspondingly for the size of those patches are investigated. Inspired by [27], the *orGAN* model uses a performance objective which is made up of linearly combined similarity measures between real and fake data. Experiments with different weights for its components are outlined in Section 5.1. Further, this thesis investigates the ability of *orGAN* to transfer to multi-instrument performances after being trained on single-instrument samples (Section 5.3) as well the effects of including an additional component for instrument classification in the discriminator (Section 5.4).

In Chapter 6, the performance of *orGAN* is evaluated relying largely on human evaluation carried out using `audival`[1], a dedicated platform for human music evaluation developed as a side-project of this thesis. In particular, the proposed synthesis model is compared against related work and two off-the-shelf synthesizers, namely `musescore 3` and Apple's `Logic Pro` revealing its superior performance in many aspects.

Finally, Chapter 7 concludes this thesis with a brief summary of the achieved as well as an outline of remaining open questions, possible extensions of *orGAN* and challenges for future work.

[1] `https://audival.io`

Chapter 2

Related Work

As mentioned in the introduction, this thesis applies a generative model: in 2014, [20] introduced generative adversarial networks (GANs), a special type of neural networks where two sub-networks, a generator and a discriminator, compete against each other where the generator tries to mimic real world data while its adversary, the discriminator, is trained to distinguish between real and fake. Such a model allows artificial intelligence to actually create data from a random seed. Variants have been proposed where a GAN additionally receives some side information to condition the generation process [18]. Consequently, there have been approaches letting the discriminator "verify" this condition by including an auxiliary classifier [49].

GANs have been mainly applied for image synthesis at first. While the original GAN was proposed as a multi-layer perceptron, i.e. a neural network of fully-connected layers, for image processing the adversarial setting has been successfully transferred to deep convolutional architectures [59]. For deep convolutional neural networks itself, various architectures have been studied for a long time. Just to mention a few, so-called autoencoders which first downsample their input by an encoder sub-network, pass it through a bottleneck and then upsample it again in another sub-network, the decoder, have been found to be very useful for learning high-dimensional data representations, i.e. for compressing data [24]. They also succeeded in denoising input data [70] and have been successfully used in conjunction with deep convolutional architectures [72] and even in adversarial settings [59]. A particular successful structure here, the U-Net [61], lets the decoder architecture mirror the one of the encoder and introduces additional "skip-connections" between corresponding encoder and decoder layers to improve gradient flow. Having a neural network consisting of convolutional layers only, a so-called fully-convolutional network, allows an application to input data of arbitrary size even after training [38].

There also exist various approaches for music synthesis using neural networks with some architectural detail of the above: most remarkably, PerformanceNet [72] is a

deep convolutional network performing exactly the task approached in this thesis, namely score-to-audio synthesis. For this, it represents the scores of one track as a binary pitch-over-time matrix, called a pianoroll [14]. Exploiting the spatial information included in this, the network is trained to map the pianoroll to a spectrogram which is transformed to waveform audio using the algorithm of Griffin and Lim [21]. The proposed network architecture is rather expensive in terms of parameters: after applying a deep convolutional autoencoder, termed ContourNet, which is intended to learn the macro structure of the spectrogram, another sub-network called TextureNet is applied for fine-tuning. The latter splits up the data trying to fine-tune each frequency band separately. This way, a lot of "external information" is already implicitly included in the training process by design. For further improvements, PerformanceNet even includes an additional on-/offset encoder which is fed on- and offset times of notes in addition to the pianoroll. PerformanceNet is actually not one model, but instead an independent instance with a hand-crafted pre-processing strategy different for every instrument on which it is trained. This not only makes training rather cumbersome, in particular there is also no capability of synthesizing multiple instruments at once, learning interdependencies and generalizing to new instruments. This model can synthesize audio with a sampling frequency of 44.1 kHz. As it is fully-convolutional, it can accept scores of arbitrary length.

Meanwhile, GANSynth [17] as the name suggests uses a generative adversarial network for this task: from a random seed and the pitch as side condition, the model is trained to produce a waveform via the spectrogram of a single note of the NSynth-Dataset. Therefore, this single model can synthesize a broad variety of instruments but only sequentially. It can be considered a major drawback that the single notes have to be merged externally in order to synthesize a whole song. However, as the timbre is determined by the seed which is drawn from a continuous space, interpolations in timbre and the generation of unreal sounds are possible.

Previous applications of GANs for audio synthesis largely focused on speech synthesis but not music. Instead, most AI music synthesis has been performed by auto-regressive models such as WaveNet [52]: this probabilistic model requires some waveform audio input based on which it derives a continuation of this sequence. By design, the model can be conditioned on the instrument but it cannot synthesize a specific score by heart, yet there are modifications for this [42]. In a subjective try-out by the author of this thesis, the music produced by WaveNet sounds largely coherent, yet noisy. The model synthesizes audio with 16 kHz. A follow-up approach with a GAN has been made with WaveGAN [12]: in an unsupervised way, it learns to synthesize audio both, via a spectrogram and directly as waveform. This DCGAN [59] based model which also produces 16kHz audio has mainly been trained on piano and drum music and puts a focus on sound effects for film and music industry. The model produces one second chunks of audio which corresponds to a comparably small output of size 128×128 in the spectrogram variant.

This master's thesis will cope with much larger spectrograms in both, time and frequency dimension. As compared to this a pianoroll representation is rather small, *orGAN* implicitly has to solve a super-resolution problem. This has been done successfully for images by the pix2pix GAN [27]: this model performs conditional image synthesis using a GAN with a U-Net autoencoder as generator and a discriminator which outputs independent real/fake classifications for equally sized patches of its input data. The highly successful model not only experiments with different sizes of these patches but also with a compound loss function including the L1 distance between real data and fakes besides the regular adversarial loss. This is intended to encourage learning of local details as well as the global macro-structure of the target data. Considering spectrograms as a special type of images, in *orGAN* the architecture of this model is merged with the approach of PerformanceNet in order to develop one single GAN model capable of multi-instrument score-to-music synthesis for high frequency audio.

Chapter 3

Dataset, Pre- and Post-Processing

3.1 The URMP-Dataset and its Preparation

The goal of this work is to develop a model that can translate scores to audio for arbitrary combinations of multiple instruments with a focus on strings and winds starting from a baseline created from single-instrument samples. This requires a dataset that contains

(i) frame level time-aligned note annotations for every waveform audio recording

(ii) songs played by multiple natural[1] instruments

(iii) songs covering a broad range of genre, tempi and styles/composers

(iv) aligned single recordings of all instruments playing the same song, from which arbitrary combinations can be formed.

The most restricting factor here are the time-consuming annotations. Those are necessary as aligning MIDI-formatted scores to audio recordings without a noticeable shift over time is a challenging problem for music information retrieval itself (see for instance [60]). One of the few freely available datasets of high quality fulfilling the criteria above is the *University of Rochester Multi-Modal Music Performance (URMP) Dataset* [34]: it includes data for 44 songs ranging from classical to modern content played with different ensembles from duets to quintets. It contains a total amount of 1.3h of mult-instrument records split into 149 single records with a total duration of approximately 4.5h. Details are provided in Table 3.1.[2]

[1] i.e. not from synthesizing devices such as an electronic piano and therefore no instruments with a MIDI-interface (which would enable a simple transcription of the played music).

[2] As the multi-instrument samples are by definition not disjoint, the total number of samples is less than the sum of the number of occurrences of all instruments.

			Occurences		
Instrument	**#Records**	**Total Duration**	**Single**	**Multi**	**Total**
Violin	34	01:00:51	3331	25324	28655
Viola	13	00:26:00	1413	15079	16492
Cello	10	00:18:40	1211	8877	10088
Double Bass	4	00:09:48	337	4103	4440
Flute	18	00:32:55	1782	9805	11587
Oboe	6	00:11:43	591	4401	4991
Clarinet	10	00:18:06	945	4895	5849
Saxophone	11	00:15:47	828	4847	5675
Bassoon	3	00:04:15	207	2060	2267
Trumpet	21	00:40:11	2230	14394	16624
Horn	5	00:10:12	532	6178	6710
Trombone	9	00:18:07	928	6834	7762
Tuba	5	00:08:56	484	4892	5376
Total	149	04:35:31			
# Samples			14819	41820	56639

Table 3.1: Statistics of the pre-processed URMP-Dataset including the number of samples from a 5s sliding window on trimmed single- and multi-instrument recordings. Note that the total number of multi-instrument samples is not the sum of the occurences of all instruments as the sets of samples per instrument are obviously not disjoint.

To obtain a large amount of data for learning multi-instrument play, during pre-processing ensembles are formed: for every song, all possible ensembles are given as the power set[3] of instruments this song has been recorded for. For a song with instruments[4] $\{1, ..., n\}$, this results in $|\mathscr{P}(\{1, ..., n\})| \setminus \{\emptyset\} = 2^n - 1$ combinations, so that the whole dataset is augmented to 544 combinations of instruments (including singles).

Despite the final model will be able to cope with inputs and outputs of arbitrary length (see Section 4.6 and Section 4.7), the model is built from equally sized chunks of scores and audio.[5] Like the most related work PerformanceNet [72], this model uses a chunk size of 5 seconds. To maximize the amount of training data, those chunks are created by a window sliding over the scores represented as a so-called Pianoroll (see Section 3.4) and over the waveform. This window simply drops the small overflow that might occur and uses a stride of 1 as a simple method of data augmentation. This is also done in PerformanceNet, but with providing additional beforehand support to the model by tailoring the stride to each type of instrument. As stated in [72, Sec. "Dataset"], the small stride leads to a high similarity of the samples, but nevertheless "it still helps the model learn better, especially when we use a small hop size in the STFT [Short-Time Fourier Transform, see Section 3.2] to compute the spectrograms". Using this stride results in a total amount of more than 14,000 samples of single recordings and more than another 41,000 samples of multi-instrument play.

The whole pre-processing of the URMP-Dataset is summarized in the following diving into detail of its single steps after starting from the overall process outlined in loose pseudocode[6] in Algorithm 1. While its subroutines `getInvolvedInstruments`, `indicesToInstruments` and `trim` are self-descriptive, others require further specification:

`formEnsembles` This routine takes the number n of available instruments for a specific song as integer input and returns the power set $\mathscr{P}(\{1, ..., n\}) \setminus \{\emptyset\}$ of the set of indices of the given instruments. This equals the set of all i-tuples of those indices with $i \in \{1, ..., n\}$ where only one permutation of each i-tuple is included (e.g. $(1, 2, 3)$, but not $(2, 1, 3)$, $(2, 3, 1)$ and so on).

`getPianoroll` This function loads the annotations for each instrument in an ensemble and forms a representation called a pianoroll out of them as described

[3] excluding, of course, the empty set

[4] There might be multiple instruments of the same type performing the same song but using different scores.

[5] This enables training with minibatches as described later in Section 4.2.2

[6] Please note that the actual implementation coming with this work might slightly differ from the described algorithm in structure and naming for programming language specific reasons, but follows the same logic.

Algorithm 1: Pre-Processing

```
Input: URMP-Dataset
Initialize preprocessedData ← [ ];
foreach song in URMP-Dataset do
    instruments ← getInvolvedInstruments(song);
    n ← length(instruments);
    ensembles ← formEnsembles(n);
    foreach e in ensembles do
        ensembleInstruments ← indicesToInstruments(e);
        wave ← sumPointwise(getSingleWaves(ensembleInstruments, song));
        (pianoroll, musicBounds) ← getPianoroll(ensembleInstruments,
         song);
        wave ← trim(wave, musicBounds);
        chunks ← chunkAndMakeSpecgrams(wave, pianoroll);
        preprocessedData ← preprocessedData + chunks;
    end
end
Output: A set of paired pianoroll and spectrogram chunks
```

in detail below in Section 3.4. Further, it determines the boundaries of playing music over all ensemble instruments as a tuple of frame indices which is used to trim leading and trailing silence in the pianoroll and later in the waveform audio as well. This routine is described further in the sequel, namely in Algorithm 3.

`getSingleWaves` returns the list of audio waveforms of the single recordings of all instruments in a given ensemble as time series of equal length. Those can be merges by simple point-wise addition.

`chunkAndMakeSpecgrams` The model is intended not to generate a waveform directly, but instead a spectrogram which is retrieved from the waveform signal using the Fourier Transform described later in Section 3.2. The creation of this spectrogram as well the chunking of both the pianoroll and the waveform into 5s samples are summarized in the algorithm above as the function `chunkAndMakeSpecgrams` which returns a list of tuples of pianoroll chunks and the spectrogram of the corresponding audio sequence.

Slightly anticipating terms introduced later in Section 4.1, it shall be mentioned here that during this pre-processing, a randomly drawn subset containing 10% of all samples is split off and stored separately as a so-called *Test Set* which will be used later for assessing the model's performance.

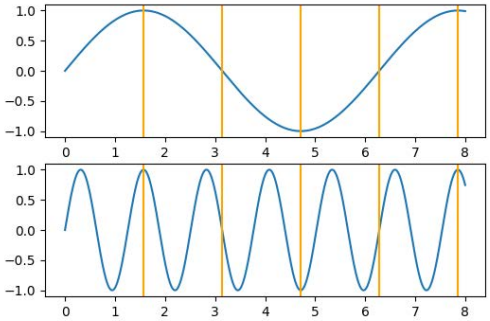


Figure 3.1: Using a low sample rate on signals with different frequency. Positions where samples are drawn are marked with orange bars. While the low sampling rate collects most of the low-frequency signal (top) sufficiently well, it largely fails to capture the details of the high-frequency signal (bottom).

3.2 Audio Processing with the Short-Time Fourier Transform

The task is of this work is to synthesize music and thus the model is trained using waveform audio signals of real instrument recordings.

Definition 1 (Audio Signal and Sampling)
An *Audio Signal* for this work is modeled as a continuous function $f : \mathbb{R}_{\geq 0} \to [-1, 1]$ in time which is discretized in the recording process using a *Sampling Operator*

$$S_h : L(\mathbb{R}) \to \ell(\mathbb{Z}) \quad \text{with} \quad S_h f(k) := f(hk),\ k \in \mathbb{Z}$$

as described in [63, Ch. 2].[7] The stride $h \in \mathbb{R}$ determines the time span (in seconds) between the drawing of two samples (also called *Frame*s) and thus determines the *Sample Rate* $s := \frac{1}{h}$ in frames per second. □

Obivously, a higher sample rate results in a more precise representation of the original signal and captures especially high frequency signals, i.e. periodic signals with a short period, more accurately as illustrated in Figure 3.1. The audio of the URMP-Dataset has been recorded using high-quality sampling with a sample rate of 48kHz, i.e. $s = 48,000$ frames per second.

Before going into detail on the processing of the URMP-Dataset audio, the mathematical concept of a signal is related to music following [71, Ch. 1]: Usually, audio

[7] $L(\mathbb{R})$ is the space of all real functions and $\ell(\mathbb{Z})$ the one of all real sequences.

recordings are a mixture of multiple sinusoidal waves, named *Pure Tone Waveforms* of different frequencies and loudness. In particular for music recordings, each note played by an instrument is made up of multiple frequencies, i.e. pure tone waveforms, summarized as *Harmonics*. Those consists of the so-called *Fundamental Frequency* of the signal, which directly corresponds to the pitch of the note and of positive integer multiples of this fundamental frequency, termed the *Overtones*. Unlike a "synthetic signal", fundamental frequency and harmonics are not constant within the playing of one note in the recording of a natural instrument, but slightly vary over time following an instrument-specific pattern, the *Timbre*. The third characteristic of a sound beside timbre and pitch, the loudness, is directly determined by the amplitude of the waveform.

Directly using waveform audio to train a machine learning model requires the model to "extract" all of those properties "hidden" in the waveform before it can synthesize music. To make the model's task more feasible, an audio representation is used which directly reveals most of the above information of a piece of audio:

Using the *Fourier Transform*, a signal can be projected from time into frequency domain (see Figure Figure 3.2), i.e. transformed into a function that decomposes the given signal by revealing the magnitude for each of the frequencies it incorporates [63]:

Definition 2 (Time-Discrete Fourier Transform)
Given a discretized signal as sequence $(c_t) \in \ell(\mathbb{Z})$, its *Time-Discrete Fourier Transform*[8] is

$$\widehat{c} : \mathbb{R}_{\geq 0} \to \mathbb{C}, \quad \xi \mapsto \sum_{t \in \mathbb{N}_0} c_t e^{-it\xi}$$

providing information on the presence of a pure tone waveform with frequency ξ in the signals. □

The Fourier transform returns a complex number whose absolute value $|\widehat{c}(\xi)|$ is the magnitude of a pure tone wave with frequency ξ in signal c while its imaginary part gives the *Phase* angle $\measuredangle \widehat{c}(\xi)$, i.e. the temporal misalignment between this pure tone waveform and the given signal [45].

Remark 3 Note that the Fourier Transform of a discrete signal is 2π-periodic as

$$\widehat{c}(\xi + 2\pi) = \sum_{t \in \mathbb{N}_0} c_t e^{-it(\xi + 2\pi)} = \sum_{t \in \mathbb{N}_0} c_t e^{-it\xi} \underbrace{(e^{i\pi})^{2t}}_{=-1} = \widehat{c}(\xi).$$

[8] In the sequel, it will be referred simply as *Fourier Transform*.

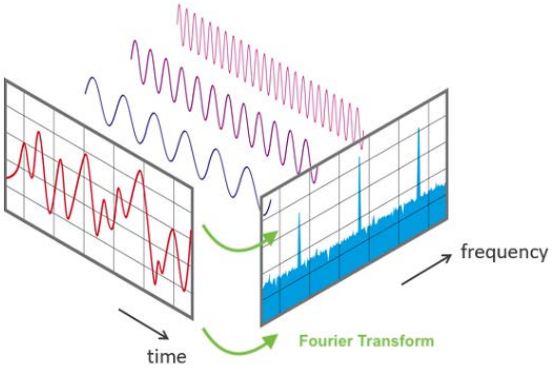


Figure 3.2: The intention of the Fourier Transform: moving from time-domain and the signal mapping time to a real value to the frequency domain and a function mapping a frequency to its magnitude in the given signal. This way, a signal can be decomposed into sinusoidal pure tone waves of different frequencies.
Graphic taken and modified from: https://upload.wikimedia.org/wikipedia/commons/6/61/FFT-Time-Frequency-View.png

As pointed out in [45], above formulation bears the computational problems of dealing with long and possibly infinite sums and a continuous frequency domain, which cannot be handled in implementation practice. The first is tackled by computing the Fourier Transform only over a fixed number $T \in \mathbb{N}$ of samples of the input signal (see [45, Sec. 2.1.3] for details). To address the second problem, the frequency range, which can be limited to $[0, 2\pi]$ due to Remark 3, is sampled as well with a sample rate of $\frac{1}{N} \in \mathbb{Q}$ [63]. A common choice is $N = 2048$. It is recommended to couple $N = T$ for invertibility and computational efficiency [45, Sec. 2.3.1].

Definition 4 (Discrete Fourier Transform)
For a time-discrete signal as sequence $(c_t) \in \ell(\mathbb{Z})$, its *Discrete Fourier Transform (DFT)* of order N, i.e. with a frequency sampling rate of $\frac{1}{N}$, is defined as

$$(c_t)_{t\in\{0,\dots,T\}} \mapsto (\widehat{c}_n)_{n\in\{0,\dots,N-1\}} \quad \text{with} \quad \widehat{c}_n := \widehat{c}(n2\pi/N) = \sum_{t=0}^{T-1} c_t e^{-itn\frac{2\pi}{N}}$$

computed over T samples representing the whole signal (c_t). □

Remark 5 A DFT of order N is N-periodic as

$$\widehat{c}_{n+N} = \widehat{c}((n+N)2\pi/N) = \widehat{c}(n2\pi N + 2\pi) \overset{\text{Remark 3}}{=} \widehat{c}_n$$

and thus can be represented by $(\widehat{c}_n)_{n\in\{0,\dots,N\}}$. Further, the DFT is symmetric in the

complex conjugate on this index set as

$$\widehat{c}_{N-n} := \sum_{t=0}^{T-1} c_t \underbrace{e^{-it2\pi}}_{=1} e^{itn\frac{2\pi}{N}} = \sum_{t=0}^{T-1} c_t \overline{e^{-itn\frac{2\pi}{N}}} = \overline{\widehat{c}_n}$$

Hence, an order-N DFT effectively is $(\widehat{c}_n)_{n\in\{0,\dots,\lfloor\frac{N}{2}\rfloor\}}$, i.e. it can be efficiently represented by $\lfloor\frac{N}{2}\rfloor + 1$ frequency "bins".[9]

To retrieve not only the frequency information averaged over the whole signal, but instead over specific points in time, a sliding window with a certain size $W \in \mathbb{N}$ and *Hop Size* $h \in \mathbb{N}$, i.e. the stride of the window, can be used to compute the DFT multiple times. The window is represented by a function $w : \mathbb{Z} \to \mathbb{R}$, which is non-zero only for a small interval $\{0, ..., W-1\}$, so that it can be convolved with the original signal to single out chunks of W frames [45]. One may note, that the larger the window, the higher is the "sampling rate in the frequency domain" and obviously the larger is the number of frames on which frequency information is averaged over. So achieving prefect resolution in both, frequency and time domain simultaneously is not possible. The above in a formal definition:

Definition 6 (Short-Time Fourier Transform)
For a time-discrete signal as sequence $(c_t) \in \ell(\mathbb{Z})$, its *Short-Time Fourier Transform (STFT)* of order N, which uses a sliding window $w : \mathbb{Z} \to \mathbb{R}$ of size W with stride h is

$$(c_t)_{t\in\mathbb{N}_0} \mapsto (\widehat{c}_{m,n})_{(m,n)\in(\mathbb{N}_0\times\{0,\dots,N-1\})} \quad \text{with} \quad \widehat{c}_{m,n} := \sum_{t\in\mathbb{N}_0} c_t \; w(t-mh) \; e^{-itn\frac{2\pi}{N}}.$$

where m indicates the sliding window's position and n the frequency band. □

Note that this is just an iterative application of the DFT for different window positions. In practice, computations are speed up highly using an efficient implementation of the DFT, the *Fast Fourier Transform (FFT)*, instead if N is a power of 2, while again coupling $W = N$ for invertibility. The interested reader is referred to [45, Sec. 2.4.3]. This work uses the STFT implementation of the `librosa` library (see [43]) along with its default window function which is also used for instance in [72], the bell-shaped symmetric *Hann-Window* defined as

$$w(k) := \begin{cases} \sin^2(\pi k/(W-1)), & k \in \{0, ..., W-1\} \\ 0, & \text{else} \end{cases}$$

[9] This is especially the case as further processing only deals with spectrograms (introduced later in this section) which require only the magnitude, i.e. the real part, of the DFT's output.

The information extracted using the STFT can be arranged in a matrix which is useful to summarize and visualize frequency information over time:

Definition 7 (Spectrogram)
Let $(c_t) \in \ell(\mathbb{Z})$ be a discretized signal and $(\widehat{c}_{m,n})_{(m,n)\in(\mathbb{N}_0\times\{0,\dots,N-1\})}$ its order-N STFT in a time range of M windows as specified above. Following [45, Sec. 2.1.4], the *Spectrogram* is a matrix $S \in \mathbb{R}^{(\lfloor \frac{N}{2} \rfloor+1)\times M}$ where

$$S_{n,m} := |\widehat{c}_{m,n}|^2 = (\text{Re } \widehat{c}_{m,n})^2 + (\text{Im } \widehat{c}_{m,n})^2$$ □

Note that taking the absolute value provides the magnitude while discarding phase information. Applying the shifted natural logarithm

$$\widetilde{S}_{n,m} := \ln(|\widehat{c}_{m,n}|^2 + 1) \tag{3.1}$$

on top of the above computations for normalization as well as the emphasis of "musical or tonal relationships" [45] and using a logarithmic scale on the y-axis results in a *Log-Magnitude Spectrogram* as used for instance in [72, 17]. Above, the logarithm incorporates a left-shift of 1 such that it only outputs non-negative values and does not tend to infinity for the given non-negative inputs. Taking the logarithm of the magnitude not only corresponds nicely to the human perception of pitch, but also compresses the large range of magnitude values in favor of small nuances which can be crucial for human audio perception [45, Sec. 3.1.2.1].
Encoding the magnitude as color, S can be visualized as in Figure 3.3: for visualization, magnitudes are converted to decibels (which are aligned to a logarithmic scale) through

$$\text{dB}(S) := 10 \ \log_{10}(S/r)$$

as described in [45, Sec. 1.3.3] where r is the reference for relatively mapping mapping all magnitude values to the logarithmic decibel scale[10]. This compression "enhance[s] small sound components that may still be perceptually relevant" and "noise-like transients" at notes' onsets [45, Sec. 2.1.4].

Overall, when it comes to choosing the parameters of spectrogram generation described above, this work follows [72], which uses the STFT not only with $N = 2048 = W$ frequency bins and frames per window but also identifies a small hop size of $h = 256$ as crucial for the quality of the generated audio. Also this work adopts the practice of slightly re-sampling audio in a way such that the number of frames per second is divisible by h: the original audio is re-sampled with

$$s_{\text{new}} := \text{wps} * h \qquad \text{with} \quad \text{wps} := \left\lfloor \frac{s_{\text{original}}}{h} \right\rfloor$$

[10] In the `librosa` [43] implementation, it is $r = 1.0$ by default.

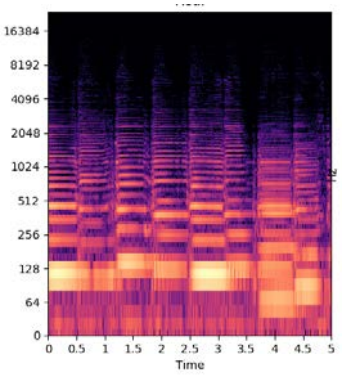


Figure 3.3: A spectrogram of a chunk of music with frequencies (in Hertz) on the y-axis, window steps at the x-axis (with time labels) and loudness in decibel encoded as color. It clearly shows the fundamental frequency ("most intense lower bar of each stack") of each note together with its overtones.

where wps is the number of STFT-windows per second. For $s_{\text{original}} = 48\text{kHz}$ this results in wps $= 187$ and $s_{\text{new}} = 47872\text{Hz}$. Taking into account Remark 5 and a time span of 5s, the resulting spectrogram with $M = 5 * \text{wps}$ has a size of

$$\left(\lfloor \frac{N}{2} \rfloor + 1\right) \times (\text{wps} * 5), \quad \text{here} \quad 1025 \times 935.$$

3.3 Back-Conversion of Spectrograms to Waves

The machine learning model developed in this worked will learn to output spectrograms. To make use of them, a procedure for a conversion back to waveform audio is required. Inverting equation 3.1, the magnitude output of the STFT can be easily reconstructed from a given log-magnitude spectrogram $S \in \mathbb{R}^{(\lfloor \frac{N}{2} \rfloor + 1) \times M}$ by

$$|\widehat{c}_{m,n}| = \sqrt{\left| exp(\widetilde{S}_{n,m}) - 1 \right|}$$

where the operations are applied element-wise. For inverting the STFT, first consider the inverse of its incorporated DFT:

Theorem 8 (Inverse DFT)
For an order-N DFT $\widehat{C} := (\widehat{c}_n)_{n \in \{0,\dots,N-1\}}$ which was computed over N samples[11] *$C := (c_t)_{t \in \{0,\dots,N-1\}}$ of a time-discrete signal $(c_t) \in \ell(\mathbb{Z})$, its* Inverse Discrete Fourier

[11] The matching $T = N$ of the order of the DFT and the number of underlying samples is required, as per DFT coefficient exactly one signal frame can be reconstructed.

Transform (IDFT) *is*

$$IDFT(\widehat{C}) := \left(\frac{1}{N} \sum_{n=0}^{N-1} \widehat{c}_n e^{itn\frac{2\pi}{N}} \right)_{t\in\{0,\dots,N-1\}} = C. \qquad \square$$

PROOF As outlined in [63], it holds

$$\begin{aligned}
\mathrm{IDFT}(\widehat{C})_t &= \frac{1}{N} \sum_{n=0}^{N-1} \widehat{c}_n e^{itn\frac{2\pi}{N}} \\
&= \frac{1}{N} \sum_{n=0}^{N-1} \left(\sum_{k=0}^{N-1} c_k e^{-ikn\frac{2\pi}{N}} \right) e^{itn\frac{2\pi}{N}} \\
&= \frac{1}{N} \sum_{k=0}^{N-1} c_k \sum_{n=0}^{N-1} e^{in\frac{2\pi}{N}(t-k)} \\
&\overset{(*)}{=} \frac{1}{N} \sum_{k=0}^{N-1} c_k N I_{\{0\}}(t-k) = c_t.
\end{aligned}$$

Short-handing $\omega := e^{2i\pi/N}$, the argument $(*)$ above is that

$$\begin{aligned}
\sum_{n=0}^{N-1} \left(e^{i\frac{2\pi}{N}(t-k)} \right)^n &= \sum_{n=0}^{N-1} (\omega^{(t-k)})^n \\
&= \begin{cases} \sum_{n=0}^{N-1} 1 = N, & \text{for } (t-k) \in \{0, N\} \\ \frac{1-(\omega^{t-k})^N}{1-\omega^{(t-k)}} = \frac{1-(\overbrace{\omega^N}^{=1})^{t-k}}{1-\omega^{(t-k)}} = 0, & \text{else} \end{cases}
\end{aligned}$$

where the latter exploits that $t - k < N$ and thus $w^{(t-k)} < w^N = 1$ and therefore the partial geometric series can be rewritten as the fraction above. ∎

To reconstruct a signal from an STFT, also the sliding window process has to be inverted as described in [66]:

Theorem 9 (Inverse STFT)
For an order-N Short-Time Fourier Transform (STFT)

$$\widehat{C} := (\widehat{c}_{m,n})_{m\in\{0,\dots,M-1\}, n\in\{0,\dots,N-1\}}$$

of T points $C := (c_t)_{t\in\{0,\dots,T\}}$ *of a time-discrete signal* $(c_t) \in \ell(\mathbb{Z})$ *using a sliding window w of length W with stride h so that it fulfills the condition for perfect reconstruction, namely*

$$0 \neq \sum_{m=0}^{M-1} (w(t-mh))^2 = const =: D \quad \forall t \in \{0, ..., T\}.$$

its Inverse Short-Time Fourier Transform (ISTFT) *is*

$$ISTFT(\widehat{C}) := \left(\frac{1}{D} \sum_{m=0}^{M-1} IDFT(\{\widehat{c}_{m,1}, ..., \widehat{c}_{m,n}\})_{t-mh} \; w(t-mh) \right)_{t \in \{0,...,T\}} = C. \qquad \square$$

PROOF As written in [66], it holds

$$\begin{aligned} \mathrm{ISTFT}(\widehat{C})_t &= \frac{1}{D} \sum_{m=0}^{M-1} \mathrm{IDFT}(\{\widehat{c}_{m,1}, ..., \widehat{c}_{m,N}\})_{t-mh} \; w(t-mh) \\ &= \frac{1}{D} \sum_{m=0}^{M-1} c_t \; w(t-mh) \; w(t-mh) \\ &= \frac{1}{D} c_t \underbrace{\sum_{m=0}^{M-1} (w(t-mh))^2}_{=D} = c_t. \end{aligned} \qquad \blacksquare$$

For an illustration of the STFT and its inversion process, refer to Figure 3.4.

It can be seen that the ISTFT cannot be directly applied having only the magnitude $|\widehat{c}_{m,n}|$ without the phase information $\measuredangle \widehat{c}_{m,n}$. This necessitates more sophisticated strategies for reconstruction such as the *Griffin-Lim Algorithm (GLA)*, which tries to iteratively estimate a signal whose STFT magnitude is similar to the given one [21]. In each iteration, a new estimation is computed by applying the STFT to the current estimation, substituting the magnitude of the output with the target magnitude while keeping the phase angle and then applying the ISTFT:

Algorithm 2: The Griffin-Lim-Algorithm

Shorthand $I := \{0, ..., M\} \times \{0, ..., N\}$;
Input: STFT Magnitudes $S := (|\widehat{c}_{m,n}|)_{(m,n)\in I}$
Input: Target singal length T and STFT parameters
Input: Number of iterations X
Initialize signal $C := (c_t)_{t\in T}$ arbitrarily;
for $x \leftarrow 1$ **to** X **by** 1 **do**
  $\widehat{C} \leftarrow \mathrm{STFT}(C)$;
  $\widetilde{C} \leftarrow \left(S_{m,n} e^{i \measuredangle \widehat{C}_{m,n}} \right)_{(m,n)\in I}$;
  $C \leftarrow \mathrm{ISTFT}(\widetilde{C})$;
end
Output: Estimated signal C

Again, this work uses `librosa`'s implementation [43] of the Griffin-Lim algorithm. This is an implementation of a variant of the GLA, the *Fast Griffin-Lim Algorithm*

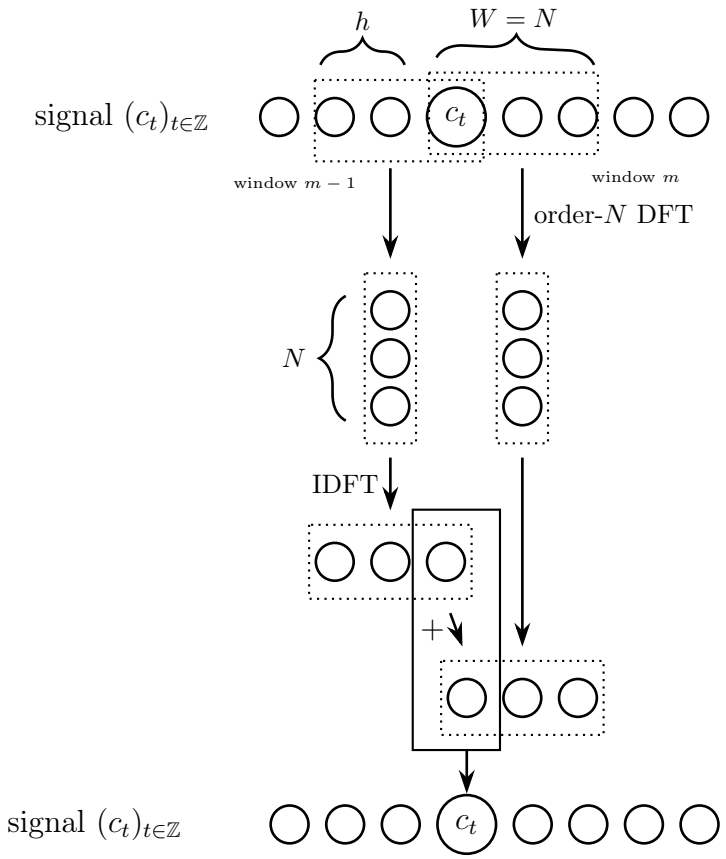


Figure 3.4: The STFT as a windowed DFT with its reconstruction through the IDFT and overlapping addition of the windowed outputs.

[56], which in a sense uses a momentum term to update the estimated signal in each iteration with a varying intensity depending on the former estimation. As the authors state, this not only speeds up computation but also causes an increase in the quality of the output. Despite those gains, the (Fast) GLA is still comparatively slow an will turn out to be the major bottleneck of computation when using this work "in production" with long sequences of music.

3.4 Pianoroll Generation

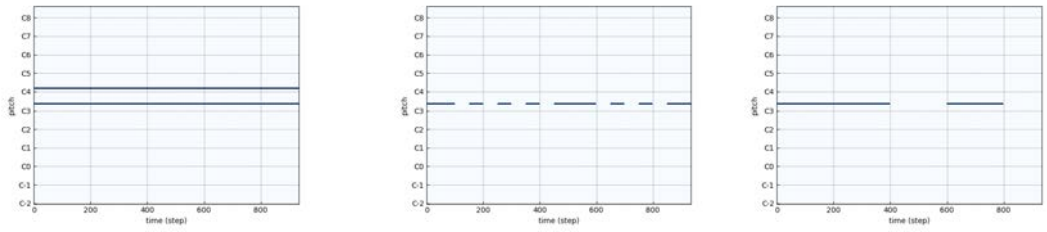
Following [72], the scores given as frame level annotations which will serve as an input to the audio generation model are represented as so-called *Pianoroll*s as proposed by [13] and also mentioned in [45, Sec. 1.2.1]: a pianoroll is a binary matrix $P \in \{0,1\}^{128\times T}$ representing a 2D-plot of the scores with time and pitch as axes. Every entry $P_{p,t}$ is 1 if and only if the note $p \in \{0, ..., 127\}$ (This is the range of the MIDI standard.) is played at timestep $t \in \{0, ..., T\}$.

The scores of multiple instruments playing the same song can be represented as a set of multiple pianorolls [13]. The approach taken in this work is to stack those pianorolls to an image-like rank 3 tensor where the position in the stack indicates the type of instrument. This makes the instrument type an intrinsic property of the model's input and eliminates the need of training separate models like in [72] or finding ways to incorporate contextual data. Anticipating Section 4.6, the underlying motivation is that this spatial encoding of the instrument type facilitates the learning of instrument specific filters by the convolutional model described later. The scores of different tracks are arranged so that those of related instruments, i.e. strings, brass and woodwinds are closely together within this stack so that the model can exploit this spatial information and is likely to apply similar transforms to instruments with similar timbre.

Unfortunately, this representation does not come without drawbacks in the form of ambiguities:

Instrument Ambiguity First, multiple instruments of the same type playing the same notes are not distinguishable from one instrument playing those notes. Accordingly, multiple instruments of the same type playing different single notes are not distinguishable from a single-instrument playing multiple notes at once.

Note Ambiguity Also, multiple notes without a pause or offset in between are not distinguishable from one note: for instance two eighth notes of the same pitch do not differ from a quarter note of the same pitch when the pianoroll is retrieved from MIDI scores. This is an issue in particular, as the model is trained with pianorolls made from frame-level annotations of real performances



(a) Instrument Ambiguity (b) Note Ambiguity (c) On-/Offset Ambiguity

Figure 3.5: Ambiguities of the Pianoroll Representation

where this effect does not occur as musicians have to do a slight on-/offset between two notes unless intended otherwise. Hence, a perfectly trained model learns to map "a continuous bar" in the pianoroll *always* to one single note while in production the same might represent multiple notes that are expected to be played with on-/offsets in between.

On-/Offset Ambiguity Further, smooth offsets (onsets) of a short note with a slow fade-out (fade-in) like at the end (beginning) of a song do not differ from the hard offset (onset) of a long note. Again, the usage of frame-level annotations versus MIDI-data opens a gap as in training, notes that should be faded out (in) slowly are represented by an accordingly long "bar" in the pianoroll while in production the model is likely to be expected to map "a much shorter bar" to the same output.

As another drawback, pianorolls are a actually quite wasteful data representation in terms of computational space as they are very sparse. Nevertheless, their huge advantage counterbalancing the mentioned issues is, that they make "the use of CNNs [Convolutional Neural Networks, see Section 4.6] feasible"[13, Sec. "Data Representation"]. As pointed out by [72], they nicely correspond to the spectrograms of their accompanying recording as it can be seen from Figure 3.6. This way, the problem of score-to-audio translation becomes a super-resolution[12] image-to-image translation task [72, Sec. "Methodology"] where a contour plot needs to be converted into a natural-looking image. The state of the art for the latter is set by the pix2pix model [27] which becomes applicable here through the usage of pianorolls.

It is to be mentioned that for visualizing such a pianoroll or retrieving it from MIDI data, this work relies on the package `pypianoroll` [14] provided by the authors of MuseGAN [13]. The creation from the CSV-formatted annotations in the URMP-Dataset is done from scratch using the procedure defined in Algorithm 3.

[12] as a pianoroll is much smaller than its corresponding spectrogram

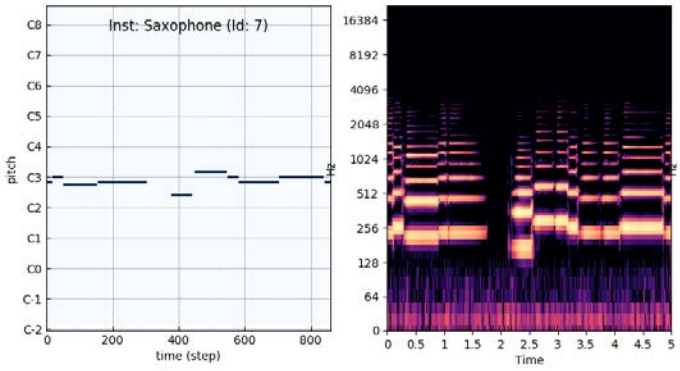


Figure 3.6: The pianoroll representation of musical scores highly corresponds in structure to the spectrogram of the associated performance recording.

Algorithm 3: getPianoroll

Input: A list of instruments and a song
Require: const $d \leftarrow$ instrumentsInDataset;
Require: const $s \leftarrow$ samplingRate;
Let $l \leftarrow$ lengthInTimesteps(song);
Initialize pianoroll $\leftarrow$ zeros(shape=(l, 128, d));
Initialize musicBounds $\leftarrow$ (`nil`, `nil`);
foreach *i in instruments* **do**
  foreach *(onsetSecond, duration, pitch) in annotations(song)* **do**
    $t_{\text{on}} \leftarrow$ onsetSecond $*s$;
    $t_{\text{off}} \leftarrow$ (onsetSecond + duration)$*s$;
    $p_{\text{old}} \leftarrow$ pianoroll[t_{on}:t_{off}, pitch, i];
    pianoroll[t_{on}:t_{off}, pitch, i] $\leftarrow$ maxPointwise(p_{old}, 1);
  end
  $t_{\text{first}} \leftarrow$ firstOnsetSecond(song) $*s$;
  $t_{\text{last}} \leftarrow$ (lastOnsetSecond(song) + lastNoteDuration(song))$*s$;
  musicBounds = (min(musicBounds[0], t_{first}), max(musicBounds[1], t_{last}))
end
Output: A pianoroll and musicBounds, i.e. a tuple describing boundary time indices of the playing music

The sampling rate used in this in algorithm is expected to be the same as in Section 3.2 so that the pianoroll representation matches the associated spectrogram's timescale made up of 935 bins for a 5s audio signal. Here, this results in a pianoroll shape of 128×935 which is smaller than the desired spectrogram output of the model by far.

Chapter 4

Method

4.1 Machine Learning Tasks

For many computer science tasks of high common interest in fields such as computer vision, natural language processing or music information retrieval a specialized pre-determined[1] algorithm cannot be easily found. Instead, such tasks can be approached with self-optimizing[2] algorithms, so-called *Machine Learning (ML)* [19, Sec. 5.1.1].

Definition 10 (Supervised Learning Task)
Given sets[3] $X \subseteq \mathbb{R}^{d_x}$ and $Y \subseteq \mathbb{R}^{d_y}$, a *Supervised Learning Task* is to approximate an unknown mapping $f^* : X \to Y$. This is done using a finite *Dataset* $\mathbb{D} := \{(x, f^*(x)\} \subset X \times Y$ whose elements consist of *Samples* x and their corresponding *Labels* or *Ground Truth* $f^*(x)$. The function f^* is called the *Learning Goal*. □

The term *supervised* indicates, that for the learning process the ground truth is provided, e.g. by a human expert. In contrast, *Unsupervised Learning* would try to find a structure (e.g. a partitioning) on X using only $\{x|(x, \cdot) \in \mathbb{D}\}$ without labels [19, Sec. 5.1.1]. As this has no relevance for the task approached in this thesis, all following considerations are restricted to Supervised Learning.

As it can be seen from the definition above, a *Machine Learning Algorithm* is required to be a highly flexible function approximator which can find $f : X \times \mathbb{W} \to Y$

[1] *Pre-determined* in the sense that all operations and input-independent parameters are fixed at design-time.

[2] *Self-optimizing* in the sense that the algorithm can adjust its parameters itself with respect to a given objective.

[3] As in [64, Def 1.1], the term *set* is used intentionally to keep the definition broad. Further, real-valued data of arbitrary dimensionality can be re-shaped to a vector, which is why the definition uses $X \subseteq \mathbb{R}^{d_x}, Y \subseteq \mathbb{R}^{d_y}$ just as in [19, Pt. 1, Sec 5.1].

using $\mathbb{D}$, such that it *generalizes* in a way that optimally $f(x, W) \approx f^*(x)$ for *all* $x \in X$. To perform this approximation, f is based on *Trainable Parameters* $W \in \mathbb{W} \subseteq \mathbb{R}^{d_W}$, expressed as $f_W := f(\cdot, W)$, which are adjusted during a so-called *Training* of the algorithm towards an optimum of a suitable, task-specific *Loss Function*[4] quantifying the approximation quality [19, Secs. 5.1.2, 5.2]. This is essentially the *Learning Process* of the model.

To determine the generalization ability of f after training, the dataset is partitioned $\mathbb{D} := \mathbb{T} \dot\cup \mathbb{U}$ in a *Training Set* $\mathbb{T}$ and a *Test Set* $\mathbb{U}$, where the latter is held out of training, i.e. not used for the approximation process[5] [19, Sec. 5.1.2]. During training, $\mathbb{T}$ is fed to f multiple times while in between the parameters W are adjusted to minimize the loss of f on the training samples. One complete processing of the whole set $\mathbb{T}$ is called an *Epoch* of the training. After training is completed, i.e. an (locally) optimal configuration of parameters of f with respect to the loss function has been found, the performance on the so far unseen[6] test set $\mathbb{U}$ is evaluated yielding the *Generalization Error*.

Modeling this from a stochastic point of view allows to derive a very common loss function [19, Ch. 3, Ch. 5]: let the distribution of $\mathbb{D}$ be described by a probability density function p_{data}. The training set then can be described by a joint *Empirical Distribution*

$$\hat{p}_{\text{data}}(x, y) := \frac{1}{|\mathbb{T}|} \sum_{(x', y') \in \mathbb{T}} \delta((x, y) - (x', y')), \quad \delta(a) := \begin{cases} 1, & \text{if } a = 0 \\ 0, & \text{else} \end{cases} \tag{4.1}$$

assuming that samples are identically distributed. As well, the data distribution produced by the machine learning model, i.e. here the function f, can be expressed via a probability density function $p_{\text{model}}(\cdot; W)$ parametrized by the weights of the model. The goal of training is then, to achieve $p_{\text{model}} \approx \hat{p}_{\text{data}}$. In particular, for $(x, y) \in \mathbb{T}$ the model should should map the sample x to the label y, i.e. $f_W(x) \overset{!}{=} y$. Hence, in particular one is interested in the conditional distribution $p_{\text{model}}(y|x; W)$. To assign the quality of this mapping quantitatively, an adequate similarity measure for these distributions is needed:

Definition 11 (Kullback-Leibler Divergence)
Given two probability distributions p, q the *Kullback-Leibler Divergence* between p and q is defined as

$$\text{KLD}(p \| q) := \int p(x) \ln \frac{p(x)}{q(x)} dx = \mathbb{E}_{x \sim p(x)} \left[\ln \frac{p(x)}{q(x)} \right]. \qquad \square$$

[4] This term will be clarified in the following.
[5] Usually, $|\mathbb{T}| \gg |\mathbb{U}|$, commonly in a 90:10 split.
[6] i.e. it has not been processed by the machine learning algorithm during training.

Note that the KLD is non-negative and zero if and only if $p = q$ but not symmetric. Applying this, the loss of a model on the training set can be expressed as

$$\begin{aligned}\mathrm{KLD}(\hat{p}_{\text{data}} \| p_{\text{model}}) &= \mathbb{E}_{(x,y)\sim\hat{p}_{\text{data}}} \left[\ln \hat{p}_{\text{data}}(x,y) - \ln p_{\text{model}}(y|x;W)\right] \\ &= \mathbb{E}_{(x,y)\sim\hat{p}_{\text{data}}} \left[\ln \hat{p}_{\text{data}}(x,y)\right] - \mathbb{E}_{(x,y)\sim\hat{p}_{\text{data}}} \left[\ln p_{\text{model}}(y|x;W)\right]. \end{aligned} \tag{4.2}$$

The goal is to find W such that $f_{W^*} \approx f^*$ and that the expression above is minimal, i.e.

$$W^* := \arg\min_{W\in\mathbb{W}} \; L(W) = \arg\min_{W\in\mathbb{W}} \; -\mathbb{E}_{(x,y)\sim\hat{p}_{\text{data}}} \left[\ln p_{\text{model}}(y|x;W)\right]$$

For this minimization, one can omit the first addend of equation (4.2) as it does not depend on W. Then a loss function can be defined as

$$L(W,\mathbb{T}) := -\mathbb{E}_{(x,y)\sim\hat{p}_{\text{data}}} \left[\ln p_{\text{model}}(y|x;W)\right]. \tag{4.3}$$

As this equals

$$L(W,\mathbb{T}) = H(\hat{p}_{\text{data}}) + \mathrm{KLD}(\hat{p}_{\text{data}} \| p_{\text{model}})$$

with $H(p) := -\mathbb{E}_{z\sim p}\left[\ln p(z)\right]$ the *Shannon Entropy*, a measure of uncertainty in a distribution, the right side of expression (4.3) is commonly termed the *Cross-Entropy* of $\hat{p}_{\text{data}}$ and p_{model} while $L(W)$ is the *Cross-Entropy Loss*.

As an alternative to the above, relying on the cross-entropy loss as a performance measure can also be justified based on the concept of likelihood [19, Sec. 5.5]: A so-called likelihood function, loosely spoken, assigns a value to any parameter configuration of a parametrized probability distribution while fixing values for the random variable. Here, the maximum likelihood over all model parameters W is

$$W^* := \arg\max_{W\in\mathbb{W}} \prod_{(x,y)\in\mathbb{T}} p_{\text{model}}(y|x;W)$$

assuming that the samples are independent identically distributed (iid). Not changing the result of arg max , one can apply a logarithmic transform to prevent numerical underflow as well as division by $|\mathbb{T}|$ yielding the expected value over the data distribution:

$$\begin{aligned} W^* &= \arg\max_{W\in\mathbb{W}} \; \frac{1}{|\mathbb{T}|} \ln \prod_{(x,y)\in\mathbb{T}} p_{\text{model}}(y|x;W) = \arg\max_{W\in\mathbb{W}} \; \frac{1}{|\mathbb{T}|} \sum_{(x,y)\in\mathbb{T}} \ln p_{\text{model}}(y|x;W) \\ &\approx \arg\max_{W\in\mathbb{W}} \; \mathbb{E}_{(x,y)\sim\hat{p}_{\text{data}}} \left[\ln p_{\text{model}}(y|x;W)\right] = \arg\min_{W\in\mathbb{W}} \; -\mathbb{E}_{(x,y)\sim\hat{p}_{\text{data}}} \left[\ln p_{\text{model}}(y|x;W)\right] \end{aligned}$$

This matches exactly the expression (4.3) derived via the KLD.

In practice, this is commonly applied to classification tasks, i.e. guessing a label y for a sample x: in this case, the label is often represented as a so-called *One-hot*

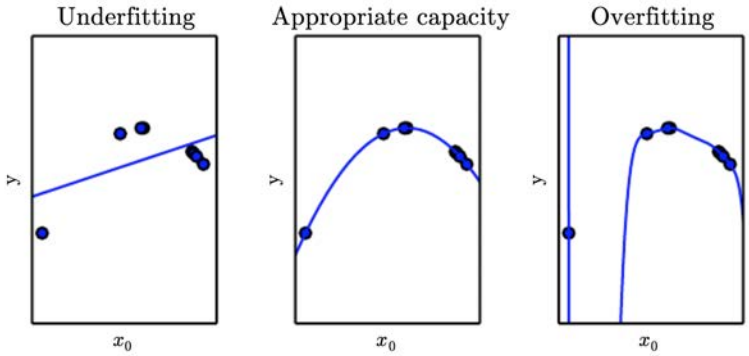


Figure 4.1: Examples for underfitting of a linear model (Left), fitting of a quadratic function (Center) and overfitting of a polynomial of higher degree (Right) taken from [19, Sec. 5.2]

.

Vector or a *Multi-hot Vector* $y \in \{0,1\}^{d_y}$ depending on whether $\|y\| \leq 1$ is required or not. Then the cross-entropy loss of the model f_W can be computed as

$$
\begin{aligned}
L(W,\mathbb{T}) &= -\mathbb{E}_{(x,y)\sim\hat{p}_{\text{data}}}\left[\ln p_{\text{model}}(y|x;W)\right] = -\int \hat{p}_{\text{data}}(x,y)\ \ln p_{\text{model}}(y|x;W)\ d(x,y) \\
&\approx \sum_{(x,y)\in\mathbb{T}} \hat{p}_{\text{data}}(x,y)\ln p_{\text{model}}(y_1,\ldots,y_m|x;W) \\
&\overset{(*)}{=} \sum_{(x,y)\in\mathbb{T}} \hat{p}_{\text{data}}(x,y)\ln\prod_{i=0}^{m} p_{\text{model}}(y_i|x;W) \\
&\overset{(4.1)}{=} \sum_{(x,y)\in\mathbb{T}} \left(\frac{1}{|\mathbb{T}|}\sum_{(x',y')\in\mathbb{T}} \delta((x,y)-(x',y'))\right)\left(\sum_{i=0}^{m}\ln p_{\text{model}}(y_i|x;W)\right) \\
&= \frac{1}{|\mathbb{T}|}\sum_{(x,y)\in\mathbb{T}}\sum_{i=1}^{m} y_i \ln(f_W^{(i)}(x)) + (1-y_i)\ln(1-f_W^{(i)}(x)) \qquad (4.4)
\end{aligned}
$$

where $f_W^{(i)}$ should indicate the i-th component of the output of f_W. At $(*)$ it is assumed that the samples are iid over classes. For classification tasks, this necessitates in particular a class-balanced training set. Assuming also the samples in the whole domain $\mathbb{D}$ to be iid, it is justified to expect f to generalize form the training set $\mathbb{T}$ to unseen data. In other words, at least a randomly chosen approximation of f^* should perform equally on both, training and test set.

However, tailoring an approximation to the training set can instead of fitting the learning goal fail in ways that either lead to

(i) bad performance on the training set called *Underfitting*, i.e. the model does not approximate f^* sufficiently well

(ii) good performance on the training set, but bad performance on the test set called *Overfitting*, i.e. the model does not generalize beyond $\mathbb{T}$.

The risk of those errors is closely related to the representational capacity of the family of functions that f can be drawn from by the algorithm called the *Hypothesis Space*: while a small hypothesis space limits approximation power and thus can cause underfitting, a too large search space might allow a too fine-grain approximation of training data and thus is prawn to overfitting [19, Sec. 5.2] (Refer to Figure 4.1 for a visualization). The hypothesis space is largely determined by the parametric form of f and the possibilities of the machine learning algorithm to adjust these parameters during training. Besides, f can employ parameters that control its behavior and in particular design choices which are not adjustable during training but instead fixed beforehand by a human. Those are called *Hyperparameter*s [19, Sec. 5.3]. To tune them, one evaluates the performance of different training results obtained from different hyperparameter configurations and adjusts those parameters manually. In order to avoid overfitting them to the test set, one splits off another partition $\mathbb{V}$ of $\mathbb{D} = \mathbb{T} \dot\cup \mathbb{V} \dot\cup \mathbb{U}$ called the *Validation Set*[7]. This is held out from the adaption process of the trainable parameters, but unlike $\mathbb{U}$ it is not just used once for evaluating the trained model, but multiple times for different settings and stages of training. This provides insights to the development of the model's generalization ability over training.

4.2 The Learning Process

4.2.1 Optimization

The task of supervised machine learning as described previously is essentially a task of finding the best approximation of an unknown function. In other words, the goal is to maximize the quality of the approximation by minimizing a loss function through adjusting parameters of the approximating function. In general:

Definition 12 (Optimization)
The *Optimization* of a function $L : \mathbb{R}^n \to \mathbb{R}$ is the task of finding the *Optimum*

$$x^* := \arg\min_{x \in \mathbb{R}^n} L(x) \quad \text{or} \quad x^* := \arg\max_{x \in \mathbb{R}^n} L(x).$$

The process of searching x^* is called *Minimization* or *Maximization* of L, accordingly. The function L is called *Objective Function*. □

As

$$\arg\max_{x \in \mathbb{R}^n} L(x) = \arg\min_{x \in \mathbb{R}^n} -L(x),$$

[7] Commonly this roughly equals a 80:20 split [19, Sec. 5.3].

the sequel focuses on minimization only. The basic formulations below are largely taken from [64, Sec. 2.1] with minor adaptions.

Definition 13 (Minimum)
A function $L : \mathbb{R}^n \to \mathbb{R}$ has a *Local Minimum* in $x \in \mathbb{R}^n$, if there is a *Neighborhood* $U \subseteq \mathbb{R}^n$ of x, i.e. it contains an *Open Set* $B(x, \epsilon)$, formally

$$\exists B \subseteq U : \ x \in B \text{ and } \exists \epsilon \in \mathbb{R}_{>0} : \forall x' \in \mathbb{R}^n : \|x - x'\| < \epsilon \implies x' \in B \tag{4.5}$$

and

$$\forall u \in U : \ L(x) \leq L(u). \tag{4.6}$$

In case $U = \mathbb{R}^n$, the point x is in particular a *Global Minimum*. □

A maximum is defined analogously. When speaking of a minimum (maximum) without further specification, this refers to a local minimum (maximum). To find such a minimum, the derivative of L along a certain vector is defined:

Definition 14 (Gradient)
For a differentiable function $L : \mathbb{R}^n \to \mathbb{R}$ and $x \in \mathbb{R}^n$, the row vector of all partial derivatives

$$\nabla L(x) := \left[\tfrac{\partial L}{\partial x_1}(x), \ldots, \tfrac{\partial L}{\partial x_n}(x)\right]$$

is called the *Gradient* of L in x. □

Definition 15 (Directional Derivative)
A function $L : \mathbb{R}^n \to \mathbb{R}$ is *directionally differentiable* in $x \in \mathbb{R}^n$ if

$$\forall v \in \mathbb{R}^n : \ D_v L(x) := \lim_{h \to 0^+} \frac{L(x + hv) - l(x)}{h} \neq \pm\infty.$$

Then, $D_v L(x)$ is the *Directional Derivative* for L in x in direction $v \in \mathbb{R}^n$. If L is differentiable[8], then

$$D_v L(x) := \frac{d}{d\alpha} L(x + \alpha v) \Big|_{\alpha=0} = v^T \nabla L(x).$$ □

Proposition 16
If a differentiable function $L : \mathbb{R}^n \to \mathbb{R}$ has a minimum at $x \in \mathbb{R}^n$, it holds

$$\forall v \in \mathbb{R}^n : \ D_v L(x) = \nabla L(x) = 0.$$ □

[8] i.e. $L'(x) = \lim_{h \to 0} \frac{L(x+h) - L(x)}{h} \neq \pm\infty$

PROOF Let $v \in \mathbb{R}^n$ arbitrarily. As x is a local minimum within an open set $B(x, \epsilon)$, it holds in particular for $h < \frac{\epsilon}{\|v\|}$

$$\|(x \pm hv) - x\| = |h| \, \|v\| < \epsilon \overset{(4.5)}{\Longrightarrow} x \pm hv \in B(x, \epsilon) \overset{(4.6)}{\Longrightarrow} L(x) \leq l(x \pm hv) \quad (4.7)$$

and thus

$$D_v L(x) = \lim_{h \to 0^+} \frac{L(x + hv) - L(x)}{h} \overset{(4.7)}{\geq} 0. \quad (4.8)$$

Overall it is

$$0 \overset{(4.8)}{\leq} D_{-v} L(x) \overset{f \text{ diff.able}}{=} -v^T \nabla L(x) = -\underbrace{D_v L(x)}_{\geq 0 \text{ by } (4.8)} \leq 0$$

hence $D_v L(x) = 0$ and as this holds for all v, it implies $\nabla L(x) = 0$. ∎

Using this, candidate points for optima can be found. To locate a potential minimum, an algorithm can follow the negative gradient from every point in order to go along the "steepest" direction:

Remark 17 The direction $v \in \mathbb{R}^n$ with minimal directional derivative of a differentiable function $L : \mathbb{R}^n \to \mathbb{R}$ in a point $x \in \mathbb{R}^n$ is a scalar multiple of the negative gradient $v = -\nabla L(x)$. [19, Sec. 4.3]

PROOF First, note, that by the law of cosine and the definition of the euclidean norm it holds for $a, b \in \mathbb{R}^n$ that

$$\begin{aligned} \|a\|^2 + \|b\|^2 - 2 \|a\| \, \|b\| \cos\gamma = \|a - b\|^2 = \|a\|^2 - 2a^T b + \|b\|^2 \\ \Leftrightarrow \quad \|a\| \, \|b\| \cos\gamma = a^T b. \end{aligned} \quad (4.9)$$

Considering only unit ball vector's[9], i.e. $\|v\| = 1$, the minimum value of the directional derivative from x is

$$\begin{aligned} \min_{v \in \mathbb{R}^n, \|v\|=1} D_v L(x) &= \min_{v \in \mathbb{R}^n, \|v\|=1} v^T \nabla L(x) \\ &\overset{(4.9)}{=} \min_{v \in \mathbb{R}^n, \|v\|=1} \|v\| \, \|\nabla L(x)\| \cos\gamma \\ &= \|\nabla L(x)\| \min_{v \in \mathbb{R}^n, \|v\|=1} \cos\gamma \\ &= -\|\nabla L(x)\| = -\frac{1}{\|\nabla L(x)\|} \|\nabla L(x)\|^2 \\ &\overset{(4.9)}{=} -\frac{1}{\|\nabla L(x)\|} \nabla^T L(x) \nabla L(x) = D_{-\frac{\nabla L(x)}{\|\nabla L(x)\|}} L(x) \end{aligned}$$

[9] Obviously, the magnitude of the direction vector is not relevant for finding the "steepest" direction.

with γ being the angle described by v and the gradient of L in x. As one can see, it holds

$$\arg\min_{v \in \mathbb{R}^n} \; D_v L(x) = -\nabla L(x).$$

■

4.2.2 Descent Algorithms

This can be used for *Gradient Descent* [10], a simple iterative procedure of minimizing a loss function $L_{\mathbb{T}} := L(\cdot, \mathbb{T})$ evaluated on the training set $\mathbb{T}$ as above: as described in Algorithm 4, starting from some parameter configuration $W \in \mathbb{W}$, the algorithm moves down the direction of steepest descent provided as $-\nabla L_{\mathbb{T}}$ according to Remark 17 until the gradient is zero, i.e. a candidate for a minimum has been reached. The step size $\epsilon \in \mathbb{R}_{>0}$ for this procedure is termed the *Learning Rate*.

Algorithm 4: Gradient Descent

Input: objective function L, initial model parameters W, training set $\mathbb{T}$, learning rate ϵ

while $\nabla L_{\mathbb{T}}(W) \neq 0$ **do**

  Apply update: $W \leftarrow W - \epsilon \nabla L_{\mathbb{T}}(W)$;

end

Output: W, a candidate for a local minimum of $L_{\mathbb{T}}$

In practice, it would often be sufficient to have $\nabla L_{\mathbb{T}}$ close[10] to zero. [19, Sec. 4.3] Revisiting equation (4.4), one can see that evaluating $L_{\mathbb{T}}(W)$ has a complexity of $\mathcal{O}(|\mathbb{T}|)$ [19, Sec. 5.9], which can slow down machine learning on large training sets dramatically. Instead, the gradient can be estimated by iteratively computing the gradient for randomly sampled[11] very small subsets $B \subset \mathbb{T}$, so-called *Minibatches* [19, Sec. 5.9]. It is common to simply speak of *Batches* with a *Batch Size* of $|B|$, which is for reasons of computational efficiency usually a power of 2 [19, Sec. 8.1.3]. Obviously,

$$\lim_{|B| \to |\mathbb{T}|} \nabla L_B(W) = L_{\mathbb{T}}(W)$$

i.e. a larger batch size results in a better estimation of the gradient on the whole training set, so that there is a tradeoff between computational cost and accuracy. Nevertheless, it has been shown [77] that even small batches can foster generalization by adding noise when used together with a sufficiently small learning rate [19, Sec. 8.1.3]. The processing of one minibatch is called one *Training Step*.

[10] in the sense of the euclidean metric

[11] It is sufficient to randomly sample minibatches once at the beginning of training [19, Sec. 8.1.3].

Slightly modifying Algorithm 4 for use with minibatches according to [19, Alg. 8.1] results in so-called *Stochastic Gradient Descent*:

Algorithm 5: Stochastic Gradient Descent (SGD)

Input: objective function L, initial model parameters W, training set $\mathbb{T}$, learning rate ϵ

Sample minibatch $B \subset \mathbb{T}$ randomly;

while $\nabla L_B(W) \neq 0$ **do**

  Sample minibatch $B \subset \mathbb{T}$ randomly;

  Apply update: $W \leftarrow W - \epsilon \nabla L_B(w)$;

end

Output: W, a candidate for a local minimum of $L_{\mathbb{T}}$

As mentioned before, a state with the gradient equal to zero won't be reached in practice, especially with stochastic gradient descent, which is only working with a minibatch-based estimation of the gradient and thus introduces a source of noise [19, Sec. 8.3.1]. Instead, after some time of training, L_B will not decrease further or even grow due to overfitting, i.e. refining the minimum of the training error too much. To tackle this, one simply does not continue training beyond this critical point from which no further improvements are made and adapts the stopping criterion of the optimization algorithm such as Algorithm 5 accordingly. This technique is called *Early Stopping* [19, Sec. 7.8].

The learning process twoards this point of stopping is highly influenced by the learning rate: this hyperparameter has to be chosen carefully, as it is crucial for the following tradeoffs coming with (stochastic) gradient descent [19, Sec. 8.3.1]:

(i) A small learning rate is useful to refine the approximation of a (potential) minimum, but will slow down the algorithm as a whole.

(ii) A large learning rate accelerates training at the beginning but can make the algorithm "jump" over a minimum (repeatedly).

(iii) When gradients are small, descent algorithms progress slowly.

(iv) When gradients are "noisy", i.e. very inconsistent over training time, a descent algorithm might zig-zag instead of approaching the minimum more directly.

The first two issues can be addressed by decreasing the learning rate over time between training step 0 and step T from ϵ_0 to ϵ_T, such that (as described among others in [19, Sec. 8.3.1]) for step t the learning rate is

$$\epsilon_t = \left(1 - \frac{t}{T}\right) \epsilon_0 + \frac{t}{T} \epsilon_T.$$

Here, a simple linear decrease of the learning rate is described, but of course, there are more sophisticated strategies as well.

Momentum The two other issues can be addressed by using a technique first described in [57]: instead of directly using the gradient, parameters are updated using a moving average of the gradients of the previous steps to control not only the direction in which to move but also the velocity, a so-called *Momentum*. This procedure is outlined in Algorithm 6 in reference to [19, Alg. 8.2]. Despite Stochastic Gradient Descent with Momentum is a rather basic and light-weight optimization algorithm, it has proven to be highly successful in various complex tasks, for instance [31], [68] and [23].

Algorithm 6: Stochastic Gradient Descent with Momentum

Input: objective function L, initial model parameters W, training set $\mathbb{T}$, learning rate ϵ, momentum decay α

Sample minibatch $B \subset \mathbb{T}$ randomly;

while $\nabla L_B(W) \neq 0$ **do**

  Sample minibatch $B \subset \mathbb{T}$ randomly;

  Calculate update: $v \leftarrow \alpha v - \epsilon \nabla L_B(w)$;

  Apply update: $W \leftarrow W + v$;

end

Output: W, a candidate for a local minimum of $L_{\mathbb{T}}$

In Algorithm 6, the decay parameter α controls the influence of gradients from previous steps and thus the balance between exploration of a "new search space" and the exploitation in the current one, i.e. the refinement. This is yet another hyperparameter, for which values such as $0.5, 0.9, 0.99$ are used widely and may be adopted over time just like the learning rate [19, Sec. 8.3.2].

Again, such a hyperparameter adaption follows a rather arbitrary, "external" strategy which does not take into account the gradients' magnitude in each step . Further, it applies the same learning rate to the update of every parameter. Both is critical as it leads to divergence (hampers convergence) of the optimization algorithm when a high (low) learning rate is used in regions with high (low) curvature and accordingly large (small) gradients [6, Sec. 2].

AdaGrad To mitigate this, a descent algorithm called *AdaGrad (Adaptive Gradient)* proposed by [15] calculates an individual learning rate for each parameter in each step through scaling the "global" learning rate by an accumulation of the corresponding partial derivatives from all previous training steps: Letting $g := \left[g_1, \ldots, g_n\right]^T := \nabla L_B(W)$ denote the gradient and defining initially $R = 0 \in \mathbb{R}^{n \times n}$, AdaGrad aims to use the outer product matrix $G := gg^T$ to update W by

$$R \leftarrow R + G$$
$$W \leftarrow W - \epsilon R^{-\frac{1}{2}} g$$

But as finding a square root of R, i.e. matrix A such that $AA = R$, is computationally impractical [15, Sec. 1.1], AdaGrad uses a slightly different update rule of

$$W \leftarrow W - \epsilon \operatorname{diag}(R)^{-\frac{1}{2}} g = W - \epsilon \begin{bmatrix} \frac{1}{\sqrt{R_{11}}} & & \\ & \ddots & \\ & & \frac{1}{\sqrt{R_{nn}}} \end{bmatrix} g$$

where R_{ii} equals the cumulative sum of the values of $g_i^2 = \frac{\partial}{\partial W_i} L_B^2$ over all iterations so far. Hence, the overall algorithm as described in [15, Sec. 1.1] can be written as:

Algorithm 7: Adaptive Gradient Descent (AdaGrad)

Input: objective function L, initial model parameters W, training set $\mathbb{T}$, global learning rate ϵ, negligibly small constant δ for numerical stability

Initialize $r := 0 \in \mathbb{R}^n$;

Sample minibatch $B \subset \mathbb{T}$ randomly;

while $\nabla L_B(W) \neq 0$ **do**

Sample minibatch $B \subset \mathbb{T}$ randomly;

Let $\left[g_1, \ldots, g_n\right] := \nabla L_B(W)$;

Accumulate squared partial derivatives: $r \leftarrow r + \begin{bmatrix} g_1^2 \\ \vdots \\ g_n^2 \end{bmatrix}$;

Apply update: $W \leftarrow W - \epsilon \begin{bmatrix} \frac{1}{\sqrt{r_1 + \delta}} g_1 \\ \vdots \\ \frac{1}{\sqrt{r_n + \delta}} g_n \end{bmatrix}$;

end

Output: W, a candidate for a local minimum of $L_{\mathbb{T}}$

Adagrad has been successfully applied among others in [65]. But it can be seen from Algorithm 7, that the larger the (accumulated) partial derivative for one parameter is, the lower will be the scaled learning rate used to update this parameter and vice versa. While this is fine for a convex environment, i.e. curves with permanently negative curvature where every local minimum is a global one, for non-convex functions a learning rate being low due to gradients being large at the first steps can lead to getting stuck in such a local minimum instead of finding the global one.

RMSProp This effect can be reduced by limiting the influence of the previously observed gradients in each step by using an exponentially weighted moving average of gradients instead of the cumulative sum [19, Sec. 8.5.2], [6, Sec. 4]. Modifying AdaGrad this way results in the so-called *RMSProp (Root Mean Square Propagation)*

algorithm by [69]: RMSProp works just like AdaGrad except from the accumulation of squared partial derivatives: the update rule used by RMSProp is

$$r \leftarrow \alpha r + (1-\alpha) \begin{bmatrix} g_1^2 \\ \vdots \\ g_n^2 \end{bmatrix} \tag{4.10}$$

where the decay rate $\alpha \in [0, 1)$ is again a hyperparameter. But initializing $r = 0$ introduces some sort of bias into the moving average such that for instance

$$r_1 = (1-\alpha)G_1 \neq G_1$$

with the subscripts of r and G indicating the value of the respective variable in iteration t of the algorithm and with $G := \left[g_1^2, \ldots, g_n^2\right]^T$. Formally, as outlined in [29, Sec. 3], consider G_t being independently drawn from a gradient distribution $G_t \sim p_{\text{grad}}(G_t)$, then unroll the recursive update rule (4.10) to

$$r_t = (1-\alpha) \sum_{i=1}^{t} G_i \alpha^{t-i}$$

and compare

$$\begin{aligned}
\mathbb{E}\left[r_t\right] &= \mathbb{E}_{G_i \sim p_{\text{grad}}} \left[(1-\alpha) \sum_{i=1}^{t} G_i \alpha^{t-i} \right] \\
&= (1-\alpha) \sum_{i=1}^{t} \alpha^{t-i} \underbrace{\mathbb{E}_{G_i \sim p_{\text{grad}}}\left[G_i\right]}_{=\mathbb{E}_{G_t \sim p_{\text{grad}}}[G_t]} = \mathbb{E}_{G_t \sim p_{\text{grad}}}\left[G_t\right] (1-\alpha) \sum_{i=1}^{t} \alpha^{t-i} \\
&= \mathbb{E}_{G_t \sim p_{\text{grad}}}\left[G_t\right] \sum_{i=1}^{t} (\alpha^{t-i} - \alpha^{t-i+1}) = \mathbb{E}_{G_t \sim p_{\text{grad}}}\left[G_t\right] (1-\alpha^t) \\
&\overset{!}{=} \mathbb{E}_{G_t \sim p_{\text{grad}}}\left[G_t\right]
\end{aligned}$$

Hence, the bias can be corrected by

$$\hat{r}_t \leftarrow \frac{r_t}{1-\alpha^t}.$$

Obviously, this holds for an update rule without squaring the gradient values as well.

Adam A descent algorithm incorporating such a bias correction is *Adam (Adaptive Moments)* proposed by [29]: to a certain extent this can be seen as a merge of RMSProp and Momentum as it maintains exponential moving averages of the gradient itself (called the first moment) and the element-wisely squared gradient (called

second moment) as well as a bias correction for both. As outlined in Algorithm 8, just like in RMSPRop the second moment is used to scale the partial derivatives, which are estimated by the first moment, individually:

Algorithm 8: Adaptive Moments (Adam)

Input: objective function L, initial model parameters W, training set $\mathbb{T}$, global learning rate ϵ, decay rates $\alpha, \beta \in [0, 1)$, negligibly small constant δ for numerical stability

Initialize $s, r := 0 \in \mathbb{R}^n$;

Initialize step $t := 1$;

Sample minibatch $B \subset \mathbb{T}$ randomly;

while $\nabla L_B(W) \neq 0$ **do**

Sample minibatch $B \subset \mathbb{T}$ randomly;

Let $\left[g_1, \ldots, g_n\right] := \nabla L_B(W)$;

Compute first moment: $s \leftarrow \beta s + (1 - \beta) \begin{bmatrix} g_1 \\ \vdots \\ g_n \end{bmatrix}$;

Compute second moment: $r \leftarrow \alpha r + (1 - \alpha) \begin{bmatrix} g_1^2 \\ \vdots \\ g_n^2 \end{bmatrix}$;

First moment bias correction: $\hat{s} \leftarrow \frac{s}{1-\beta^t}$;

Second moment bias correction: $\hat{r} \leftarrow \frac{r}{1-\alpha^t}$;

Apply update: $W \leftarrow W - \epsilon \begin{bmatrix} \frac{1}{\sqrt{\hat{r}_1}+\delta} \hat{s}_1 \\ \vdots \\ \frac{1}{\sqrt{\hat{r}_n}+\delta} \hat{s}_1 \end{bmatrix}$;

Increment $t \leftarrow t + 1$;

end

Output: W, a candidate for a local minimum of $L_{\mathbb{T}}$

Despite this combination "does not have a clear theoretical motivation" [19, Sec. 8.5.3], Adam has turned out to be extremely successful: it outperforms SGD, AdaGrad and RMSProp [29, Sec. 6] and has shown great success in recent work similar to or relevant for this project of music synthesis such as [27, 72, 17, 49, 59, 36, 28]. Therefore it can be seen as a de-facto standard in this field.

A big advantage of Adam is, that according to the authors its hyperparameters require little fine-tuning for a specific task [29], [19, Sec. 8.5.3]. Default values recommended by [29] for this algorithm are $\epsilon = 0.001, \alpha = 0.9$ and $\beta = 0.999$.

4.2.3 Backpropagation

Note: this section is written relying more on basic knowledge gained from various lectures and projects in the field rather than an explicit source. Nevertheless, in parts it loosely follows [19, Sec. 6.5], in particular regarding the idea of using computational graphs.

Definition 18 (Jacobian)
For a differentiable multi-variate function,

$$f : \mathbb{R}^n \to \mathbb{R}^m, \quad x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \mapsto \begin{bmatrix} f_1(x) \\ \vdots \\ f_m(x) \end{bmatrix}$$

the matrix consisting of the gradients of the component functions f_i as rows

$$Df(x) := \begin{bmatrix} \nabla f_1(x) \\ \vdots \\ \nabla f_m(x) \end{bmatrix} \in \mathbb{R}^{m \times n}$$

is called the *Jacobian* of f in x. □

All of the algorithms described above heavily rely on the gradient of the objective function being evaluated multiple times. But evaluating the gradient can be computationally expensive: consider a chain of n differentiable functions

$$f := f_n \circ \ldots \circ f_1, \quad f_i : \mathbb{R}^{m_i} \to \mathbb{R}^{m_{i+1}} \tag{4.11}$$

where $(f_j \circ f_i)(\cdot) := f_j(f_i(\cdot))$ denotes composition. According to the chain rule, it holds

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \quad \text{for} \quad z := g(y),\ y := h(x)$$

for scalar functions $g, h : \mathbb{R} \to \mathbb{R}$ and

$$D(g \circ h)(x) = Dg(h(x))\ Dh(x) \in \mathbb{R}^{k \times n}$$

for the multivariate case with $h : \mathbb{R}^n \to \mathbb{R}^m$, $g : \mathbb{R}^m \to \mathbb{R}^k$ in $x \in \mathbb{R}^n$. Applying this recursively, the derivative of function (4.11) in $x \in \mathbb{R}^{m_1}$ is given as

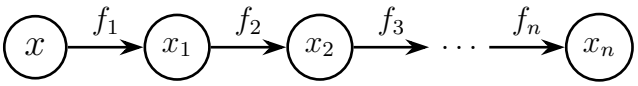
$$Df(x) = Df_n(f_{n-1}(\ldots(f_1(x)))) \cdot Df_{n-1}(f_{n-2}(\ldots(f_1(x)))) \cdot \ldots \cdot Df_1(x) \tag{4.12}$$

It can be seen, that equation (4.12) is highly redundant as a straightforward implementation would evaluate the same sub-expressions multiple times causing high runtime.

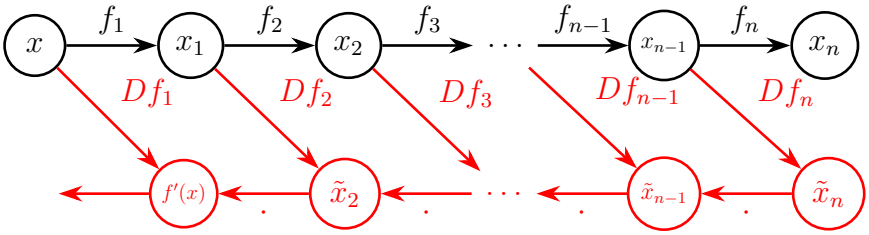
This can be avoided by evaluating the derivative of each chain element locally instead of evaluating the derivative "as a whole": consider the function (4.11) as a *Computational Graph*, i.e. an acyclic directed graph with variables and "intermediate results" as nodes and operations as edges. For simplicity, in this section a computational graph is restricted to have only one input node and not more than one child per node[12]. For this, let

$$x_i := (f_i \circ \ldots \circ f_1)(x), \quad i \in \{1, \ldots, n\}$$

denote the input to f_{i+1} or respectively the output of f_i. Then the computational graph representing the forward pass of the input value through the chain of functions is:



Then the derivative can be computed and evaluated for each node *locally* starting from the output node and then this value can be reused in all parent nodes: computation starts with the local derivative of the edge to the last node $\tilde{x}_n := Df_n(x_{n-1})$ which is then passed to the parent node to compute its local derivative as $\tilde{x}_{n-1} := Df'_{n-1}(x_{n-2}) \cdot \tilde{x}_n$ and so on. This results in the backward pass:



In this procedure, the values of the components of $f'(x)$ are "propagating backwards through the graph" starting from the output node. Hence, the algorithm performing

[12] This is without loss of generality as every node, i.e. every variable does not have to be a scalar value but instead can also be a vector or matrix [19, Sec. 6.5.1].

this computation is called *Back Propagation*, first described in [62]. In pseudocode:

Algorithm 9: Simplified Back Propagation

Input: A computational graph $G = (V, E)$ consisting of a set V of nodes and a set E of edges containing one start node and not more than one child per node.
Input: An input value x
Initialize $J \leftarrow 1$;
Initialize $v \in V$ the output node of the graph;
while `hasParent`(v) **do**
  Let $p \leftarrow$ `parent`(v);
  Let $f \leftarrow$ `value`(`incomingEdge`(v));
  Update $J \leftarrow Df(\texttt{value}(p)) \cdot J$;
  Optionally: use J to update parameters of f or store J in some data structure for later usage;
end
Output: The Jacobian J of the function represented by G at point x

Note that the requirement to G of having only one input node and not more than one child per node directly implies that the graph has a unique output node and one parent per node.

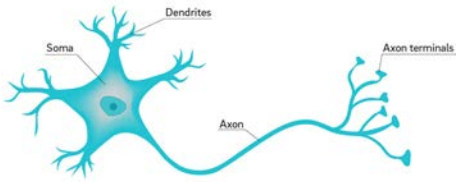
This algorithm is just a brief sketch of the idea. Of course other implementations might be preferably depending on the use case. Variants might be recursive, store the "intermediately" computed derivatives of all f_i to update their parameters later in bulk (as it is done in [62]) or they might be updated on the fly as well. Most current machine learning frameworks such as TensorFlow [1] do not evaluate derivatives directly but first compute symbolic derivatives for later evaluation which are used to augment the computational graph just as visualized above [19, Sec. 6.5.5]. Overall, backpropgation enables the efficient application of gradient-based optimization methods to machine learning models of high complexity[13].

4.3 Neural Networks

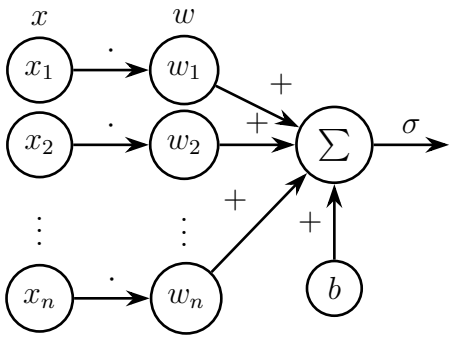
4.3.1 Neurons

One technique from the field of Machine Learning that has recently succeeded in a variety of complex applications and thus gathered lots of attention in research are so-called *Neural Networks*. As the name suggests, this biology-inspired algorithmic concept tries to mimic the behavior of the human brain: it consists of multiple

[13] in the sense of the length of function chains and the number of parameters.



(a) The biological concept of a neuron receiving inputs via multiple dendrites, which are processed inside the cell body and fire an output over the axon in case of an activation of the neuron.



(b) Computational Graph of a neuron with input $x \in \mathbb{R}^n$, weights $w \in \mathbb{R}^n$, bias $b \in \mathbb{R}$ and an activation function σ.

Figure 4.2: Biological and mathematical illustration of a neuron.
Left graphic taken and modified from: https://ucsdnews.ucsd.edu/pressrelease/why_are_neuron_axons_long_and_spindly

computational cells, the *neurons*, each receiving multiple numeric inputs which are reduced by weighted addition to produce an output signal scaled by a non-linear *Activation Function*[14] and shifted by a *bias* term (See Figure 4.2). In the following, this concept will be modeled formally while mostly relying on [19, Pt. II, Ch. 6]:

Definition 19 (Neuron)
Given weights $w \in \mathbb{R}^n$, a non-linear function $\sigma : \mathbb{R} \to \mathbb{R}$ and a *Bias* $b \in \mathbb{R}$, a *Neuron* is a function

$$h : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}, \quad (x, w, b) \mapsto \sigma(\sum_{i=0}^{n} w_i x_i + b) \; = \; \sigma(w^T x + b) \tag{4.13}$$

processing an input x of dimension $n \in \mathbb{N}$. The function σ is called *activation function*[15]. □

The construction of Definition 19 is visualized in Figure 4.2b.

[14] Loosely spoken within the analogy, the activation function determines whether and to what extent the neuron will be activated by the input signals. Details can be found in the sequel.

[15] The only requirement for σ at this point is to be a scalar function, which is non-linear (i.e. not of the form $\sigma(x) = ax + t$). There are multiple other desirable properties such as differentiability or monotonicity, but none of them is necessarily required for *all* activation functions. Thus, the definition is kept broad here intentionally. Details follow. It is not explicitly defined as a parameter of h as it is fixed at the design time of a neural network unlike w and b.

Notation For $u := (u_1, \dots, u_n)^T \in \mathbb{R}^n, v := (v_1, \dots, v_m)^T \in \mathbb{R}^m$ let

$$\begin{bmatrix} u \\ v \end{bmatrix} := (u_1, \dots, u_n, v_1, \dots, v_m)^T.$$

Remark 20 (Bias Packing) Substituting $\hat{w} := \begin{bmatrix} w \\ b \end{bmatrix}$ and $\hat{x} := \begin{bmatrix} x \\ 1 \end{bmatrix}$, the function in (4.13) can be shorthanded

$$h_{\hat{w}}(x) := h(x, w, b) = \sigma(\hat{w}^T \hat{x}).$$

For simplicity and compact reading, w and b will be combined to one vector and commonly termed *Weights* instead of "weights and biases" in the following.

Multiple neurons can be arranged in a *Layer*:

Definition 21 (Fully-Connected Layer)
A *Fully-Connected Layer* or *Dense Layer* with $m \in \mathbb{N}$ *Units* is a function f comprised of a family $(h^{(i)}_{w_i})_{i \in \{1,\dots,m\}}$ of neurons sharing the same activation function σ and receiving an input $x \in \mathbb{R}^n$:

$$f : \mathbb{R}^n \times \mathbb{R}^{m \times n+1} \to \mathbb{R}^m, \quad (x, W) \mapsto \begin{bmatrix} h^{(1)}_{W_1}(x) \\ \vdots \\ h^{(m)}_{W_m}(x) \end{bmatrix}$$

Note, that the weights (and biases) of all $h^{(i)}$ have been summarized in a matrix $W \in \mathbb{R}^{m \times n+1}$ for which $W_i := w_i$ denotes the i-th row. The above can be shorthanded

$$\hat{f} : \mathbb{R}^n \times \mathbb{R}^{m \times n+1} \to \mathbb{R}^m, \quad (x, W) \mapsto \sigma(W\hat{x}) \quad \text{with} \quad \hat{x} := \begin{bmatrix} x \\ 1 \end{bmatrix}$$

where σ is applied element-wise. Again, write $\hat{f}_W := f(\cdot, W)$. □

The layer defined above is called *fully-connected* or *dense* as each single element of the input vector is fed to each neuron separately as visualized in figure Figure 4.3.

In particular, for further considerations the reader may note that the output size equals the number of units of the layer and has to be respected when "stacking" multiple of those layers to construct a neural *network*:

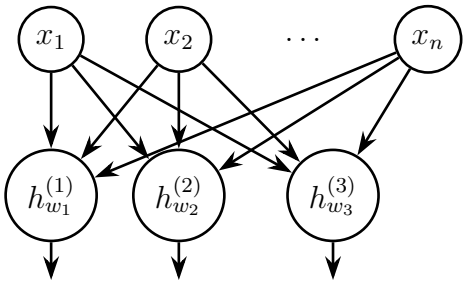


Figure 4.3: Computational Graph of a Fully-Connected Layer with input $x \in \mathbb{R}^n$ and three neurons $h^{(1)}_{w_1}$, $h^{(2)}_{w_2}$ and $h^{(3)}_{w_3}$ according to Definition 21.

Definition 22 (Neural Network)
A *Neural Network* of depth $d \in \mathbb{N}$ with input size $n \in \mathbb{N}$ consists of a family $(f^{(i)}_{W^{(i)}})_{i\in\{1,\dots,d\}}$ of fully-connected layers[16] using weights $W := (W^{(i)})_{i\in\{1,\dots,d\}}$ such that the input dimension of the i-th layer equals the number $m_{i-1} \in \mathbb{N}$ of units (and thus the output size) of the previous layer:

$$\forall i \in \{0,\dots,d\}: \quad f^{(i)}_{W^{(i)}} : \mathbb{R}^{m_{i-1}} \to \mathbb{R}^{m_i} \quad \text{with} \quad m_0 := n.$$

Then letting $x \in \mathbb{R}^n$ be some input, the neural network is given as

$$f_W(x) := f(x, W) := f^{(d)}_{W^{(d)}} \circ \dots \circ f^{(1)}_{W^{(1)}}.$$

In this setting, $f^{(1)}_{W^{(1)}}$ is called *Input Layer*, $f^{(d)}_{W^{(d)}}$ analogously *Output Layer* while $f^{(i)}_{W^{(i)}}, 1 < i < d$ are *Hidden Layer*s. □

4.3.2 Activation Functions

The neural network is intended to address a supervised learning task and thus to approximate f^* according to Definition 10. It is noteworthy that the non-linearity of σ according to Definition 19 is crucial for this ability: if σ was of the form $\sigma(y) = ay + t$, then a fully connected layer $f^{(i)}_W(x) = \sigma(W^{(i)}x) = aW^{(i)} \cdot + t$ would

[16] In general, of course any kind of (differentiable) function can be used as a layer of a neural network. But as it will be shown later in this work, the only other type of layer relevant for a computational analysis of this project (neglecting "non-computational" layers such as data reshaping) is a convolutional layer. It will be shown that a convolutional layer is computationally equivalent to a dense layer. Thus, the restriction to dense layers here appears reasonable.

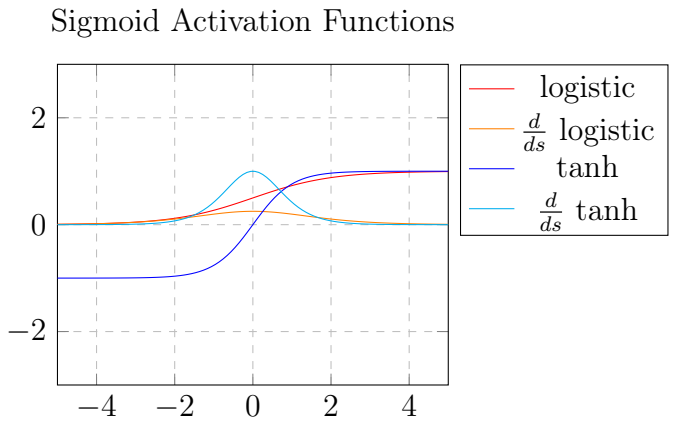


Figure 4.4: Plots of two common sigmoid activation functions, tanh and logistic along with their derivatives.

be a linear function and so would the whole network:

$$\begin{aligned} f_W(x) &= f^{(d)}_{W^{(d)}} \circ \ldots \circ f^{(1)}_{W^{(1)}} \\ &= aW^{(d)} ... aW^{(1)}\hat{x} + t + t \\ &= a^d \underbrace{\prod_{i=1}^{d} W^{(i)}}_{=:\widetilde{W}} x + dt =: \widetilde{\sigma}(\widetilde{W}x) \end{aligned}$$

This would heavily counteract the intention of using a neural network as a highly flexible function approximator. Further, as the calculations above show, with a linear σ the whole network could be reduced to one single layer. Hence, non-linearity is required for an activation function. Beside this, there are other desirable properties, yet none of them is necessary for every activation function. Those will be briefly described in the following while going through common choices for activation functions:

One wide-spread type of activation function since the early rise of neural networks are *Sigmoid Functions* [32, Sec. 4.4], [22, Sec. 1]:

Definition 23 (Sigmoid Function)
A bounded, differentiable and monotonically increasing function $\sigma : \mathbb{R} \to \mathbb{R}$ is called *Sigmoid Function* if the limits for $\pm\infty$ exist. □

The limited range, the convergence in both limits and its monotonicity nicely fit the biological analogy of a neuron which is either activated or not by its input signals, for which a higher weight results in a higher activation of the neuron. But of course, in particular differentiability and non-linearity are of practical relevance. It has been

shown early, that even a neural network of depth $d = 1$ using sigmoid activation is a universal function approximator, i.e. it can approximate arbitrary continuous scalar functions up to a negligible error [11]. For completeness, it shall be mentioned that this has been generalized to "locally-bounded piecewise-continuous *non-polynomial* activation functions" by [33] and to arbitrary functions by [25]. Those results are known as the *Universal Approximation Theorem*.

Common examples of sigmoid functions depicted in Figure 4.4 are the *Logistic Function*

$$\sigma(x) := \frac{1}{1 + e^{-x}}$$

and the *Tangens Hyperbolicus*

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

as described in [32, Sec. 4.4]. Despite one is an instance of the other, the terms *logistic function* and *sigmoid function* are often used interchangeably or even defined as synonyms [75]. The logistic function has major drawbacks: the derivative of the logistic function is given as

$$\frac{d}{ds}\frac{1}{1 + e^{-s}} = -\frac{-e^{-s}}{(1 + e^{-s})^2} = (\sigma(s))^2 e^{-s}.$$

Hence, for a neuron processing $x \in \mathbb{R}^n$, the gradient $\nabla h_w(x) = \left[\frac{\partial}{\partial x_1} h_w \quad \dots \quad \frac{\partial}{\partial x_n} h_w\right](x)$ consists of

$$\frac{\partial}{\partial x_i} h_w = \frac{\partial}{\partial x_i}\sigma(w^T x) = (\frac{\partial}{\partial x_i}\sigma)(w^T x) \cdot (\frac{\partial}{\partial x_i} w^T x) = (\sigma(w^T x))^2 e^{-w^T x} \cdot w_i \tag{4.14}$$

Therefore, with a length d chain of multiple neurons $h := h^{(d)}_{w_d} \circ \ldots \circ h^{(1)}_{w_1}$ such as in a neural network with multiple fully-connected layers, the gradient is provided by the chain rule as

$$\begin{aligned} \frac{\partial}{\partial x_i} h &= (\frac{\partial}{\partial x_i} h^{(d)}_{w_d}) \circ h^{(d-1)}_{w_{d-1}} \circ \ldots \circ h^{(1)}_{w_1} \cdot \frac{\partial}{\partial x_i}(h^{(d-1)}_{w_{d-1}} \circ \ldots \circ h^{(1)}_{w_1}) \\ &= ((\underbrace{\sigma(\sigma(\ldots(\sigma(\cdot)))))^2}_{d \text{ times}}\ e^{-w_d^T x}\ w_d) \cdot (\underbrace{\sigma(\sigma(\ldots\sigma(\cdot))))^2}_{d-1 \text{ times}} e^{-w_{d-1}^T x}\ w_{d-1}) \cdot \ldots \end{aligned} \tag{4.15}$$

As one can see, $\frac{\partial}{\partial x_i} h$ includes the derivative of σ and thus the evaluation of σ and the exponential factor of equation (4.14) multiple times. The effect is summarized among others in [41, Sec. 2], [55, Sec. 2]: due to the low range of $\frac{d}{ds}\sigma$ of $(0, 0.25]$ and the exponentials e^{-s}, gradients of lower[17] layers of h tend to zero. Conversly,

[17] A layer within a chain of layers is called *lower* in reference to its closeness to the end of the chain.

with negative input values, $e^{-w^T s}$ causes the gradients to increase dramatically. These two problems are also known as the *Exploding Gradients Problem* and the *Vanishing Gradients Problem*. The latter was first described by [7] for another type of neural networks[18]. It becomes clear from the descriptions of optimization algorithms in Section 4.2.1, that this is a major drawback as gradients directly determine the magnitude of updates of a neural network's weights. Thus, using the logistic function for activation in multiple layers could hamper the learning process. This is especially the case for $\lim_{x\to\infty} \sigma(x)$ and hence, large (initial) values for a neuron's weights *saturate* σ and smaller gradients dramatically [32, Sec. 4.4].

Remark 24 One may note, that

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^x(1 - e^{-2x})}{e^x(1 + e^{-2x})} = \frac{2 - (1 + e^{-2x})}{1 + e^{-2x}} = 2\sigma(2x) - 1.$$

Thus, the Tangens Hyperbolicus suffers from vanishing gradients and saturation problems as well, but in contrast to the logistic function, it is zero-centered as

$$tanh(x) = 0 \Leftrightarrow e^x - e^{-x} \Leftrightarrow x = -x \Leftrightarrow x = 0.$$

This property is desirable as it causes the image to have a mean close to zero, which has been found to support the learning and training process of the neural network: a mean value which is large and positive (negative) causes also gradients to have largely positive (negative) values and thus the weights are largely increased (decreased) all together so that multiple updates have to be carried out to both, lower and higher weights [19][32, Sec. 4.3, 4.4]. Therefore, one should prefer *tanh* over the logistic function.

The problems with sigmoidal functions described above have been overcome with the invention of an activation function called *Rectified Linear Unit (ReLU)* defined as $relu(x) := \max(0, x)$[20]. Intentionally ignoring that ReLU is not differentiable at zero, it is proposed along with

$$\frac{d}{dx}\, relu = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases} =: I_{>0}$$

as its derivative [46].

[18] So-called *Recurrent Neural Networks (RNNs)*.

[19] Which leads to a zigzagging in updates.

[20] Precisely spoken, a Rectified Linear Unit is a *neuron* employing the function $\max(0, \cdot)$ as activation function. In contrast, ReLU is commonly used to refer to the activation function itself (which is also done in this work).

So, analogously to Equation (4.14), the gradient of a ReLU neuron has components of the form

$$\frac{\partial}{\partial x_i} h_w = (\frac{\partial}{\partial x_i} relu)(w^T x) \cdot (\frac{\partial}{\partial x_i} w^T x) = I_{>0}(w^T x) \cdot w_i$$

and hence rewriting equation (4.15) gives

$$\frac{\partial}{\partial x_i} h = I_{>0}(\underbrace{relu(...(relu(\cdot)))}_{d-1 \text{ times}}) w_d \cdot I_{>0}(\underbrace{relu(...(relu(\cdot)))}_{d-2 \text{ times}}) w_{d-1} \cdot \ldots \tag{4.16}$$

for the elements of the gradient of a chain of neurons with ReLU activation. This has not only a high ease of computation, it also preserves the gradients of previous layers while they are "flowing" back through the function chain, which has been found to accelerate the optimization process of a neural network tremendously. The impact has been shown clearly in the *AlexNet* [31], the breakthrough of a so-called *Convolutional Neural Network (CNN)* (This network type will be described later in this chapter, namely in Section 4.6).
It has been shown, that also networks employing ReLU are universal function approximators [40]. However, as it becomes clear from equation (4.16), once a neuron's output becomes non-positive due to its weights, the gradient of this neuron becomes zero and thus the unit's weights will not be updated anymore (by a gradient-based procedure) causing the *Dying ReLU Problem* [55, Sec. 2], [41, Sec. 2].

This can be overcome by allowing a small gradient for negative input values instead of sharply thresholding at zero: the *Leaky Rectified Linear Unit (LReLU)* or *LeakyReLU* is defined by [41] as

$$lrelu(x) := \begin{cases} x, & x > 0 \\ 0.01x, & x \leq 0 \end{cases} \quad \text{with} \quad \frac{d}{dx}\, lrelu = \begin{cases} 1, & x > 0 \\ 0.01, & x \leq 0 \end{cases}.$$

LReLU has already succeeded multiple times in related work [17, 27, 72, 49, 36, 12, 59, 28] and thus is also preferred in this project. Some other less common variants of ReLU are summarized and compared in [55], yet not part of this work due to their rare usage in this field.

4.3.3 Dropout and Batch Normalization

Neural Networks are often prawn to overfitting described earlier in Section 4.1. A broad variety of countermeasures has been proposed. One of the simplest but yet most effective ones is to randomly "disable" single neurons (and thus gradient-based updates on them) by multiplying their output with a variable which is drawn from a Bernoulli distribution every time data is fed to the network. This procedure is called *Dropout* [67]:

Definition 25 (Dropout Layer)
A *Dropout Layer* of a neural network, or simply *Dropout*, is a function

$$f : \mathbb{R}^n \to \mathbb{R}^n, \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \mapsto \begin{bmatrix} x_1 s_1 \\ \vdots \\ x_n s_n \end{bmatrix} \quad \text{with} \quad s_i \sim \text{Bernoulli}(1-p)$$

where $p \in [0, 1)$ is the *Dropout Rate*, i.e. that probability that a neuron "drops out of the network" as its output is multiplied by zero. □

Clearly, this function is differentiable with respect to x, i.e. it is compatible with backpropagation and it can be inserted after every kind of neural network layer. Its biggest advantage is its computational inexpensiveness. The motivation behind dropout is to simulate an ensemble of slightly different neural network architectures (Precisely, 2^m variants for a network with m units.) with shared weights trained simultaneously which is well-known to improve learning results [67]. The authors also extensively describe motivation from biology and genetics.

Note that the dropout rate above is a hyperparameter which has to be tuned carefully: a high value for p will reduce the representational capacity of the network at each run too much and therefore hamper training. On the other hand, choosing p too low might not tackle the problem of overfitting sufficiently well. A common setting is $p = 0.5$. Usually, dropout is disabled at test time as it is not functionally needed anymorew at this stage and would turn the trained model into an non-deterministic algorithm. (To ensure that the actual output of each neuron in this situation does not differ from the expectation value of its output during training, in this case the output of each neuron is multiplied with its associated keep probability $p-1$.) While non-determinism after training is clearly not desirable for e.g. classifiers, so-called generative adversarial networks as introduced later in Section 4.4 rely on randomness at production time and can utilize dropout as a source of noise eliminating the need of feeding a random "seed" [27].

Additionally, training can be improved by linearly transforming the input data of a neural network to have a mean of zero and a standard deviation of one [32, 76], called *Normalization*. This also applies to every sub-network, i.e. to every single layer. Hence, [26] introduces a mechanism to apply this normalization not only to the network's input, but also between layers. While ideally mean and standard deviation of the output of each layer should be computed after processing the whole training data, this is mostly impracticable and thus, in minibatch learning (see the previous Section 4.2.2) those statistics are computed over one batch of data motivating the term *Batch Normalization*. As an additional simplification, [26] suggests to normalize each feature independently:

Definition 26 (Batch Normalization Layer)
A *Batch Normalization Layer* of a neural network, or simply *Batch Normalization*, with trainable parameters $\gamma, \beta \in \mathbb{R}^n$ is a function

$$f : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n, \left(\left[x_i\right]_{i\in\{0,..,n\}}, \gamma, \beta \right) \mapsto \left[\gamma_i \, \frac{x_i - \mathbb{E}[x_i]}{\sqrt{\mathrm{Var}[x_i] + \epsilon}} + \beta_i \right]_{i\in\{0,..,n\}}$$

where exception and variance are estimated over a batch B of samples separately for each feature by

$$\mathbb{E}\left[x_i\right] \approx \mu_{B,i} := \frac{1}{|B|} \sum_{x\in B} x_i \quad \text{and} \quad \mathrm{Var}\left[x_i\right] \approx \sigma^2_{B,i} := \frac{1}{|B|} \sum_{x\in B} (x_i - \mu_{B,i})^2$$

and ϵ is negligibly small constant for numerical stability. □

This operation indeed shifts the mean to zero as for a set A and $\widetilde{A} := \{\frac{a-\mu_A}{\sigma_A} | a \in A\}$ it is

$$\mu_{\widetilde{A}} = \frac{1}{|A|} \sum_{a\in A} \frac{a - \mu_A}{\sigma_A} = \frac{1}{\sigma_A} \left(\frac{1}{|A|} \left(\sum_{a\in A} a - \sum_{a\in A} \mu_A \right) \right) = 0 \tag{4.17}$$

and

$$\sigma_{\widetilde{A}} = \sqrt{\frac{1}{|A|} \sum_{a\in A} \left(\frac{a - \mu_A}{\sigma_A} - \mu_{\widetilde{A}} \right)^2} \overset{(4.17)}{=} \frac{1}{\sigma_A} \sqrt{\frac{1}{|A|} \sum_{a\in A} (a - \mu_A)^2} = 1.$$

A few remarks on the practical use of the definition above: the trainable parameters γ and β introduced above restore the full representational power of the network threatened by the normalization [26]: for instance, letting $\gamma_i = \sqrt{\mathrm{Var}\left[x_i\right]}$ and $\beta_i = \mathbb{E}\left[x_i\right]$, the network can still represent the identity transform for feature i. The function above is differentiable with respect to x, γ and β enabling the application of gradient descent algorithms. Usually, batch normalization is applied even before the activation function of a layer [26, 19].

Using batch normalization enforces the same data distribution in the input of each layer and through this makes the network less sensitive to the choice of the learning rate and initial weights, which otherwise can cause heavy changes of those distributions as effects are amplified by backpropagation through various layers. Further, in experiments it turned to out to act as a regularizer that helps to tackle overfitting just like dropout [26].

4.4 Generative Adversarial Networks

Neural networks as described above approximate input-output mappings where the output is some information extracted from the input data. This typically tackles classification and regression problems. In contrast, so-called *Generative Models* try to represent a real world data distribution such that "new samples" can be drawn from it. One class of generative models, which has turned out to be highly successful in tasks such as image generation [59, 27, 28, 20] relies on the game-theoretic concept of a *zero-sum game*, a special instance of a *strategic game*, between two parties competing via individual actions:

The following formulations are taken from [53, Ch. 2].

Definition 27 (Strategic Game)
A *Strategic Game* $(N, (A_i), R)$ consists of

- a finite non-empty set $N \subset \mathbb{N}$ of players
- finite non-empty sets $(A_i)_{i \in N}$ of available actions for each player $i \in N$
- an order $\geq_i$ on $\times_{j \in N} A_j$ as a *Preference Relation* for each player $i \in N$, notate $R := (\geq_i)_{i \in N}$.[21] □

Remark 28 (Utility Function) The preference relations of a strategic game are mostly implemented as utility functions $U := (u_i)_{i \in N}$ with $u_i : A \to \mathbb{R}, i \in N$ via

$$a \geq_i b \quad \Leftrightarrow \quad u_i(a) \geq u_i(b), \qquad a, b \in A.$$

In this case, denote the strategic game as $(N, (A_i), U)$.

Strategic games can (but not necessarily have to) bear an equilibrium in which no player prefers another action over the one he has taken already:

Definition 29 (Nash Equilibrium)
A strategic game $(N, (A_i), R)$ has a *Nash Equilibrium* in

$$a^* := (a_0^*, ..., a_N^*) \in \times_{i \in N} A_i$$

if

$$\forall i \in N : \quad \forall a_i \in A_i : \; a^* \geq_i (a_0^*, ..., a_{i-1}^*, a_i, a_{i+1}^*, ..., a_N)$$

[21] Defining the relation $\geq_i$ over all constellations of actions of all players instead of restricting to A_i distinguishes a strategic game from a decision problem [53, Sec. 2.1].

i.e. if every player i prefers its action a_i^* over any other of its available actions with respect to the actions of the other players in a^* or – loosely spoken – if no player can make improvements considering the situation of the other players. □

A special form of strategic games are zero-sum games where two players are direct opponents in the sense that an action's utility value for one player is the inverse of its utility value for the other:

Definition 30 (Zero-sum Game)
A strategic game $(N, (A_i), R)$ of two players $N = \{n_1, n_2\}$ is called *Zero-sum Game* if with $A := \times_{j \in N} A_j$ it holds

$$\forall a, b \in A: \quad a \geq_{n_1} b \quad \Leftrightarrow \quad b \geq_{n_2} a. \tag{4.18}$$

Condition (4.18) can also be expressed wLoG via utility functions u_{n_1}, u_{n_2} as

$$u_{n_1} = -u_{n_2}, \quad \text{i.e.} \quad u_{n_1} + u_{n_2} = 0$$

motivating the term "zero-sum". The game is called *finite*, if all A_i are finite. □

Definition 31 (Maxminimizer)
In a zero-sum game $(\{1, 2\}, (A_i), (u_i))$, an action $x^* \in A_1$ is a *Maxminimizer* for player 1 (and analogously for player 2), in case

$$x^* = \arg\max_{x \in A_1} \min_{y \in A_2} u_1(x, y).$$

□

According to [53, Sec. 2.5], it can be shown, that maxminimizers equal Nash equilibria:

Theorem 32
A zero-sum game $(\{1, 2\}, (A_i), (u_i))$ *has a Nash equilibrium in* (x^*, y^*) *iff* x^* *and* y^* *are maxminimizers for player 1, player 2 and additionally*

$$\max_{x \in A_1} \min_{y \in A_2} u_1(x, y) = \min_{y \in A_2} \max_{x \in A_1} u_1(x, y) \tag{4.19}$$

□

Remark 33 For finite zero-sum games, condition (4.19) is always fulfilled, see [47, 50] and thus having maxminimizers for each player is sufficient for the existence of a Nash equilibrium.

Remark 34 For a zero-sum game $(\{1,2\},(A_i),(u_i))$, it holds

$$\begin{aligned}\max_{y\in A_2}\min_{x\in A_1} u_2(x,y) &= \max_{y\in A_2}\min_{x\in A_1}(-u_1(x,y))\\ &= \max_{y\in A_2}\left(-\max_{x\in A_1} u_1(x,y)\right) = -\min_{y\in A_2}\max_{x\in A_1} u_1(x,y).\end{aligned}$$

PROOF (THEOREM 32) "$\Longrightarrow$": Let (x^*,y^*) be a Nash equilibrium. Then

$$\begin{aligned}&\forall y\in A_2:\ u_2(x^*,y^*) \ \geq\ u_2(x^*,y)\\ \overset{u_2=-u_1}{\Longrightarrow}\ &\forall y\in A_2:\ u_1(x^*,y^*) \ \leq\ u_1(x^*,y)\\ \Longrightarrow\ & u_1(x^*,y^*) = \min_{y\in A_2} u_1(x^*,y) \ \leq\ \max_{x\in A_1}\min_{y\in A_2} u_1(x,y).\end{aligned}$$

Also it holds because of (x^*,y^*) being a Nash equilibrium that

$$\begin{aligned}&\forall x\in A_1:\ u_1(x^*,y^*) \ \geq\ u_1(x,y^*)\\ \Longrightarrow\ & u_1(x^*,y^*) = \max_{x\in A_1} u_1(x,y^*) \ \geq\ \max_{x\in A_1}\min_{y\in A_2} u_1(x,y).\end{aligned}$$

Hence,

$$u_1(x^*,y^*) = \max_{x\in A_1}\min_{y\in A_2} u_1(x,y)$$

i.e. x^* is a maxminimizer for player 1. The corresponding result for y^* follows analogously. Therefore, it holds additionally

$$\begin{aligned}\max_{x\in A_1}\min_{y\in A_2} u_1(x,y) &= u_1(x^*,y^*) = -u_2(x^*,y^*)\\ &= -\max_{y\in A_2}\min_{x\in A_1} u_2(x,y) \overset{\text{Remark 34}}{=} \min_{y\in A_2}\max_{x\in A_1} u_1(x,y).\end{aligned}$$

"$\Longleftarrow$": Let $v^* = \max_{x\in A_1}\min_{y\in A_2} u_1(x,y) = \min_{y\in A_2}\max_{x\in A_1} u_1(x,y)$ and let x^*,y^* be maxminimizers for player 1 and 2. Because of the latter, it is

$$\begin{aligned}\forall(x,y)\in a_1\times A_2:\quad & u_1(x^*,y)\geq v^*\\ \wedge\quad & u_2(x,y^*)\geq \max_{y\in A_2}\min_{x\in A_1} u_2(x,y) \overset{\text{Remark 34}}{=} -v^*\\ \Longrightarrow\ & u_1(x^*,y^*)\geq v^* \quad\wedge\quad u_2(x^*,y^*)\geq v^*\\ \overset{u_1=-u_2}{\Longrightarrow}\ & u_1(x^*,y^*)\geq v^*\geq u_1(x^*,y^*)\\ \Longrightarrow\ & u_1(x^*,y^*)=v^* \quad\wedge\quad u_2(x^*,y^*)=-v^*\end{aligned}$$

Thus, according to the definition of v^*, player 1 prefers none of its actions over x^* in $(\,\cdot\,,y^*)$ and accordingly does player 2. So, (x^*,y^*) is indeed a Nash equilibrium of this strategic game. ∎

Remark 35 From Theorem 32 and its proof, it can be seen that the utility value of a Nash equilibrium $u_1(x^*, y^*) = \max_{x \in A_1} \min_{y \in A_2} u_1(x, y)$ does not depend on x^* or y^*. Hence, all Nash equilibria of a zero-sum game yield the same payoff.

Now above formulations can be used to define an adversarial setting between two independent neural networks from which one tries to approximate a probability distribution of real world data, the generator network, while the other one, called discriminator, strives for distinguishing the generator from the true data distribution by mapping a sample to the probability that this sample is from the real distribution [20, 19]:

Definition 36 (Generative Adversarial Network)
A *Generative Adversarial Network (GAN)* is a zero-sum game

$$(\{G, D\}, \{A_G, A_D\}, \{u_D, u_G\})$$

with

- two neural networks $G : (Z \times \mathbb{W}_G) \to X$ and $D : (X \times \mathbb{W}_D) \to [0, 1]$ with weights $W_G \in \mathbb{W}_G, W_D \in \mathbb{W}_D$ [22] as players.
- action sets $A_G := \mathbb{W}_G$, $A_D := \mathbb{W}_D$
- utility functions

$$\begin{aligned} u_D(W_D, W_G) :=& \mathbb{E}_{x \sim p_{\text{data}}(x)} \left[\ln(D(x, W_D))\right] \\ &+ \mathbb{E}_{z \sim p_z(z)} \left[\ln(1 - D(G(z, W_G)W_D))\right] \end{aligned} \tag{4.20}$$

  and $u_G := -u_D$

where p_{data} is the density describing the probability distribution from which the training data points $x \in \mathbb{R}^{d_x} =: X$ are drawn and p_z the distribution of a random noise variable $z \in \mathbb{R}^{d_z} =: Z$ serving as "seed" for G to generate data. □

The intuition behind the utility functions is the following: the discriminator's quality is determined by its ability to distinguish samples from the generator and the true data distribution, i.e. its utility should be high if it outputs 1 for real samples (i.e. $\mathbb{E}_{x \sim p_{\text{data}}(x)} \left[\ln(D(x, W_D))\right]$ is high) and conversely returns 0 for fake samples (i.e. $\mathbb{E}_{z \sim p_z(z)} \left[\ln(1 - D(G(z, W_G), W_D))\right]$ is high, too). The worse the generator can do this distinction, the better is the expected quality of generator outputs, hence one

[22] The expressions $\mathbb{W}_G, \mathbb{W}_D$ shall – in a simple way – represent the spaces where the weights of G, D can be drawn from. Those spaces consist of all families of real-valued matrices matching the shapes required for the weights of the layers of G or respectively of D.

defines $u_G := -u_D$. Above in equation (4.20), the logarithm is applied "because of its interpretation as the likelihood [... but] still makes intuitive sense if we replace [it] by any monotone function $\phi : [0, 1] \to \mathbb{R}$" [4] or even the identity [3].

As [51] states, "GANs are represented using floating point numbers, of which, for a given setup [i.e. a certain GAN implementation], there is only a finite (albeit large) number [of actions (weight configurations) in practice]". This allows considering GANs as finite zero-sum games and in particular omitting condition (4.19) when applying Theorem 32 according to Remark 33: therefore, a Nash equilibrium of this game exists and can be found in maxminimizers for D and G, in particular involving the generator action

$$W_G^* = \arg\max_{W_G} \min_{W_D} u_G(W_D, W_G) \overset{\textit{Remark 34}}{=} \arg\min_{W_G} \max_{W_D} u_D(W_D, W_G)$$

As stated in Remark 35 it suffices to find any solution to the above problem in order to achieve the highest possible utility for G. The equation above is re-written as arg min to match the specifications of the whole optimization process of neural networks which has been defined as minimization earlier in Section 4.2.1. Both neural networks are trained simultaneously according to the maxminimization above: while the discriminator is trained to directly maximize u_D (implemented as minimizing $-u_D$), the objective of the generator is to minimize u_D. Note that the first addend of u_D is constant for all W_G and that W_D are not part of G's optimization process. Hence, one can define the loss functions

$$\begin{aligned} L_G(W_G) &:= \mathbb{E}_{z \sim p_z(z)} \left[\ln(1 - D(G(z, W_G), W_D))\right] \\ L_D(W_D) &:= -u_D(W_D, W_G) \end{aligned}$$

where W_D are treated as fixed in L_G and vice versa. The distinction between the utility functions of the zero-sum game and the loss functions for optimization here is made as in the following GAN variants are introduced that optimize additional measures (that are not part of the zero-sum game) *in parallel* to the adversarial loss that *arises from* the utility function.

Above formulations incorporate that after an ideal training process the generator produces "fake data" so well, that the discriminator cannot distinguish them from the real data anymore and outputs a probability of $\sim 50\%$ for each sample to be real. At this state, the discriminator does not provide helpful "feedback" for the generator anymore, i.e. both G and D cannot improve any longer, a Nash equilibrium has been found and the discriminator can be discarded while the generator may be applied in production [20, 19].

Now it shall be verified, that indeed a Nash equilibrium is reached when the probability distribution of the data produced by the generator, p_g, equals the true data distribution p_{data} following [20]:

Theorem 37 (Goodfellow, [20])
The best possible discriminator of a GAN with a fixed generator is

$$D(x, W_D^*) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x; W_G)}$$

where p_g denotes the distribution produced by the generator with weights W_G. □

PROOF Shorthand $D_{W_D} := D(\cdot, W_D)$ and for the generator G analogously, then re-write the function (4.20) as

$$\begin{aligned} u_D(W_D, W_G) &= \int p_{\text{data}}(x) \ln(D_{W_D}(x))\ dx + \int p_z(z) \ln(1 - D_{W_D}(G_{W_G}(z)))\ dz \\ &= \int p_{\text{data}}(x) \ln(D_{W_D}(x)) + p_g(x; W_G) \ln(1 - D_{W_D}(x))\ dx. \end{aligned} \tag{4.21}$$

For any $(a, b) \in \mathbb{R}_+ \setminus \{(0, 0)\}$ and function $f : s \mapsto a \ln(s) + b \ln(1 - s)$it holds

$$\begin{aligned} & 0 = \frac{df}{ds} = \frac{a}{s} - \frac{b}{1-s} = \frac{a - s(a+b)}{s - s^2} \\ \iff \quad & 0 = a - x(a+b) \quad \iff \quad x = \frac{a}{a+b}. \end{aligned}$$

and

$$\frac{df}{ds^2} = -\frac{a}{s^2} - \frac{b}{(1-s)^2} < 0$$

and hence f has a unique maximum in $\frac{a}{a+b}$. As "the discriminator does not need to be defined outside of $\text{supp}(p_{\text{data}}) \cup \text{supp}(p_g)$" [20] where it might be $p_{\text{data}}(x) = 0 = p_g(x; W_G)$, this result can be directly applied to (4.21) yielding the claim of the theorem. ■

Hence, a "virtual" training criterion for the generator based on the best possible adversary can be formulated as

$$\begin{aligned} L_G^*(W_G) &:= \max_{W_D} u_D(W_D, W_G) \\ &= \mathbb{E}_{x \sim p_{\text{data}}(x)} \left[\ln(D(x, W_D^*))\right] + \mathbb{E}_{z \sim p_z(z)} \left[\ln(1 - D(G(z, W_G), W_D^*))\right] \\ &= \mathbb{E}_{x \sim p_{\text{data}}(x)} \left[\ln(D(x, W_D^*))\right] + \mathbb{E}_{x \sim p_g(x; W_G)} \left[\ln(1 - D(x, W_D^*))\right] \\ &\overset{\text{Theorem 37}}{=} \mathbb{E}_{x \sim p_{\text{data}}(x)} \left[\ln \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x; W_G)}\right] + \mathbb{E}_{x \sim p_g(x; W_G)} \left[\ln \frac{p_g(x; W_G)}{p_{\text{data}}(x) + p_g(x; W_G)}\right]. \end{aligned}$$

In preparation for the proof of the following theorem, another similarity measure between probability distributions is established briefly:

Definition 38 (Jensen–Shannon Divergence)
With KLD denoting the Kullback-Leibler divergence as introduced in Definition 11, the *Jensen-Shannon Divergence* of probability distributions p, q is

$$\mathrm{JSD}(p\|q) := \frac{1}{2}\mathrm{KLD}\left(p \,\middle\|\, \frac{p+q}{2}\right) + \frac{1}{2}\mathrm{KLD}\left(q \,\middle\|\, \frac{p+q}{2}\right)$$

. □

Theorem 39 (Goodfellow II, [20])
For a GAN with true data distribution p_{data} and generated distribution p_g from a generator G with weights W_G^*, it holds

$$p_{data} = p_g(\cdot\,; W_G^*) \quad \Longleftrightarrow \quad W_G^* = \arg\min_{W_G} L_G^*$$

and $L_G^*(W_G^*) = -\ln(4)$. □

Proof For the virtual training criterion of the generator as formulated above it holds

$$\begin{aligned}
L_G^*(W_G) &= \mathbb{E}_{x\sim p_{\mathrm{data}}(x)}\left[\ln \frac{p_{\mathrm{data}}(x)}{p_{\mathrm{data}}(x) + p_g(x; W_G)} + \ln 2 - \ln 2\right] \\
&+\mathbb{E}_{x\sim p_g(x;W)}\left[\ln \frac{p_g(x; W_G)}{p_{\mathrm{data}}(x) + p_g(x; W_G)} + \ln 2 - \ln 2\right] \\
&= \mathbb{E}_{x\sim p_{\mathrm{data}}(x)}\left[\ln \frac{p_{\mathrm{data}}(x)}{\frac{1}{2}(p_{\mathrm{data}}(x) + p_g(x; W_G))}\right] \\
&+\mathbb{E}_{x\sim p_g(x;W_G)}\left[\ln \frac{p_g(x; W_G)}{\frac{1}{2}(p_{\mathrm{data}}(x) + p_g(x; W_G))}\right] - \ln 4 \\
&= \mathrm{KLD}\left(p_{\mathrm{data}} \,\middle\|\, \frac{(p_{\mathrm{data}} + p_g)}{2}\right) + \mathrm{KLD}\left(p_g \,\middle\|\, \frac{(p_{\mathrm{data}} + p_g)}{2}\right) - \ln 4 \\
&= 2\ \mathrm{JSD}\left(p_{\mathrm{data}}\ \|p_g\right) - \ln 4.
\end{aligned}$$

As $\mathrm{JSD} \geq 0$ and $\mathrm{JSD}(p\|q) = 0 \iff p = q$, the virtual training criterion L^* has a global minimum with value $-\ln 4$ in a weight configuration W_G^* which results in $p_g = p_{\mathrm{data}}$. ■

In the setting of a GAN as described above, the noise variable z is crucial to introduce randomness to the trained generator and allows it to produce arbitrary output

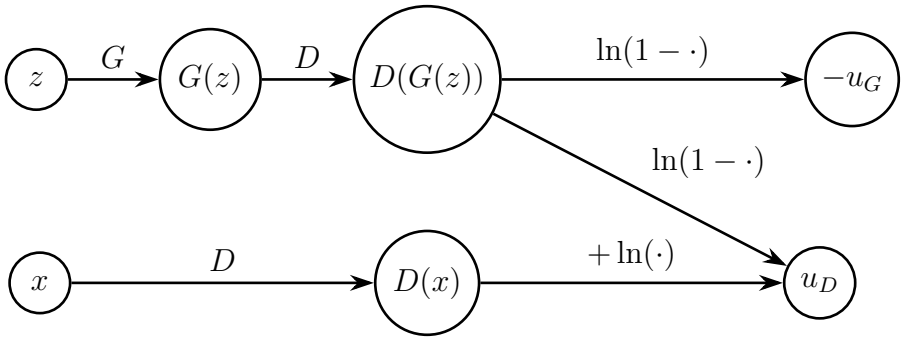


Figure 4.5: The computation of the utility u_G a of the generator network G and the utility u_D of the adversarial discriminator.

samples non-deterministically. To exert some control over the output, for instance to produce images of a certain class, *cGAN* [18] extends a GAN by feeding it side conditions as additional input:

Definition 40 (Conditional GAN)
A GAN $(\{G, D\}, \{A_G, A_D\}, \{u_D, u_G\})$ on weight spaces $\mathbb{W}_G, \mathbb{W}_D$, data space X and noise space Z can be extended to a *Conditional GAN (cGAN)* by adding contextual information $y \in \mathbb{R}^{d_y} =: Y$ drawn from the training data $(x, y) \sim p_{\text{data}}(x, y)$ according to a marginal distribution $y \sim p_{\text{data}}(y)$ such that

$$G : (Z \times Y \times \mathbb{W}_G) \to X \quad \text{and} \quad D : (X \times Y \times \mathbb{W}_D) \to [0, 1].$$

and

$$\begin{aligned} u_D(W_D, W_G) :=& \mathbb{E}_{(x,y)\sim p_{\text{data}}(x,y)} \left[\ln(D(x, y, W_D))\right] \\ &+ \mathbb{E}_{z\sim p_z(z), y\sim p_{\text{data}}(y)} \left[\ln(1 - D(G(z, y, W_G), y, W_D))\right] \end{aligned}$$

with loss functions formulated accordingly. □

It is remarkable, that no restrictions are imposed on the nature of the condition y In particular, y can also be a (re-shaped) pianoroll as introduced at the beginning in Section 3.4.

The generator here represents a conditional distribution $p_g(x|y; W_G)$ which ideally should approximate the joint true distribution $p_{\text{data}}(x, y)$. The training process remains the same as for a regular GAN, while the discriminator is expected to "first acquire a use for the data y" before the generator follows [18]. This also bears the danger of the discriminator improving so quickly compared to the generator that it perfectly distinguishes fake and real for any generator output so that the generator cannot improve anymore, i.e. collapses. One attempt to mitigate this is to slow

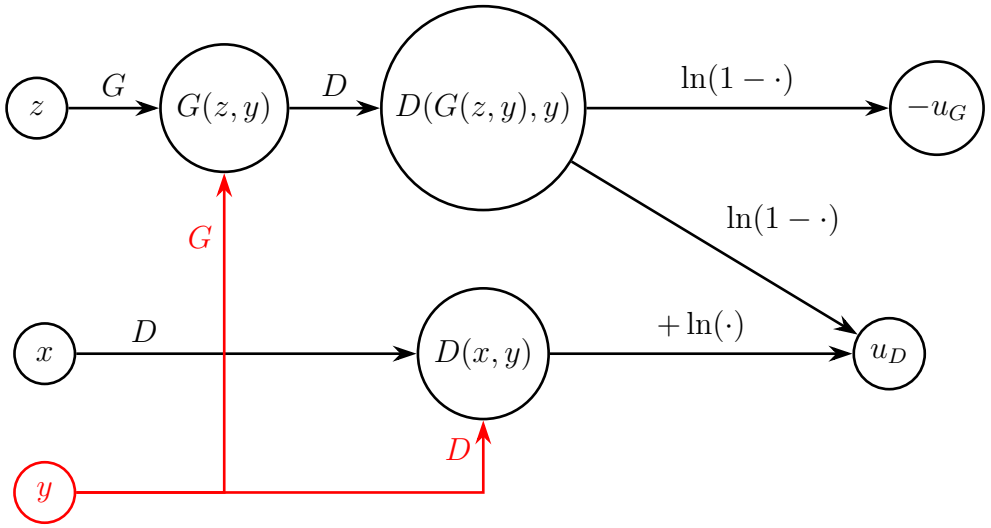


Figure 4.6: The computation of the utility u_G a of the generator network G and the utility u_D of the adversarial discriminator in a conditional GAN with data x, condition y and noise z.

down discriminator training by applying updates only every second epoch of training or simulating this by cutting the discriminator loss (and thus its gradient) in half [27].

It is to mention that there are different approaches on how and at which point to actually incorporate context information y within the neural networks G and D: for instance, the original cGAN [18] uses y from the very top (input) layer of the generator but at the last (output) layer of D. In contrast, the approach of [44] already includes the condition in the first layers of the discriminator, too, encouraging greater influence on the whole neural network.
Another way to incorporate the side information in the discriminator is to additionally task it with reconstructing y from x instead of feeding it directly [49]:

Definition 41 (Auxiliary Classifier GAN)
A cGAN $(\{G, D\}, \{A_G, A_D\}, \{u_D, u_G\})$ as defined above is an *Auxiliary Classifier GAN (AC-GAN)*, if the discriminator in addition to distinguishing real and fake samples is trained to reconstruct a side condition y from data x, i.e.

$$G : (Z \times Y \times \mathbb{W}_G) \to X \quad \text{and} \quad D : (X \times \mathbb{W}_D) \to [0, 1] \times Y.$$

The utility functions from Definition 40 remain unchanged. Letting p_d be the distribution modeled by D and considering the marginal $p_d(y)$ one can define

$$\begin{aligned} L_y(W_D, W_G) := -(&\mathbb{E}_{(x,y)\sim p_{\text{data}}(x,y)} [\ln p_d(y|x; W_D)] \\ &+ \mathbb{E}_{z\sim p_z(z), y\sim p_{\text{data}}(y)} [\ln p_d(y|G(z, y, W_G); W_D)]) \end{aligned}$$

and loss functions

$$L_D(W_D) := -u_D(W_D, W_G) + L_y(W_D, W_G)$$
$$L_G(W_G) := \mathbb{E}_{z\sim p_z(z), y\sim p_{\text{data}}(y)} \left[\ln(1 - D(G(z, y, W_G), y, W_D))\right] + L_y(W_D, W_G).$$

□

According to [49], such an auxiliary classifier cannot only leverage further improvements of a trained network using pre-trained classifiers, but also helps stabilize GAN training. A prominent application of this technique in the field of music is GANsynth [17], where the side information is the pitch of a note for which a waveform should be generated.

In case GANs are applied to generate images or image-like data, it is desirable that the network not only learns the distribution of images as a whole, but instead also local structure. In particular, early attempts of image synthesis [30] could produce detailed images only for a very limited output size. In parallel, systems have been proposed which are dedicated to texture synthesis using so-called *Markovian Fields* [35], i.e. a structure[23] of locally independent random variables. One way to combine the latter with GANs is described by [36, 27]: instead of letting the discriminator of a GAN output a scalar for real/fake classification of the output (image) as a whole, it outputs a matrix where each entry classifies one patch of the input image independently from the others, i.e. classifying local structure:

Definition 42 (Patch-based GAN)
A *Patch-based Generative Adversarial Network (PatchGAN)* with $n \times m$ *Patches* is a GAN with a discriminator $D : (X \times \mathbb{W}_D) \to [0, 1]^{n\times m}$ and utility function

$$\begin{aligned} u_D(W_D, W_G) &:= \mathbb{E}_{x\sim p_{\text{data}}(x)} \left[\ln(D(x, W_D))\right] + \mathbb{E}_{z\sim p_z(z)} \left[\ln(1 - D(G(z, W_G), W_D))\right] \\ &= \mathbb{E}_{x\sim p_{\text{data}}(x)} \left[\ln \prod_{i,j}^{m} D_{i,j}(x, W_D)\right] + \mathbb{E}_{z\sim p_z(z)} \left[\ln \prod_{i,j}^{m} (1 - D_{i,j}(G(z, W_G), W_D))\right] \\ &= \sum_{i,j}^{m} \mathbb{E}_{x\sim p_{\text{data}}(x)} \left[\ln D_{i,j}(x, W_D)\right] + \mathbb{E}_{z\sim p_z(z)} \left[\ln D_{i,j}(G(z, W_G), W_D)\right] \end{aligned}$$

where the outputs $D_{i,j}$ for patches (i, j) are assumed to be iid. As in Definition 36, it is still $u_G := -u_D$ with according loss functions. □

[23] a graph or matrix

The underlying consideration is that values (pixels) outside one patch are independent. Therefore, the loss of a generator in terms of a discriminator outputting such a "classification heat map" can be seen as textural loss only that should be combined with other measures (such as an L_p-metric between output and target) in order to penalize also loss in global structure as it is done in [27]. Note that n and m are hyperparameters which heavily affect the details in the images produced by the GAN: as [27] reports, a low number of accordingly large patches produces sharp and detailed results but might cause tiling artifacts while a high number of therefore smaller patches appears to cause blurriness.

4.5 Autoencoder Architectures

In the previous Section 4.4, while introducing GANs no assumption has been made on the nature of the generator except that is has been required to be a neural network. While "classic" neural networks have been typically applied to classification and regression problems, one special type of neural network architecture which can achieve this efficiently are so-called *Autoencoder*s [19, Ch. 14]:

Definition 43 (Autoencoder)
An *Autoencoder* $a : (\mathbb{R}^{d_{\text{data}}} \times \mathbb{W}_f \times \mathbb{W}_g) \to \mathbb{R}^{d_{\text{data}}}$ with encoding size d_h accepting input of dimension d_{data} is a neural network consisting of two neural networks

$$f : (\mathbb{R}^{d_{\text{data}}} \times \mathbb{W}_f) \to \mathbb{R}^{d_h} \quad \text{and} \quad g : (\mathbb{R}^{d_h} \times \mathbb{W}_g) \to \mathbb{R}^{d_{\text{data}}}$$

combined as $a := g \circ f$ where $\mathbb{W}_f, \mathbb{W}_g$ are again spaces of weights. In case $d_h << d_{\text{data}}$, a is called an *Undercomplete Autoencoder*.[24] □

The autoencoder is trained to approximate an identity mapping on the data space. Then designing the autoencoder as undercomplete, i.e. making the encoding $h := f_{W_f}(x)$ a bottleneck, "forces" generalization by learning a lower-dimensional representation of the input. From a stochastic point of view, encoder and decoder provide conditional distributions $p_f(h|x; W_f)$ and $p_g(x|h; W_g)$ and their common objective is the negative log-likelihood $-\ln p_g(x|h; W_g)$ [19, Sec. 14.4].

In previous applications one was mainly interested in exploiting just the compression property and using encoder and decoder separately in production [24]. Another way is to use the network as a whole is to task it with reconstructing the original data from a noisy version exploiting the abstraction capability of the autoencoder enforced by its bottleneck [70]:

[24] As is this is the standard case in most applications [19], this work will refer to them simply as *autoencoders*, too.

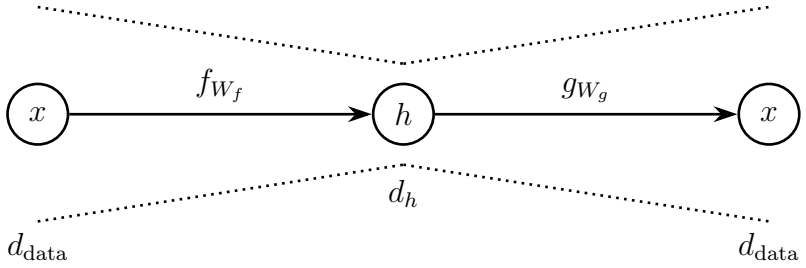


Figure 4.7: An (undercomplete) autoencoder with $h := f_{W_f}(x)$ as a low-dimensional bottleneck between encoder f_{W_f} and decoder g_{W_g}, where the latter tries to reconstruct $x \in \mathbb{R}^{d_{\text{data}}}$ from encoding $h \in \mathbb{R}^{d_h}$.

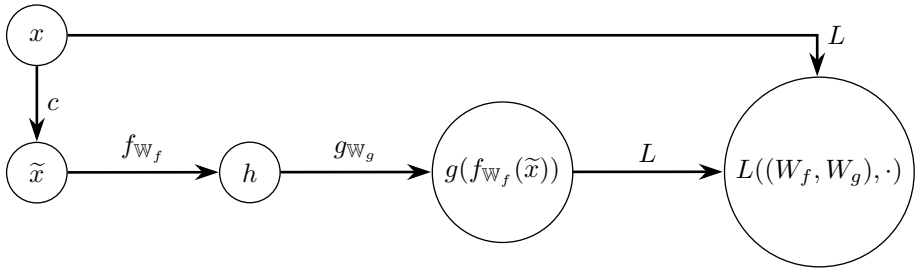


Figure 4.8: A denoising auto-encoder consisting of an encoder f and a decoder g being fed data $x \in \mathbb{R}^{d_{\text{data}}}$ corrupted by process $c : \mathbb{R}^{d_{\text{data}}} \to \mathbb{R}^{d_{\text{data}}}$ including computation of the loss L between original data x and its reconstruction.

Definition 44 (Denoising Autoencoder)
A *Denoising Autoencoder (DAE)* is an autoencoder a with encoder f and decoder g, that given a training data distribution represented by $x \sim p_{\text{data}}(x)$ and a corruption process of this data modeled by a conditional distribution $\widetilde{x} \sim p_c(\widetilde{x}|x)$ has the learning objective of minimizing

$$L((W_f, W_g), \mathbb{T}) := -\mathbb{E}_{x \sim p_{\text{data}}(x)} \left[\mathbb{E}_{\widetilde{x} \sim p_c(\widetilde{x}|x)} \left[\ln p_g(x | f_{W_f}(\widetilde{x}); W_g) \right] \right]. \qquad \square$$

DAEs have been successfully applied to lots of problems including for instance speech enhancement [39] and it has been formally shown by [2, 8], that the denoising training forces the encoder and decoder network to implicitly learn the structure of the true data distribution. By interpreting the original data as "noisy" version of some high-dimensional information semantically and structurally related to this data, one can apply those results for DAEs and use autoencoders to model almost any kind of high-dimensional input-to-output mapping such as image-to-segmentation-map [5] provided that the mentioned interpretation is justified. Hence, a DAE architecture is well-suited for being used as the generator of a GAN performing image-to-image translation.

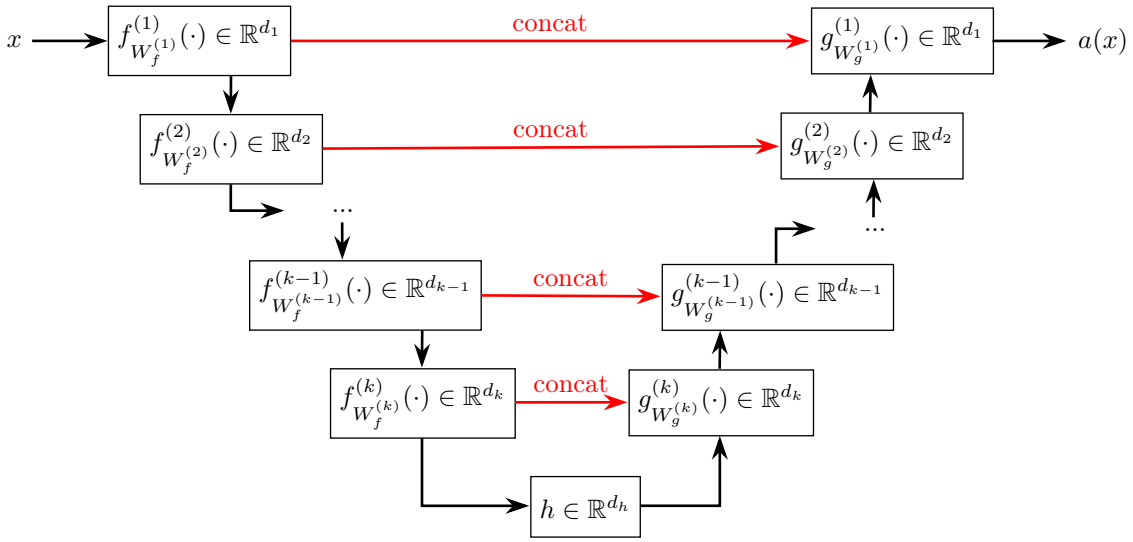


Figure 4.9: An autoencoder a with encoder f and decoder g following a U-Net structure [61] with skip connection (red). The component functions $f^{(1)}, ..., f^{(k)}, g^{(1)}, ..., g^{(k)}$ are the layers of the neural networks f and g where the all pairs $f^{(i)}, g^{(i)}$ have matching output dimensions.

For tasks like the latter, where image-like data containing spatial information is processed, autoencoders largely benefit from utilizing convolutional layers as introduced in Section 4.6. In this special case, it has been found to be beneficial to design the autoencoder in a way that not only the decoder's architecture mirrors the one of the encoder exactly, but also to introduce so-called *Skip Connections* by concatenating the output of each encoder layer to the one of the corresponding layer in the decoder [61]. As [72] mentions, this is especially useful to tackle the problem of vanishing gradients in deep neural networks discussed earlier in Section 4.3, as during backpropagation encoder gradients are allowed to "bypass" the decoder. Such an architecture as depicted in Figure 4.9 has been named *U-Net* and has been very successfully applied not only by the original authors [61], but also for score-to-audio mappings [72] and even in conjunction with a cGAN on image-to-image translation [27].

4.6 Up- and Downsampling through Convolution

This section loosely follows [19, Ch. 9]

Lots of common problems for neural networks are defined over high-dimensional image-like data, i.e. data where each sample is a large multi-dimensional matrix, in

which the elements (e.g. pixels of an image) are not independent but instead there is local structure and the spatial position of each element bears semantic information. Attempts to process such data with fully-connected layers as introduced in Section 4.3 are likely to fail because of the following key issues:

Expensiveness As it becomes clear from Definition 21, a fully-connected layer with m neurons requires a weight for each of its pairwise connections to its input data. Assuming for example a medium sized RGB image of shape $512 \times 512 \times 3$, this results in more than $700.000 \times m$ weights for the input layer alone making it very expensive in terms of required computation time and memory.

Redundancy By design a fully-connected layer does not respect spatial information of the elements of its input data, i.e. it primarily extracts *global* patterns from the data. To exploit small *local* structures such as edges in an image, the weight matrix has to reflect this pattern at every possible location (i.e. sub-matrix) making the weight information highly redundant.

Translation variance Input data with slight modifications that do not affect semantics such as shifting an image by one pixel activates different neurons of a fully-connected layer and thus its output can change heavily under translation.

Fixed Shape Once a fully-connected layer is defined, every data it is fed has to match the size of the layer. From a pratical point of view, this restricts the usage of a trained model in production as for instance an image classifier cannot be directly applied to images larger than those it has been trained on.

These problems can be overcome by using not a fully-connected layer but instead (a stack of) very small matrices, so-called *Filters*[25], each one capturing a small pattern, which are multiplied with the input data in a sliding window and reduced by summation:

Definition 45 (Convolutional Layer)
Let $H \in \mathbb{R}^{m \times u}$ be a stack of m 1D-filters $H_k \in \mathbb{R}^u$ of size u. Then a 1D-convolutional layer with activation function σ, bias $b \in \mathbb{R}^m$ and filters H applied with stride s to inputs $x \in \mathbb{R}^n$ which are padded with p zeros at every side is a function

$$\begin{aligned} f :& \mathbb{R}^n \times \mathbb{R}^{m \times u} \times \mathbb{N} \times \mathbb{N} \to \mathbb{R}^m \times \mathbb{R}^{\lfloor (n+2p-u)/s+1 \rfloor} \\ & (x, H, s, p) \mapsto \left[\sigma(\widetilde{x} *_s H_k + b_k)\right]_{k \in \{1,..,m\}} \end{aligned}$$

[25] In the literature, also the term *Kernel* is widely-spread.

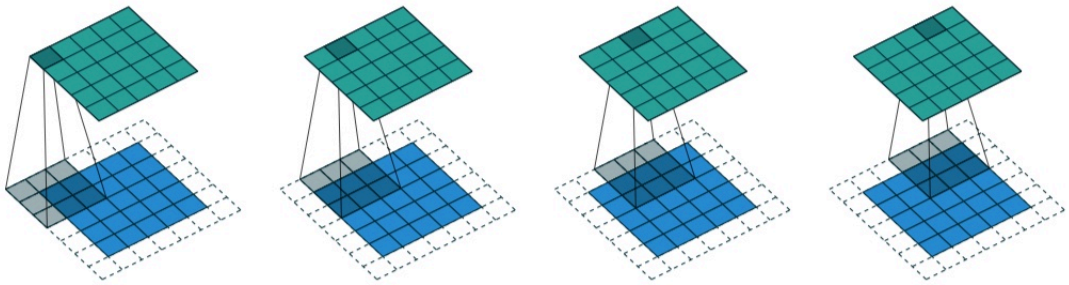


Figure 4.10: A 2D convolution operation on a 5×5 input (blue) with one 3×3 filter (gray), padding $p = 1$ (dashed) and unit stride resulting in an output (green) that matches the input's shape. Taken from [16].

with the padded input

$$\widetilde{x} := \begin{bmatrix} 0_1 & \dots & 0_p & x_1 & \dots & x_n & 0_1 & \dots & 0_p \end{bmatrix}$$

and the operation

$$*_s : \mathbb{R}^a \times \mathbb{R}^b \to \mathbb{R}^{\lfloor (a-b)/s+1 \rfloor}, \quad A * B := \left[\sum_{i=1}^{a} A_{st+i} B_i \right]_{t \in \{0,\dots,\lfloor (a-b)/s \rfloor\}}$$

Reformulations for the *2D-Convolution* are straight forward.[26] □

Note that padding the input with zeros does not affect the result of the convolution operation. The operation $*_s$ defined above is actually not a (discrete strided) convolution in the standard sense, but instead one that flips one of its arguments and is termed *Cross-Correlation*. Nevertheless, literature and in particular machine learning libraries refer to it just as "convolution" [19, 1] and so does this work.
A visualization of the convolution operation for one filter in the 2D case is provided in Figure 4.10. Notice that similar to the number of neurons in a fully-connected layer above the number of filters determines the depth of the output as it can be seen from Figure 4.11. The remaining shape detail of the output is determined by the filter size and the window stride: in the setting of Figure 4.10 with $p = \lfloor \frac{u-1}{2} \rfloor$ and unit stride, the convolution operation preserves the input's shape. Hence, setting $p = \lfloor \frac{u-1}{2} \rfloor$ is often called *SAME Padding*. Another option is to simply drop overflowing values, i.e. setting $p = 0$, resulting in an output size of $m \times \lfloor (n-u)/s + 1 \rfloor$. As this mode does not "artificially" augment the input data, it is widely termed *VALID Padding*.

A convolutional layer indeed solves the problems described at the beginning of this section: the required memory for storing weights and executing operations depends

[26] but are omitted here to avoid confusion by an overflow of notation and index variables.

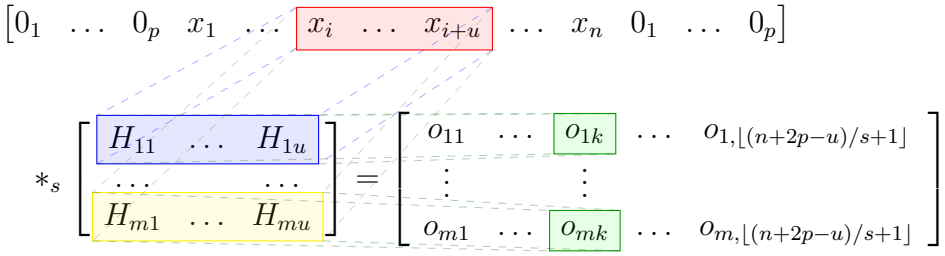


Figure 4.11: Illustration of one step of a 1D-convolution on input data $x \in \mathbb{R}^n$ with a stack $H \in \mathbb{R}^{m \times u}$ of m filters of size u and stride s. The output is denoted as $o \in \mathbb{R}^{m \times \lfloor (n+2p-u)/s+1 \rfloor}$. The graphic does not including activation function or bias terms.

only on the number and size of filters, but not on the input. As filters in most cases are of quadratic shape not exceeding a side length of 5 [19] and used in stacks with a depth commonly ranging from 2^0 to 2^9, they can be efficiently stored and processed.[27] Being independent of the input dimension also enables the application of a (trained) filter to arbitrary inputs. Further, each filter can learn its own local structure and detect it anywhere in the input data as it is slid over. Hence, there is no redundancy by design.

It is remarkable that a convolutional layer is in essence just an "efficient version" of a fully-connected layer and all statements made for fully-connected layers can be directly applied to convolutional layers, too, as the following holds: by re-arranging the filter data, every convolutional layer can be reformed as a functionally equivalent fully-connected layer: for filter H_k, form a matrix $\widetilde{H}_k \in \mathbb{R}^{\lfloor (n+2p-u)/s+1 \rfloor \times (n+2p)}$ with all possible filter positions on the input data and multiply this with the padded input signal [37]. This can be seen in detail in Figure 4.12. The convolutional layers then can be expressed as $\sigma(\widetilde{H}_k \widetilde{x}) + b$, which is exactly the form of a fully-connected layer. The whole procedure naturally generalizes to a stack of filters and to the 2D case. One can clearly see the redundancy and sparsity within $\widetilde{H}_k$ introduced here.

It has been made clear that a convolutional layer with SAME padding and unit stride only alters the input's depth but preserves all other shape details regardless of filter design. This can be changed deliberately in order to reduce the input size while increasing information depth, i.e. to compress the input: using SAME padding and a stride greater than 1 or just no padding will result in *Downsampling* the input

[27] Again, using powers of two for the number of filters in every layer is beneficial for computational efficiency.

1D-convolution with one filter

$$\sigma(\begin{bmatrix}0_1 & \dots & 0_p & x_1 & \dots & x_n & 0_1 & \dots & 0_p\end{bmatrix} * \begin{bmatrix}H_{k1} & \dots & H_{ku}\end{bmatrix} + b) = \begin{bmatrix}o_1 & \dots & o_{\lfloor(n+2p-u)/s+1\rfloor}\end{bmatrix}$$

Equivalent fully-connected layer

$$\sigma\left(\begin{bmatrix} H_{k1} & \dots & H_{ku} & 0 & \dots & 0 \\ \hline \vdots & & & & & \vdots \\ \hline 0 & \dots & 0 & H_{k1} & \dots & H_{ku} \end{bmatrix}^{\lfloor(n+2p-u)/s+1\rfloor\times(n+2p)} \times \begin{bmatrix}0_1\\ \vdots \\ 0_p \\ x_1 \\ \vdots \\ x_n \\ 0_1 \\ \vdots \\ 0_p\end{bmatrix}^{(n+2p)\times 1} + b\right) = \begin{bmatrix}o_1 \\ \vdots \\ o_{\lfloor(n+2p-u)/s+1\rfloor}\end{bmatrix} \tag{4.22}$$

Transposed convolution

$$\sigma\left(\begin{bmatrix} H_{k1} & & 0 \\ \vdots & & \vdots \\ H_{ku} & \dots & 0 \\ 0 & & H_{k1} \\ \vdots & & \vdots \\ 0 & & H_{ku} \end{bmatrix}^{(n+2p)\times\lfloor(n+2p-u)/s+1\rfloor} \times \begin{bmatrix}o_1 \\ \vdots \\ o_{\lfloor(n+2p-u)/s+1\rfloor}\end{bmatrix} + b\right) = \begin{bmatrix}\widetilde{x}_1 \\ \vdots \\ \widetilde{x}_{n+2p}\end{bmatrix}^{(n+2p)\times 1} \tag{4.23}$$

Figure 4.12: Reformulation of a convolutional layer (top) as a fully connected layer (bottom). In the latter, the inputs are multiplied by a matrix with one row for each position of the filter when being slid over the padded input. Restoring the original shape from the convolution output by another convolution can be achieved by multiplying the transposed filter matrix (bottom).

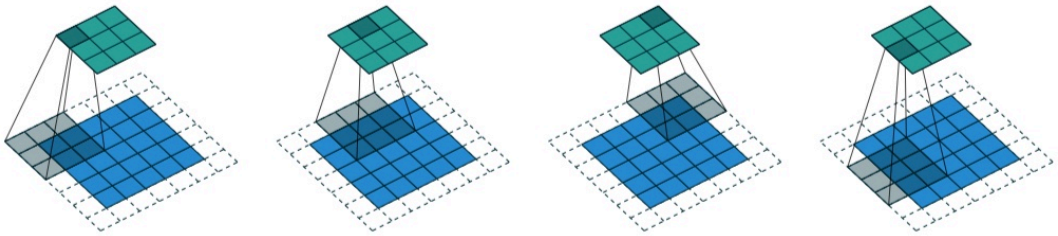


Figure 4.13: Downsampling in the form of a 2D convolution operation on a 5×5 input (blue) with one 3×3 filter (gray), SAME padding $p = 1$ (dashed) and stride $s = 2$ resulting in an output (green) a lot smaller than the input. Taken from [16].

(see Figure 4.13). This enables the use of convolutional layers in the encoder of an autoencoder architecture as introduced in Section 4.5. To build a mirroring decoder, the convolution operation has to be "undone" in the sense that the original shape of the input[28] is restored. This can be achieved by multiplication with the transposed of each filter [37] depicted in the bottom of Figure 4.12. This operation, which is widely referred to as *Transposed Convolution*, is fairly easy to implement as it is simply a well-known convolutional layer but with a certain configuration:

Definition 46 (Transposed Convolution)
For a convolutional layer $f(\cdot, H, s, p)$ with $H \in \mathbb{R}^{m \times u}$ and input size n, the corresponding *Transposed Convolution* layer with filter stack[29] $\widetilde{H} \in \mathbb{R}^{\widetilde{m} \times u}$ and input $y \in \mathbb{R}^{n'}$ is a convolutional layer $f(\widetilde{y}, \widetilde{H}, 1, u - p - 1)$ with

$$\widetilde{y} := \left[\begin{array}{ccccccccccc|ccc} y_1 & 0_1 & \dots & 0_{s-1} & y_2 & \dots & y_{n'} & \dots & 0_1 & \dots & 0_{s-1} & 0_1 & \dots & 0_a \end{array} \right].$$

where $a := (i + 2p - k) \mod s$. As derived in [16] its output size is then

$$s(n' - 1) + a + k - 2p$$

□

Note that in order to match equation 4.23, after each input value $s - 1$ zeros need to be inserted in y resulting in $\widetilde{y}$. The additional a zeros at the left[30] of $\widetilde{y}$ are compensating a possible loss of values due to overflowing windows during the original "forward" convolution. Again, generalization to the 2D transposed convolution is straightforward, yet rather cumbersome to notate.

[28] not necessarily the data itself

[29] Notice: usually, neither are the values of the filters of the transposed convolution related to the ones of the "forward convolution" nor does transposed convolution (try to) invert convolution.

[30] In the 2D case, they would be correspondingly at the bottom, too.

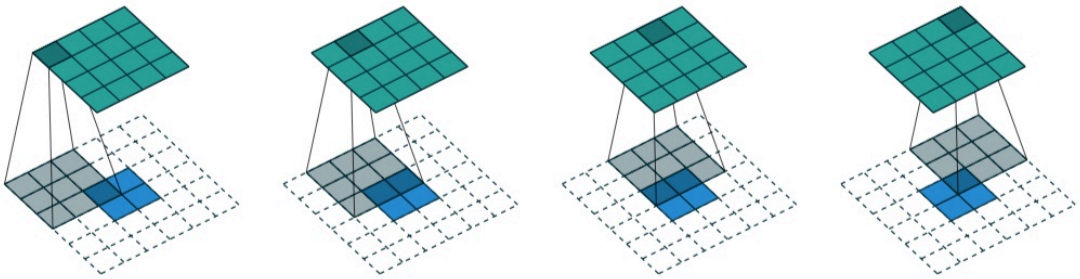


Figure 4.14: Upsampling in the form of a 2D convolution operation on a 2×2 input (blue) with one 3×3 filter (gray), SAME padding $p = 2$ (dashed) and stride $s = 2$ resulting in an output (green) with dimensions twice the ones of the input. Taken from [16].

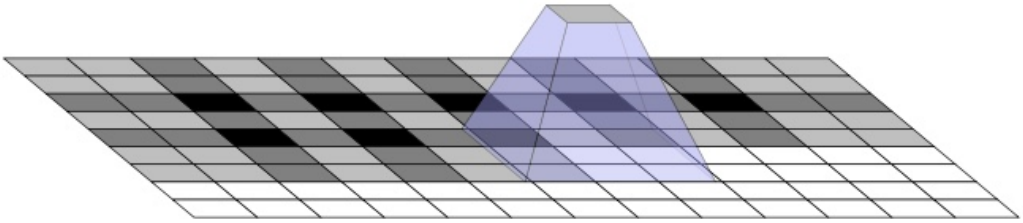


Figure 4.15: Checkerboard artifacts in upconvolution caused by uneven overlap of the sliding window in in which a filter is applied. This arises from the filter size (here $u = 3$) not being divided by the chosen stride (here $s = 2$). Taken from [48].

About terminology: when using a verbalization such as "a transposed convolutional layer with m filters of size u, stride s and padding p", this work, strictly speaking, refers to *the transposed of a convolutional layer* with the mentioned specification. As the transposed convolution operation yields an output larger than the input, it performs a kind of *Upsampling* and therefore is often called *Upconvolution* as well. Also the name *Fractionally Strided Convolution*[31] can be found [37]. Some publications even refer to it as *Deconvolution*, but as this term is used differently in the field of signal processing, this work sticks to the other terminology.

A common pitfall when using upconvolutional layers is to use a filter size which is not divided by the stride: this leads to an uneven overlap of the sliding windows in

[31] Because when $s > 1$, moving one step in the upconvolution's input corresponds to "less than one step" in its output. (In such a situation, the in-between insertion of zeros in $\widetilde{y}$ is necessary.)

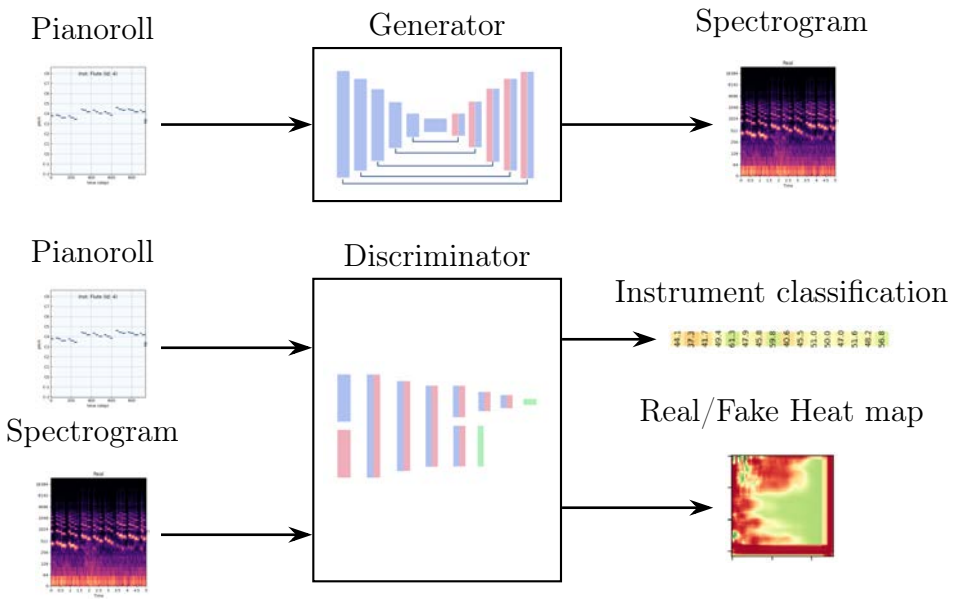


Figure 4.16: High-level illustration of the *orGAN* system: the generator performs score-to-audio translation via a U-Net like deep fully-convolutional autoencoder while the discriminator downsamples a spectrogram concatenated with the corresponding pianoroll to output a patch-wise real/fake classification of the spectrogram and a vector of probabilities for all instruments for being played in this spectrogram.

which a filter is applied and in consequence can lead to checkerboard artifacts in the output [48]. The simplest countermeasure is an appropriate layer design.

4.7 The *orGAN*-Architecture

All of the concepts described above are combined in this thesis to construct a generative adversarial network named *orGAN* for solving the problem of score-to-audio translation. The used architecture is adopted from Pix2Pix [27] and PerformanceNet [72]: the *orGAN* is an auxiliary classifier PatchGAN illustrated from a high-level perspective in Figure 4.16:

The Generator The generator receives a batch of scores of shape $(bs, 935, 128, 16)$ as "side condition" , i.e. a batch of multi-instrument pianorolls as introduced in section 3.4. This shape can be justified as follows: the number 128, as already mentioned, arises from the 128 different pitches available in the MIDI music data format. *orGAN*'s design offers data slots to play 16 instruments at once, from which only 13 are used for training with the URMP dataset. The remaining slots are introduced to

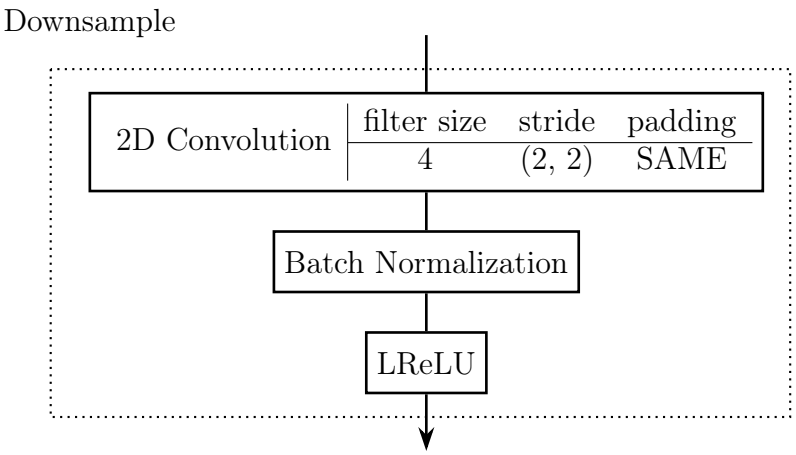


Figure 4.17: A downsampling module consisting of a 2D convolution with stride 2, filter size 4 and SAME padding followed by batch normalization and leaky ReLU activation.

obtain a shape of powers of 2 for computational efficiency as well as easy processing by convolutional layers and, more importantly, to encourage a later extension of the model for new instruments without requiring architectural changes. The number 935 is the number of windows per second (here 187) of the Fourier transform times the duration of one sample (5 seconds). For details on this, consider Section 3.2 and Section 3.4. The batch size bs is set to the unusually low number of 4 samples due to limitations in available computational resources. Note that the generator does not receive a random seed as additional input in the favor of a permanent use of dropout following [27].

The generator consists of a U-net autoencoder architecture with skip connections between corresponding encoder and decoder layers. It consists of convolutional layers only, i.e. it is a fully-convolutional model [38] that can process songs of arbitrary length at once. To make the input nicely processable by the autoencoder, it is heavily padded with zeros in both, time and frequency dimension to have a quadratic shape with side length 1024. This shape almost matching the desired output shape of a spectrogram opens the possibility to build a decoder which is an exact mirror of the encoder – a practice that has been found to be beneficial for lots of autoencoder architectures. For a comparison, consider that the pix2pix super-resolution model [27] has been trained on images sized 256×256. The encoder of the generator then samples its input down to a high-depth tensor of shape $(\mathrm{bs}, 1, 1, 512)$. For this, it uses modules consisting of 2D convolution, batch normalization and LReLU (in this order) summarized in Figure 4.17 following among others again [27]. To actually achieve downsampling, those convolutions are carried out with a stride of 2 in both dimensions and SAME padding while using a filter size of 4. The number of filters

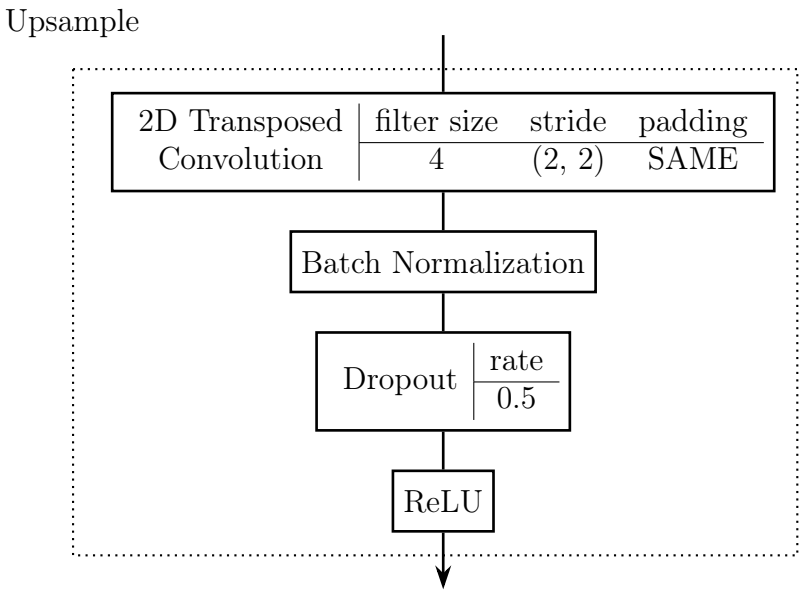


Figure 4.18: An upsampling module consisting of a 2D transposed convolution with stride 2, filter size 4 and SAME padding followed by batch normalization, dropout (optionally) and ReLU activation.

per layer and therefore the output depth is doubled after each layer starting from 64 and holding at 512 filters. Batch normalization is used everywhere, except from the input layer.

The decoder of the generator, as already mentioned, is build to perfectly mirror the encoder. This means, it consists of 10 convolutional layers used for upsampling from the bottleneck to a $(bs, 1024, 1024, 16)$ output. To perform upsampling, those layers perform a 2D transposed convolution with stride 2, SAME padding and a filter size of 4. This configuration should in particular avoid checkerboard artifacts as described in Section 4.6. After each upconvolution, batch normalization and ReLU are applied (see Figure 4.18). The number of filters is held at 512 for the first 4 upsampling layers and then halved after each layer down to an output depth of 16 which is held for the last two layers. All of those layers have U-Net like skip connections to their counterparts in the encoder, i.e. their outputs are concatenated along the depth axis after each layer. Like in [27], the first 3 of the layers also apply dropout with a rate of 50% after batch normalization, but before applying the activation function.

After this stack, additional operations are applied to obtain an output in spectrogram shape (see Section 3.2), namely $(\text{bs}, 935, 1025, 1)$: first, another upconvolu-

tion with one filter of size 2, unit stride[32], VALID padding and LReLU yields a $(bs, 1025, 1025, 1)$ output. Then cropping the formerly added padding on the time axis yields the target shape. In the output layer, LReLU is preferred over ReLU as the latter would drop information in the negative domain. The negative output of LReLU is flipped later when taking the absolute value during the conversion to waveform audio in post-processing (see Section 4.1). The whole generator architecture is also described by Figure 4.19.

The Discriminator The discriminator accordingly receives two inputs: the spectrogram to classify as real or fake and the corresponding multi-instrument pianoroll as condition to verify. At first, those inputs are processed separately: the spectrogram is padded in time to have a quadratic shape and then ran through a 2D convolutional layer with filter size 2, unit stride and VALID padding resulting in a shape of $(\text{bs}, 1024, 1024, 1)$. The pianoroll, meanwhile, is just padded to $(\text{bs}, 1024, 1024, 16)$. After this point, both tensors can be concatenated along the depth axis and then processed jointly by a stack of downsampling modules as in the generator from which the first one does not apply batch normalization while the last two use dropout in order to prevent overfitting and a "too good" performance of the discriminator which might cause the generator to collapse. The data is downsampled towards the shape $(\text{bs}, 64, 64, 256)$ while increasing depth. This output is passed to two different branches in parallel: the first branch leads to a real/fake classification via an additional shape-preserving convolutional layer with batch normalization and LReLU before a final convolution with one filter (and unit stride) and sigmoid activation flattening the output to a $(\text{bs}, p, p, 1)$ "heat map". This "heat map" is the patch-wise real/fake classification of the input spectrogram making *orGAN* a PatchGAN. This output is denoted D^{map}. Above, p is the number of patches per axis. This work experiments with $p \in \{32, 64, 128\}$. To monitor the behavior of the discriminator during training, the mean *Accuracy* of its classifications with a threshold of $t = 0.5$ on a batch of scores $B_z \subset \{0, 1\}^{d_{z_1} \times d_{z_2}}$ and an equally large batch of real spectrograms $B_y \subset \mathbb{R}^{d_{y_1} \times d_{y_2}}$ is calculated as

$$\text{acc}(B_z, B_y, t) := \frac{1}{|B_z|} \sum_{(z,y) \in (B_z \times B_y)} \frac{1}{2p^2} (\text{TP}(D^{\text{map}}(y), 1, t) + \text{TN}(D^{\text{map}}(G(z)), 0, t)) \tag{4.24}$$

where

$$\begin{aligned} &\text{TP}, \text{FP} : \ ([0,1]^{p \times p} \times \{0,1\}^{p \times p} \times [0,1]) \to \mathbb{N}_0, \\ &\text{TP}(a, l, t) := \sum_{i,j} I_t(a_{ij}) l_{ij}, \qquad \text{FP}(a, l, t) := \sum_{i,j} I_t(a_{ij})(1 - l_{ij}) \end{aligned}$$

[32] An upconvolution with this configuration as output layer is also beneficial for avoiding artifacts [48].

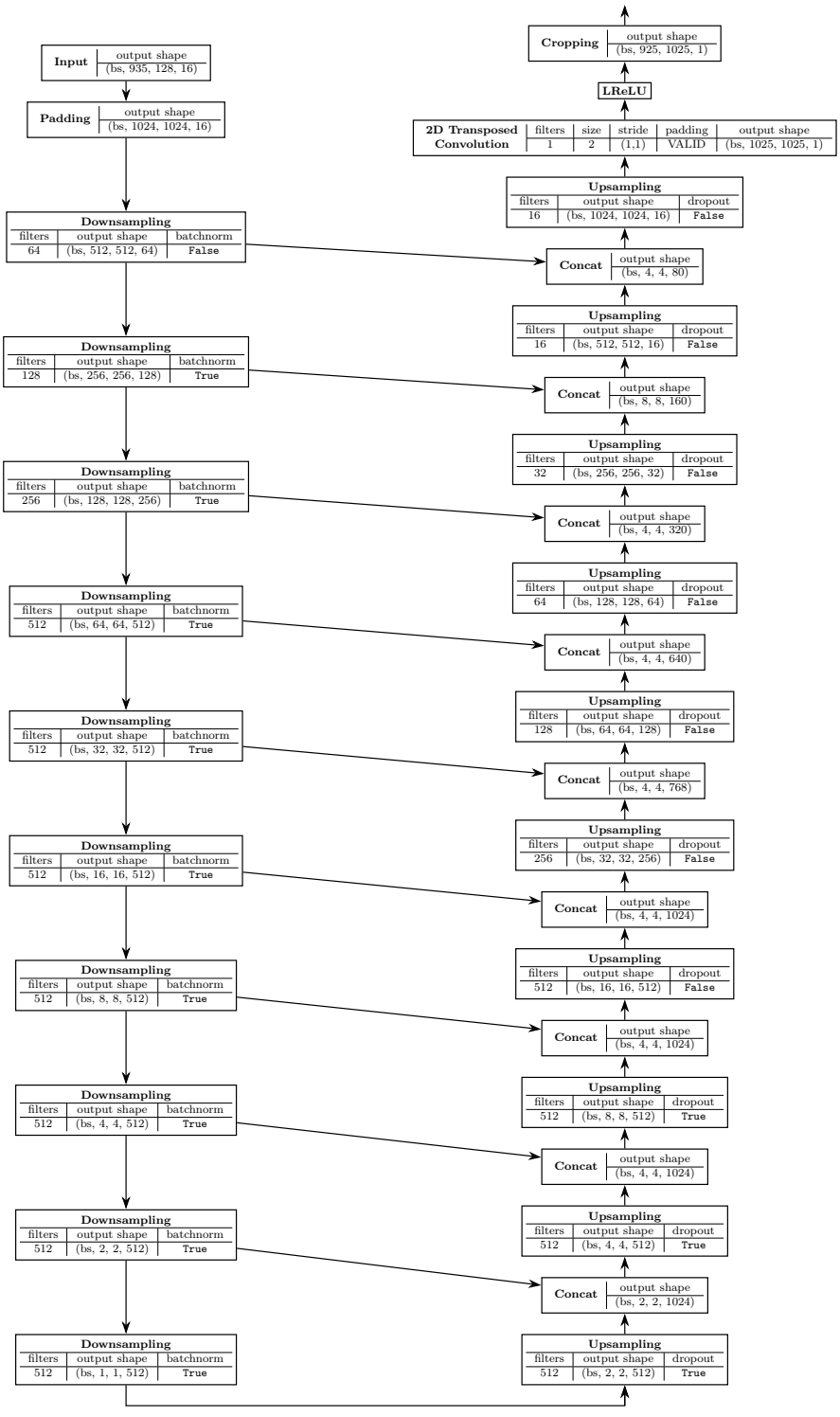


Figure 4.19: A summary of the architecture of the generator used in *orGAN* relying on multiple convolutional layers arranged in an U-Net like Encoder-Decoder design with skip connections.

are the numbers of *True Positives* and *False Positives* (and analogously TN, FN the *True Negatives* and *False Negatives*) given data a, labels l and threshold t with

$$I_t(b) := \begin{cases} 1, & \text{if } b \geq t \\ 0, & \text{else} \end{cases}.$$

Ideally, this accuracy should tend to 0.5 over training so that the discriminator cannot distinguish real and fake samples anymore.

The Instrument Classifier The second branch within the discriminator is the auxiliary classifier intended to identify all playing instruments analogously to the pitch classifier of GANSynth [17]: after downsampling further to $(\text{bs}, 2, 2, 32)$ using the usual configuration with dropout in the last three layers, another convolution with stride 2 and 16 filters of size 2 finally yields an output that can be flattened to a $(\text{bs}, 16)$ vector of so-called *Logits*, i.e. the unscaled values indicating the probability for each instrument to be playing after they are passed through a suitable activation function, which is here the sigmoid function.[33] This output of the discriminator D is denoted D^{inst}. Experiments with including the instrument classifier or not for different hyperparameter configurations are described later in Section 5.4. This discriminator outputs a multi-hot vector $y \in [0,1]^{16}$ whose quality is measured via the cross-entropy loss: the instrument classifier loss over a batch B of tuples (x, y) of a real spectrogram x and the corresponding multi-hot vector indicating playing instruments according to equation (4.4) is

$$L_{\text{inst}}(W_D, (B_x, B_y)) := \\ -\frac{1}{|B_x|} \sum_{(x,y)\in(B_x\times B_y)} \sum_{i=1}^{16} w_i y_i \ln(D_i^{\text{inst}}(x)) + w_i(1-y_i)\ln(1-D_i^{\text{inst}}(x)). \quad (4.25)$$

The training data here is highly imbalanced, i.e. some classes (instruments) are represented by more samples than others. To avoid the loss being misguided by this, the loss for each instrument class $i \in \{0, ..., 16\}$ is weighted with

$$w_i := \frac{1}{|\text{samples for class } i|} \frac{|\text{samples in total}|}{16}.$$

For further insights on the learning progress, the following statistics on the instrument classifier are tracked, yet not optimized: first, the accuracy according to equation (4.24) as well as

[33] The softmax function is not suitable here as it "squeezes" the logits to a probability distribution over all output variables. Instead, what is desired here is an independent probability value for each of the instruments. Hence, the sigmoid function scaling every single logit down to the range $[0, 1]$ independently is chosen here.

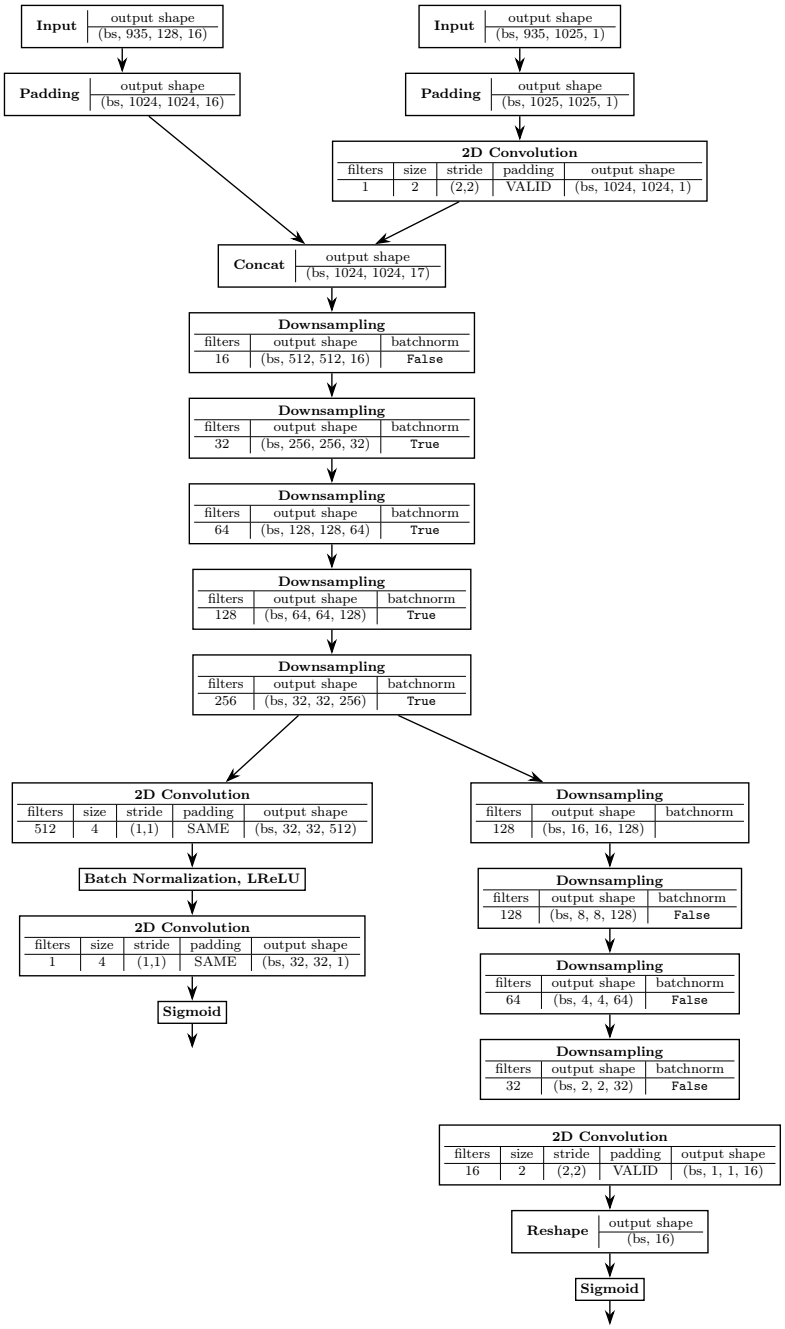


Figure 4.20: A summary of the architecture of the discriminator used in *orGAN* relying on multiple convolutional layers arranged in different pipelines: one input pipeline for the score and one for the corresponding spectrogram (fake or real), downsampling both in parallel before their output is concatenated and passed to a common downsampling stack. The result is fed to both, a "Real vs. Fake"-Patch Classifier and an Instrument Classifier pipeline. Note that dropout when used in downsampling modules is included at the same point as in upsampling modules (see Figure 4.18). Depicted is the architecture variant for 32 patches per axis.

- *Precision*: $\text{prec} := \frac{\text{TP}}{\text{TP+FP}}$
- *Recall* aka *True Positive Rate (TPR)*: $\text{rec} := \text{TPR} := \frac{\text{TP}}{\text{TP+FN}}$
- *False Positive Rate (FPR)*: $\text{FPR} := \frac{\text{FP}}{\text{FP+TN}}$

It is important that the accuracy can be misleading when the numbers of samples per class (here: instrument) are imbalanced which is clearly the case in this project (see again Section 3.1).[34] For a binary classifier as in this work, one can instead consider the curve described by points $(\text{TPR}_{d,l}(\cdot), \text{FPR}_{d,l}(\cdot)) := (\text{TPR}(d, l, \cdot), \text{FPR}(d, l, \cdot))$ as a function of the classification threshold $t \in [0, 1]$ while data d and labels l are fixed, the so-called *Recevier Operating Characteristic (ROC)*. An ideal binary classifier perfectly separates data labeled 0 and 1 and therefore for every t, $\text{TPR} = 1 = 1 - \text{FPR}$. Hence, one can strive for maximizing

$$\text{AUC}(d, l) := \int_{t=0}^{1} \text{TPR}_{d,l}(\text{FPR}_{d,l}^{-1}(t))\ dt,$$

the *Area under Curve (AUC)* or more precisely the *Area under the Receiver Operating Characteristic (AUROC)*. On can see that for a perfect classifier, AUC approaches 1. It also has to be mentioned that all those measures so far rate the classification of each instrument in a multi-instrument sample independently. For further insights, additionally the accuracy comparing multi-class classification outputs as a whole is tracked: in this work this is termed the *Macro-Accuracy* of the instrument classifier defined as

$$\text{acc}_{\text{macro}}(B, t) := \frac{1}{|B_y|} \sum_{(x,y)\in B} I_{16}(\text{TP}(D^{\text{inst}}(x), y, t) + \text{TN}(D^{\text{inst}}(x), y, t))$$

which can be expected to be very low until a late stage of training. All of those measures of the instrument classifier are calculated only over the real samples as it, for instance, would be misleading having the accuracy influenced by a sample which is labeled "violin" while the fake produced by the generator sounds like "flute".

The Composite Loss Inspired by [27] which designs a "composite loss" for the generator as a linear combination of the "regular" GAN loss and the L1 distance between real and fake output, this work experiments with different loss components for both, generator and discriminator. Starting with the generator: first, for a batch of scores $B_z \subset \{0, 1\}^{d_{z_1} \times d_{z_2}}$, the adversarial loss for the generator G is computed as cross-entropy loss

$$L_{\text{adv}_G}(W_G, B_z) := -\frac{1}{|B_z|} \sum_{x\in B_z} \sum_{i,j\in\{1,\dots,p\}} \ln(D_{i,j}^{\text{map}}(G(z)))$$

[34] For instance, when given a dataset with 90% of all samples labeled "A", even a classifier which can not classify samples at all but instead always outputs "A" would achieve 90% accuracy.

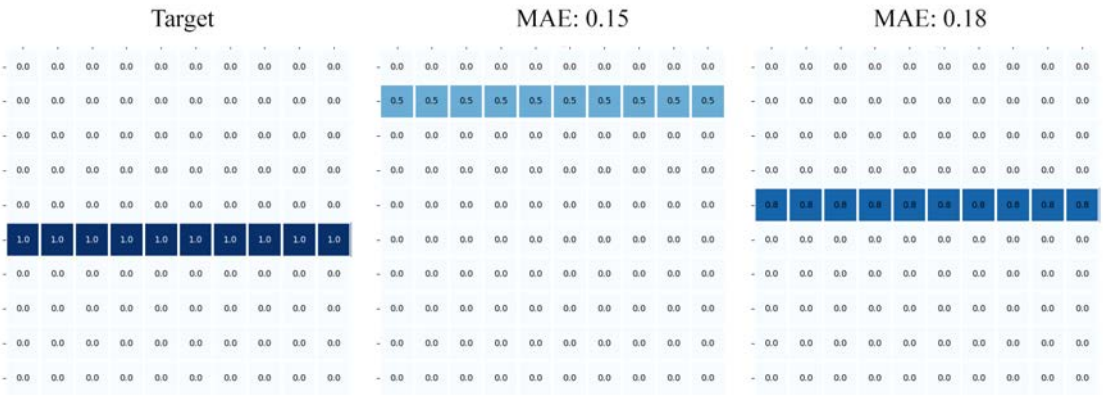


Figure 4.21: A toy example to illustrate a drawback of the L1 loss (Mean Absolute Error): a (schematic) target spectrogram with one pure tone (left) has a lower L1 distance to one containing a much a higher tone of lower intensity (center) than one that represents a higher intensity tone very close to the target (right). This is because the MAE as the name suggests is an average over all pixels and therefore insensitive to structural properties. This can be overcome by the Multiscale Structural Similarity (MS-SSIM) [74].

as the discriminator D with p^2 patches should (in the best case for G) output $1 \in [0,1]^{p \times p}$ classifying the fakes as real. Further, the mean L1 distance to the real spectrograms $B_y \subset \mathbb{R}^{d_{y_1} \times d_{y_2}}$ is calculated:

$$L_1(W_G, (B_z, B_y)) := \frac{1}{|B_z|} \sum_{(z,y) \in (B_z \times B_y)} \frac{1}{d_{y_1} d_{y_2}} \sum_{i=0}^{d_{y_1}} \sum_{j=0}^{d_{y_2}} |G(z)_{i,j} - y_{i,j}|$$

This mean L1 loss, also called *Mean Absolute Error (MAE)*, has a major drawback for this application: it is computed pixel-wise and then returns the average error over the whole spectrogram, i.e. it does not respect spatial structure. This can be seen clearly from Figure 4.21: given a pure tone in the spectrogram at a certain level, a spectrogram containing this tone at a much higher frequency has the same L1 distance as one where this tone is only "one frequency bin away" from the target. If the far away tone now has a lower intensity than the close one, its L1 distance to the target is even better than the one of the close tone. Practically, this means that a synthesizing model minimizing the L1 distance between generated and real data might get stuck in a local optimum yielding a solution not even close to the desired one. For instance, it could learn to play all tones of an instrument at higher frequencies and make it sound like another (related) instrument.

In pix2pix, the discriminator is expected to compensate this as it is designed to provide feedback for local structure only. Instead of relying on this, the drawback can

be overcome by using a similarity index interpreting the means μ_a, μ_b of images $a, b \in \mathbb{R}^{n \times m}$ as luminance, standard deviations σ_a, σ_b as contrast and their covariance σ_{ab}, i.e. their "tendency to vary together", as structural similarity yielding the following comparisons for luminance l, contrast c and structure s [73]:

$$l(a,b) := \frac{\mu_a \mu_b + C_1}{\mu_a^2 + \mu_b^2 + C_1}$$
$$c(a,b) := \frac{\sigma_a \sigma_b + C_2}{\sigma_a^2 + \sigma_b^2 + C_2}$$
$$s(a,b) := \frac{\sigma_{ab} + C_3}{\sigma_a + \sigma_b + C_3}$$

with constants

$$C_1 := (K_1 L)^2, \qquad C_2 := (K_2 L)^2 \qquad \text{and} \qquad C_3 := C_2/2$$

for numerical stability with L the maximal possible distance for two values in the image and hyper parameters $K_1, K_2 << 1$. Typically, the latter are set to $K_1 = 0.01, K_2 = 0.03$ following the experimental results of [73]. Based on these comparisons, an index for structural similarity of images can be build:

Definition 47 (Structural Similarity Index)
Given two images a, b, their *Structural Similarity Index (SSIM)* is defined as

$$\text{SSIM}(a,b) := [l(a,b)]^\alpha \; [c(a,b)]^\beta \; [s(a,b)]^\gamma$$

with $\alpha, \beta, \gamma > 0$ chosen manually. □

As shown by [73], SSIM is symmetric and has a unique maximum in 1 which is reached if and only if $a = b$. It is recommended to compute it locally as among others "image statistical features are usually highly spatially non-stationary" [73]: while a basic sliding window approach is likely to produce block artifacts, it is beneficial to convolve[35] the images with a quadratic Gaussian filter, usually sized 11×11 with a standard deviation of 1.5, which computes the SSIM for every location of this filter on the image before the results are averaged.
To better comply with human perception by comparing image structure at multiple levels of perception, an extended similarity index using downsampling and SSIM iteratively has been proposed:

[35] with unit stride, VALID padding

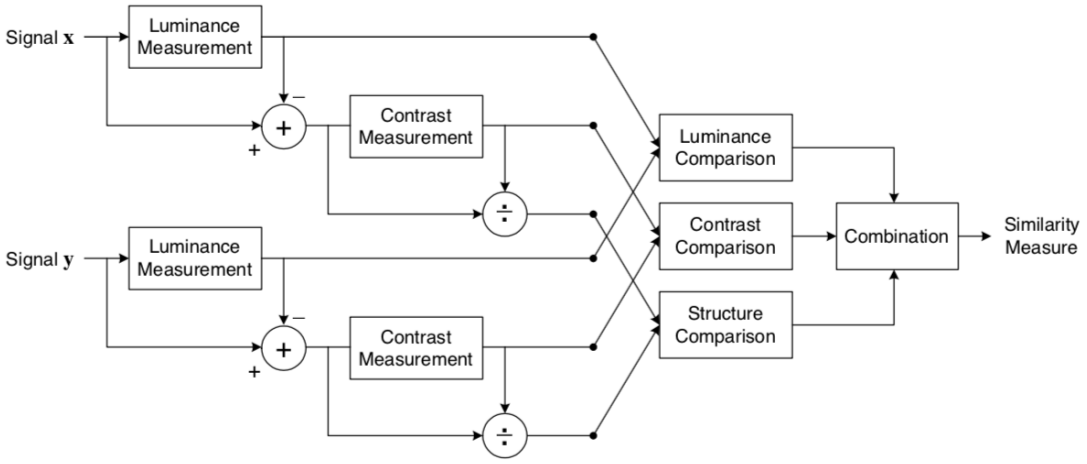


Figure 4.22: The computation of the structural similarity index (SSIM) between two signals (images) x and y. Taken from [73].

Definition 48 (Multiscale Structural Similarity Index)
Given two images a, b, their *Multiscale Structural Similarity Index (MS-SSIM)* of M *levels* or *scales* is defined as

$$\text{MS\_SSIM}(a,b) := [l_M(a,b)]^{\alpha_M} \prod_{j=1}^{M} [c_j(a,b)]^{\beta_j} [s_j(a,b)]^{\gamma_j}$$

with $l_j(a,b) := l(down^{j-1}(a), down^{j-1}(b))$ the luminance comparison after a, b have been downsampled $j-1$ times (and c_j, s_j analogously) and with $\alpha_j, \beta_j, \gamma_j > 0$ chosen manually. □

The superscript $down^{j-1}$ shall indicated iterated application. Above, downsampling can be implemented for instance via 2×2 average pooling as it is done in Tensorflow [1]: this means, every 2×2 region of the input image is reduced to the average over its values. This equals a 2D convolution with one filter $\begin{bmatrix} 1/4 & 1/4 \\ 1/4 & 1/4 \end{bmatrix}$, strides $(2, 2)$ and no bias. Just like SSIM, the MS-SSIM is also symmetric and has a unique maximum in 1 reached at $a = b$ as well. This work uses the default values from [74, 1], namely $M = 5$ scales with parameter values $v := (0.0448, 0.2856, 0.3001, 0.2363, 0.1333)$ for $\alpha_j := \beta_j := \gamma_j := v_j$ for each of the "scales".

Based on this, *orGAN* can incorporate another loss component penalizing lacks in structure which is in this application even more important than in common image processing tasks as a few malformed pixels might not be perceived as visually disturbing while even deliberate changes in a spectrogram can corrupt the resulting audio:

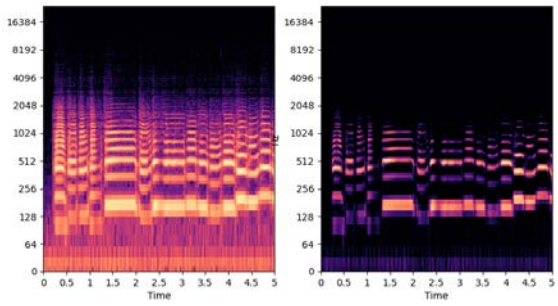


Figure 4.23: Log magnitude spectrograms (right) as produced by *orGAN* contain values close within a small range and therefore have a high MS-SSIM by default. The MS-SSIM yields more meaningful results for spectrograms with values on a decibel scale (left) which show details more clearly.

$$L_{\text{msssim}}(W_G, (B_z, B_y)) := \frac{1}{|B_z|} \sum_{(z,y)\in(B_z\times B_y)} [1 - \text{MS\_SSIM}(G(z), y)]$$

This is expected to drive training towards achieving a MS-SSIM value close to 1 between real and fake spectrograms. As for the spectrograms produced by *orGAN* magnitudes are close within a small range (which is beneficial for training), the MS-SSIM between two of them is already very high by default (see Figure 4.23). Therefore, before evaluating MS-SSIM, spectrograms are converted to decibel scale which is also how they are presented to humans.

The "natural" loss function for the discriminator outputting the probability to be real data for p patches, i.e. the cross-entropy loss, is

$$L_{\text{adv}_D}(B_z, B_y) := -\frac{1}{|B_z|} \sum_{(z,y)\in(B_z\times B_y)} \sum_{i,j\in\{1,\dots,p\}} \ln(1 - D^{\text{map}}_{i,j}(G(z))) + \ln(D^{\text{map}}_{i,j}(y))$$

Additionally, the discriminator should respect the instrument classification on the real samples according to equation (4.25).
The overall loss functions for the generator G and discriminator D of *orGAN* finally are linear combinations of all the component losses described above:

$$L_G := L_{\text{adv}_G} + \lambda_{L1} L_1 + \lambda_{\text{msssim}} L_{\text{msssim}} + \lambda_{\text{inst}} L_{\text{inst}}$$
$$L_D := \frac{1}{2} L_{\text{adv}_D} + \lambda_{\text{inst}} L_{\text{inst}}$$

where all weights are non-negative. The factor $1/2$ in the discriminator loss is intended to slow down the learning of D relative to G preventing a collapse as it has been mentioned in Section 4.4 and proposed by [27]. Experiments for different

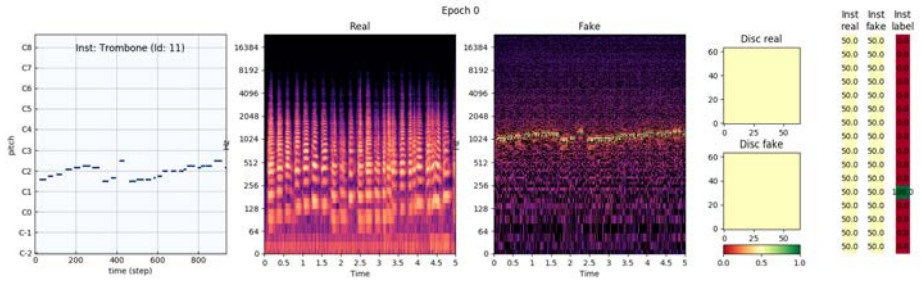


Figure 4.24: The output of the untrained *orGAN* model for one sample as a sanity check of the architecture.

combinations of values for those weights and results will be described in Section 5.1. Optimization is done by the Adam optimizer with a learning rate of $2 \cdot 10^{-4}$ and $3 \cdot 10^{-4}$ for certain setups. The whole training is executed on a single NVIDIA Volta GPU, the `Titan V X`.

Noise Subtraction Most of the records come with decent background noise in the lower frequencies of the spectrograms partially mixed with artifacts from the STFT. In particular, there are samples containing "silence" within a song. *orGAN* tries to synthesize this background noise, too. The problem is, that even delight deviations from the natural background sounds can result in distributing noise. Therefore as a step of post-processing, this work applies a very simple form of "denoising": after a model is trained, *orGAN* is fed an empty pianoroll resulting in a spectrogram of the synthesized background noise and artifacts only. Afterwards, when synthesizing music, this "standard noise" is subtracted from every generated spectrogram noticeably improving the perceived audio quality. Note that this is done *outside* the training loop and beyond all performance measures, so that there is no influence on empirical results regarding the model. As an advantage beside the high ease of computation, this procedure largely preserves instrument-specific detail and in particular can be assumed to not affect the synthesized timbre.

As a simple sanity check of the network architecture, a visualization of *orGAN*'s output without having had any training before can be considered: in Figure 4.24, it can be seen that the network produces random noise due to the initialization of its weights while the score consisting of ones is passed through the whole network without modifications due to the skip-connections and the usage of ReLU. It can also be validated that the discriminator does not suffer from any bias but instead outputs a probability of 0.5 everywhere.

Chapter 5

Experiments

5.1 Loss Composition

It has been described in Section 4.7 that inspired by pix2pix, *orGAN* uses a loss function composed of multiple measures:

$$L_G := L_{\text{adv}_G} + \lambda_{L1} L_1 + \lambda_{\text{msssim}} L_{\text{msssim}} + \lambda_{\text{inst}} L_{\text{inst}}$$
$$L_D := \frac{1}{2} L_{\text{adv}_D} + \lambda_{\text{inst}} L_{\text{inst}}$$

While fixing the weight of the adversarial loss for the discriminator and the generator at one, experiments are conducted with all other weights of these linear combinations: Following [27] and [54], first the differences in using $\lambda_{L1} = 0$ (termed *Vanilla GAN*) versus $\lambda_{L1} = 100$ (shortened *L1 model*) are explored.[1]. Then this is compared against a variant with $\lambda_{L1} = 0$ but $\lambda_{\text{msssim}} = 100$ (referred to as *MS-SSIM model*) to check, whether the deviation of the MS-SSIM from one is indeed a performance measure that is superior to the L1 distance in this task as conjectured in Section 4.7. All those experiments are conducted with a baseline for the discriminator's output size of 64×64. To verify that the MS-SSIM loss does not make the adverasrial loss unnecessary, but instead *orGAN* benefits from both, all those models are compared against a model in which the adversarial discriminator is omitted but instead only the generator with $\lambda_{\text{msssim}} = 100$ is trained (referred to as *Vanilla MS-SSIM*). For the models that include an instrument classifier (see Section 5.4), a constant weight of $\lambda_{\text{inst}} = 0.5$ is used so that it is roughly on the same level as the adversarial loss at the beginning of training.

Vanilla GAN vs L1 Figure 5.1 illustrates that incorporating the L1 loss with $\lambda_{L1} = 100$ in addtion to the adversarial loss highly speeds up training: after starting

[1] In statements like this in this section, all weights that are not explicitly mentioned can be considered zero.

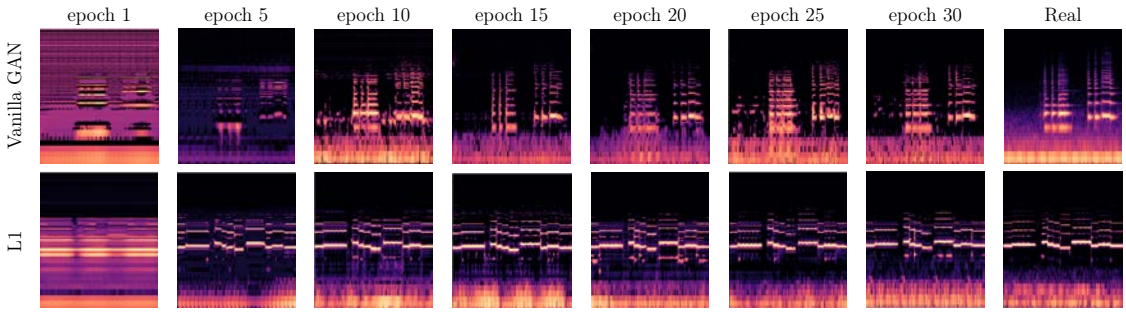


Figure 5.1: The results for two randomly selected samples after different epochs of training in a vanilla GAN (i.e. $\lambda_{L1} = 0$) versus a model with $\lambda_{L1} = 100$ referred to as *L1*. It can be seen that the L1 model arrives at smooth but detailed and globally coherent results significantly faster than the Vanilla GAN.

with a high degree of bluriness, the L1 model quickly learns to produce detailed yet smooth local structure that is also globally coherent. In constrast, the vanilla GAN focuses on fine-grain structural details with the drawback of producing prominent artifacts more often and appearing rather jaggy. Nevertheless, both models converge to the same level of L1 loss but, as one would expect, the L1 model achieves a slightly lower minimum value in less training time (see Figure 5.2). However, it is remarkable that from this metric, both models are very similar which does neither reflect the differences in the quality of the produced spectrograms nor in the subjectively perceived audio quality.

L1 vs MS-SSIM Figure 5.4 shows exemplary that the MS-SSIM model appears to produce much more accurate details than the L1 variant, especially when it comes to overtones in the high frequencies which are crucial for perceiving the music as clear instead of dull. Further, at a closer look the L1 model more often fails to generate locally coherent structure. Also when it comes to faking on- and offsets realistically, the MS-SSIM model outperforms the L1 variant.

MS-SSIM vs Vanilla MS-SSIM In order to verify that the generator actually benefits from the discriminator's feedback, an *orGAN* instance with $\lambda_{\text{msssim}} = 100$ (MS-SSIM) is compared to a model running the generator in stand-alone mode without the discriminator, termed *Vanilla MS-SSIM*. It can be seen from Figure 5.3 that outside the adversarial setting the generator model trained with respect to the MS-SSIM only produces blurry results. Those results become very similar to the real spectrograms in global structure already after the first epochs of training. However, after this point results do not improve further and the model completely fails to generate the fine-grain frequency modulations making up timbre. Therefore, the music synthesized with the vanilla MS-SSIM model lacks instrument specific characteristics and is rather perceived as synthetic "beeps".

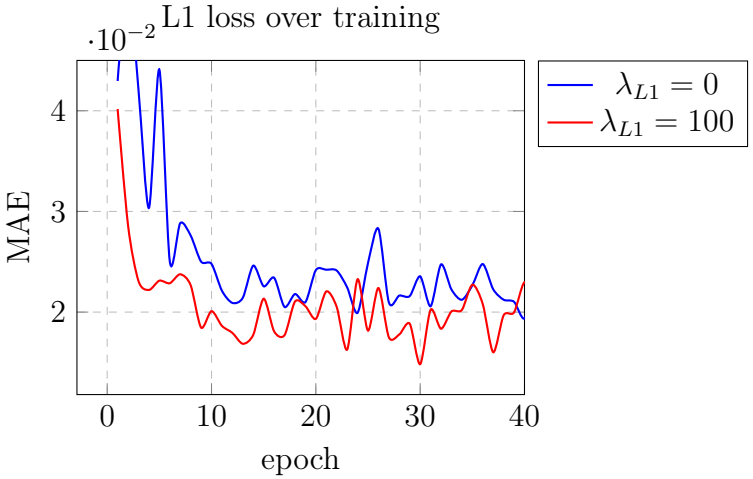


Figure 5.2: The development of the L1 loss on the validation set over training time: both, the vanilla GAN with $\lambda_{L1} = 0$ and the variant with $\lambda_{L1} = 100$ converge to the same level. However, as one can expect, the L1 variant is faster and achieves a better minimum.

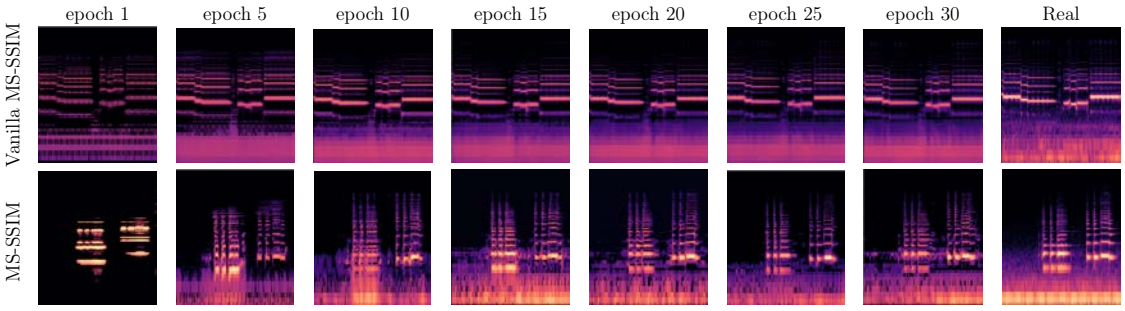


Figure 5.3: The results for two randomly selected samples over different epochs of training in a non-adversarial vanilla MS-SSIM (i.e. $\lambda_{L1} = 0$) versus a GAN model with $\lambda_{msssim} = 100$ referred to as *MS-SSIM*. The latter takes more time of training to achieve convincing results, but captures details accurately while the Vanilla MS-SSIM model suffers from blurriness and largely fails to fake different timbres via small frequency modulations.

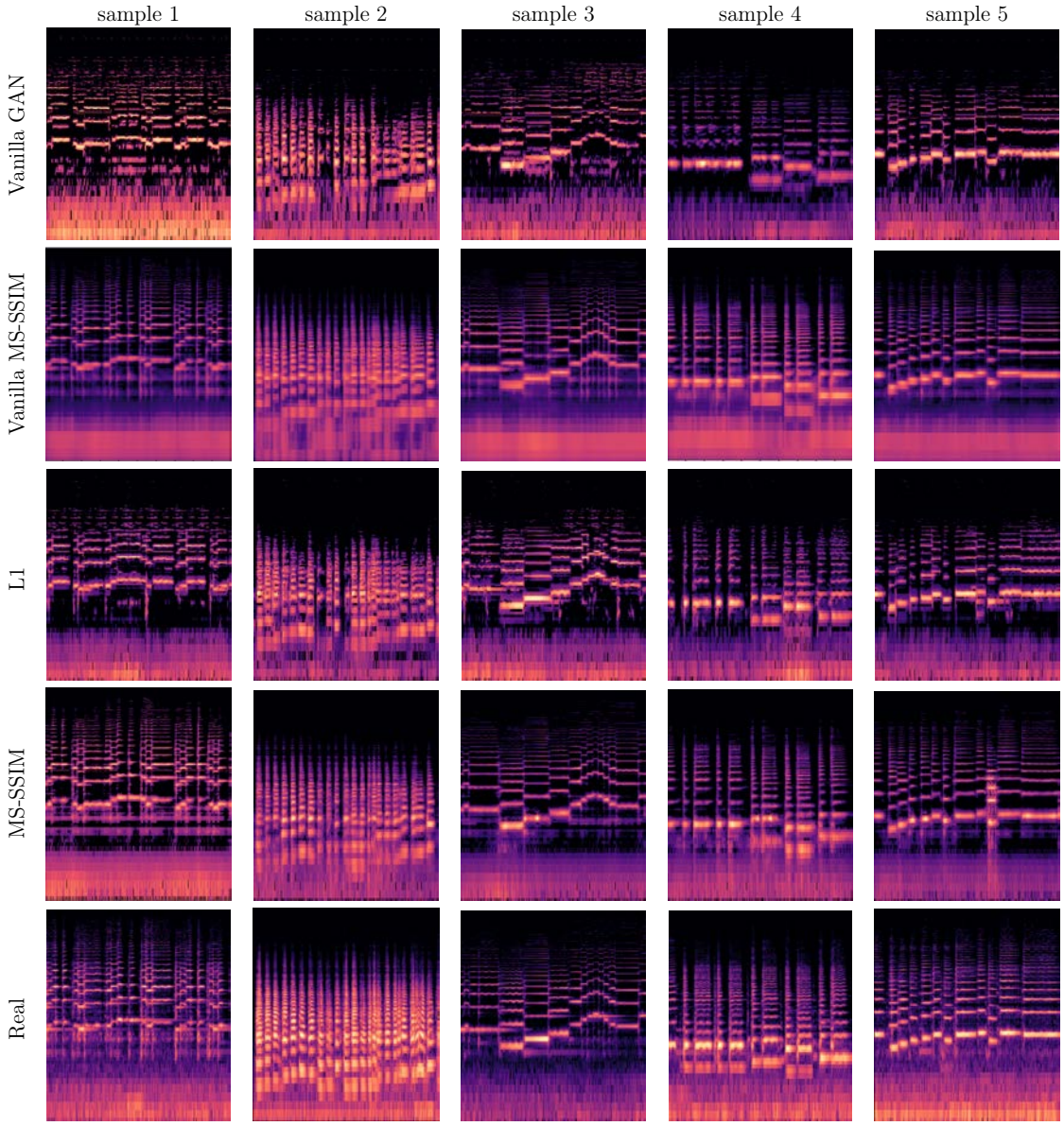


Figure 5.4: Results obtained from a model trained with $\lambda_{L1} = 100, \lambda_{\text{msssim}} = 0$ (marked *L1*) versus one trained with $\lambda_{L1} = 0, \lambda_{\text{msssim}} = 100$ (marked *MS-SSIM*) and spectrograms from real recordings. As a baseline, also samples from a configuration incorporating only the adversarial loss (*Vanilla GAN*) and another one using the generator outside the adversarial setting to maximize the MS-SSIM (*Vanilla MS-SSIM*) are provided.

5.2 Varying Patch Size

As described before, *orGAN* uses a PatchGAN architecture, i.e. its discriminator outputs not a single scalar classifying the input as a whole but instead a $n \times n$ tensor where each pixel corresponds to the classification of one patch in the input tensor. For given output dimensions, the resulting patch size can be calculated by considering the arithmetic of a convolutional layer (Section 4.6): one value in the output of a 2D convolution with SAME padding, stride s and filter size $u \times u$ has a "preimage" (also called the *Receptive Field*) just as large as the filter. Therefore, an output of size $v \times v$ has a receptive field sized

$$r(v, u, s) := (v - 1) * s + u.$$

Applying this iteratively for all convolutional layers $f_m \circ \ldots \circ f_1$, a single value in the output, i.e. $v = 1$, can be traced back to its corresponding patch size $p \times p$ in the input [27]:

$$p = r(r(\ldots r(r(1, u_m, s_m), u_{m-1}, s_{m-1}) \ldots), u_1, s_1)$$

The output size $n \times n$ is not explicitly mentioned above but instead latent in the network architecture, i.e. in the number and configuration of the convolutional layers. For the baseline of experiments here, a $\mathbf{64 \times 64}$ output is used. The discriminator architecture for this described as triples[2] (a, u, s) from input to output layer where a is the number of filters is

$$(16, 4, 2) \to (32, 4, 2) \to (64, 4, 2) \to (128, 4, 2) \to (256, 4, 1) \to (512, 4, 1) \to (1, 4, 1)$$

This results in $\mathbf{p = 190}$. Additionally, a $\mathbf{32 \times 32}$ output with architecture

$$(16, 4, 2) \to (32, 4, 2) \to (64, 4, 2) \to (128, 4, 2) \to (256, 4, 2) \to (512, 4, 1) \to (1, 4, 1)$$

and accordingly large $\mathbf{p = 286}$ is tried. As a third variant, a fine-grain $\mathbf{128 \times 128}$ output is used with the architecture

$$(16, 4, 2) \to (32, 4, 2) \to (64, 4, 2) \to (256, 4, 1) \to (512, 4, 1) \to (1, 4, 1)$$

For this, the patches are much smaller having size $\mathbf{p = 94}$. In order to avoid side effects on the experimental results, above architectures are designed so that differences between them are as small as possible. The used compound loss here is the MS-SSIM variant which has been found to be most beneficial in Section 5.1. To emphasize the effect of the patch size via the adversarial loss, the experiments here are carried out with $\lambda_{\text{msssim}} = 10$.

[2] Actually, the number of filters per layer is irrelevant for the computations here but nonetheless is provided for a complete overview of the architecture. Anyway, in-between operations such as ReLU and batch normalization are omitted here for compact reading.

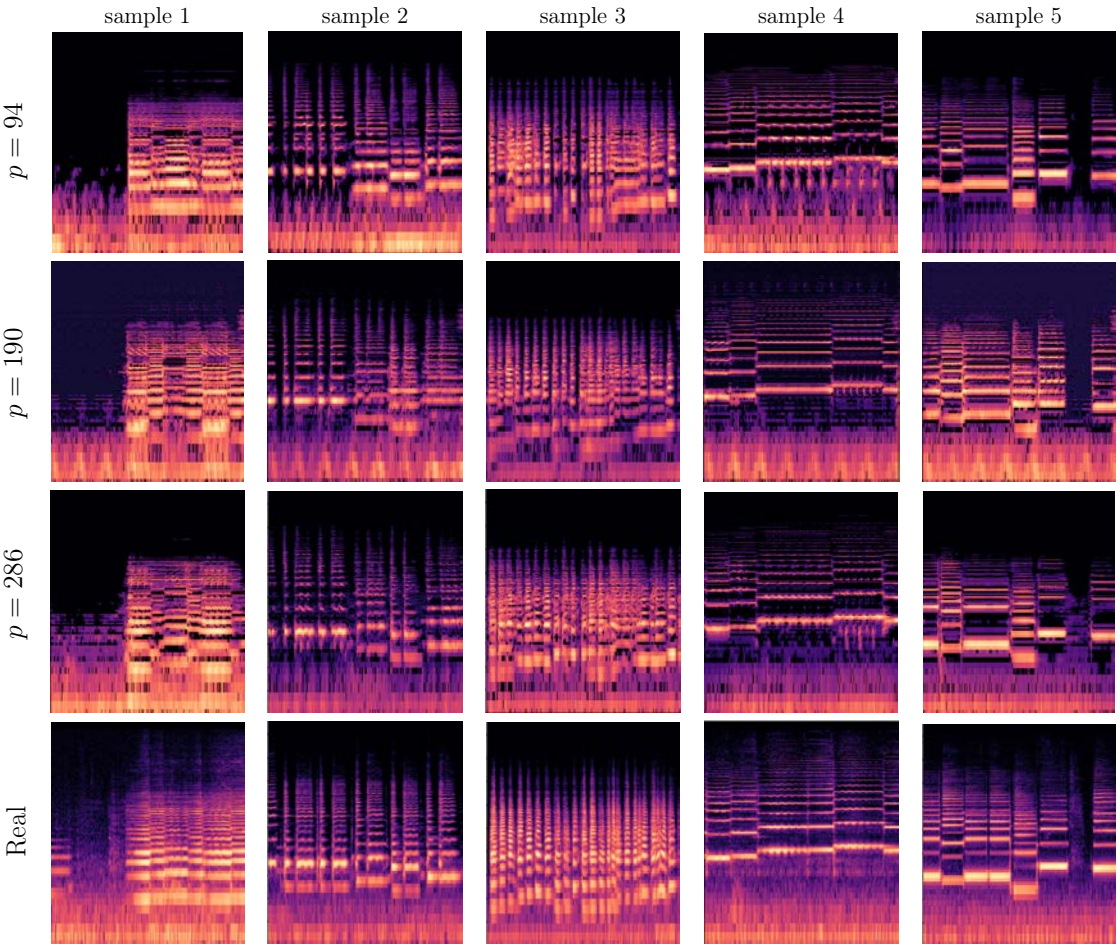


Figure 5.5: Fake spectrograms for different patch sizes in the discriminator: a 128×128 output corresponding to 94×94 patches leads to tiling artifacts while the results of the other variants are quite similar. However, medium sized patches of 190×190 appear to lead to slightly less structural errors compared to the model with $p = 286$.

From a subjective visual evaluation of the generated spectrograms as in Figure 5.5, the $p = 286$ configuration appears to be be superior over the one with $p = 94$ as it suffers much less from tiling artifacts. On the other hand, at a closer look it produces erroneous structural details slightly more often than the medium patch variant with $p = 190$ while both are on a very similar level of quality. This basically coincides with the findings of [27].

To gain another qualitative insight, in Figure 5.6 the "noise" produced by the three models is compared: the term *noise* here refers to the output of a model when it is fed an empty pianoroll. Ideally, this should result in an audio containing only silence or at most low-level background sounds that are also present in the training data. Here, one has to observe artifacts instead. This is partially because of the STFT which already produces some artifacts during pre-processing which the model tries (and largely fails) to mimic along with "real audio". Focusing on the model-specific differences in this "synthesized noise", one may notice that larger patches appear to correspond to a lower recurring period of the artifact patterns. Conversely, a low patch size results in smaller but more frequently recurring elements. Artifacts can be partially reduced by assigning a higher weight to the MS-SSIM in the compound loss: when choosing $\lambda_{\text{msssim}} = 100$ instead of $\lambda_{\text{msssim}} = 10$ as it has been for this experiments, artifacts violating globally coherent structure are penalized more and therefore they become less prominent (see Figure 5.7).

As the L1 loss is not directly optimized here, it can be used for a brief quantitative comparison: from Figure 5.8 it can be inferred that all variants converge to the same L1 loss. It has been mentioned before, that because of the nature of L_p metrics this does not necessarily imply that their results are of the same musical and acoustic quality. However, it can be seen that the configuration with $p = 286$ minimizes the L1 loss a little faster than the others. In terms of the MS-SSIM, which is incorporated in the compound loss function here and thus optimized directly, all models perform in a very similar way while the $p = 268$ and $p = 190$ variants achieve better peaks than the architecture with small $p = 94$ patches.

5.3 Transfer Learning for Multi-Instrument Play

So far, all *orGAN* models have been trained to synthesize only one instrument at a time. A whole ensemble then can be faked by feeding *orGAN* the pianoroll for each instrument separately and merging the resulting audio tracks via addition.

Native multi-instrument play To investigate the model's capabilities for native multi-instrument play, first it is analyzed how a model that has been trained on single-instrument samples behaves when it is provided multi-instrument pianorolls. In Figure 5.10, sample results of this experiment with the two models that have

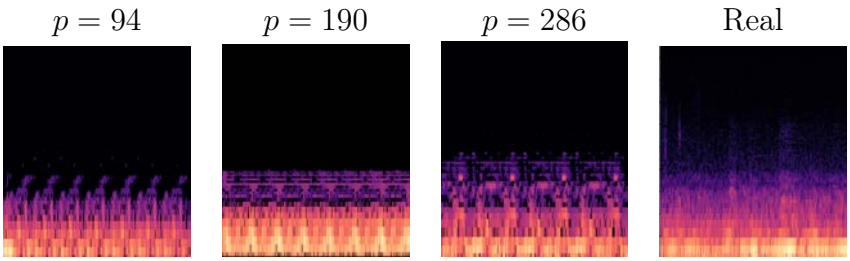


Figure 5.6: The artifacts produced when an *orGAN* model with patch size $p \times p$ is fed an empty pianoroll which should result in silence or background noise at most (as in the spectrogram of a real recording at the left). The models for all patch sizes show tiling artifacts and recurring patterns.

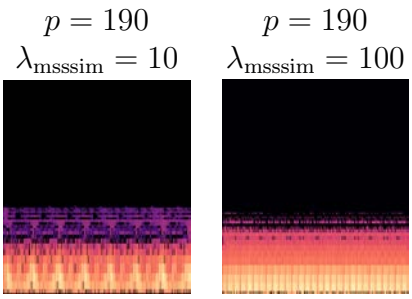


Figure 5.7: The artifacts in the output for models with different weights for the MS-SSIM in the compound loss: penalizing the structural loss more leads to "smoother" and "less random" artifacts already similar to the one in a spectrogram from a real recording as shown in Figure 5.6. Also tiling patterns seem to vanish largely.

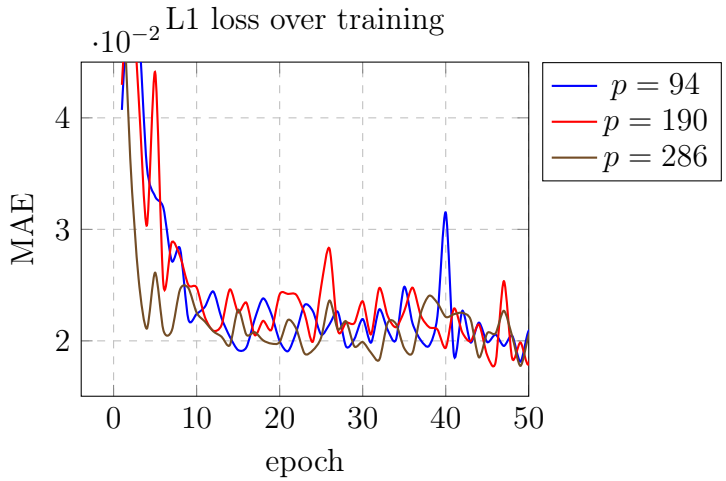


Figure 5.8: The development of the L1 loss on the validation set over training time: all variants of the PatchGAN with different patch sizes p converge to the same level of L1 loss in the long run. It can be seen that the configuration with $p = 286$ and a 32×32 discriminator output reaches its minimum a little faster than the others.

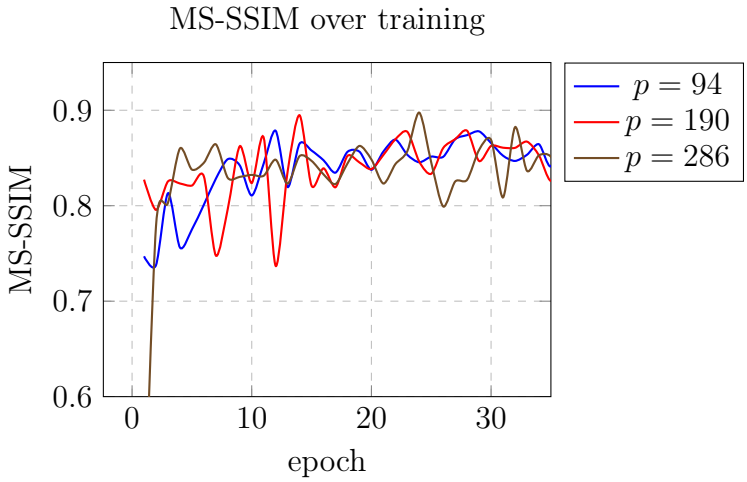


Figure 5.9: The development of the MS-SSIM on the validation set over training time: the different patch configurations turn out to perform similarly over time without systematical or remarkable differences.

performed best so far, i.e. the MS-SSIM models with $p = 190$ and $p = 286$, are provided: expectedly, both models fail to merge the different tracks right away. Instead, only some elements of each track are present in the resulting spectrogram while others appear to "multiply each other out" while being passed through the convolutional layers.

Transfer learning A more sophisticated approach is to apply the concept of transfer learning: instead of training a new model for multi-instrument play from scratch, the trained single-instrument model can be used to refine its internal representations according to new multi-track training data. This is expected to make the task of learning the distribution of multi-instrument data more feasible, as the score-to-audio translation for all instruments has already been learned. The "remaining" task then is to learn an adequate content-based strategy to merge tracks. The result might be as simple as addition, but in this way, the model is given control over the interaction of instruments. During this training, the model is not only fed the new multi-instrument data but the already seen single-instrument data as well, as it still should be able to synthesize both even after transfer learning. To be able to rate the effects of transfer learning, another model is trained on both, single- and multi-instrument samples from the very beginning.
From the samples in Figure 5.10 it can be seen, that even after training on multi-instrument samples both models do not achieve the same fake quality as they did in the single-instrument case. While the $p = 286$ model appears to miss detailed structural feedback from the discriminator so that its produced spectrograms highly lack local coherence in fine-grain structure, the $p = 190$ model's results are visually closer to the real spectrograms. The model that is trained on all samples from the start performs overall similar to the $p = 190$ after transfer learning: the results are acoustically quite close to the real data but still far from being visually equal. Because of this, the resulting audio is clearly different from the real recording often missing tones or mixing up instrument timbres. Nevertheless the additional training at least improves the visual and acoustic quality of single-instrument samples (see Figure 5.11) by reducing blurriness, enhancing contrast between frequency bands and faking on-/offsets more accurately. The reason for this effect has to be suspected in the additional amount of training data rather than in the multi-instrument nature of the samples.

Playing all at once versus playing separately This raises the question, whether letting a model (before or after transfer learning) fake all instruments of one song separately and merging them via addition of the generated waveform audio can provide more convincing results. This question may be answered by analyzing the samples in Figure 5.12: there, the following variants are compared: first, samples from the MS-SSIM model with $p = 190$ *after* transfer learning where all instruments are played simultaneously, i.e. the model produces one pianoroll for all instruments

(*After transfer + at once*) versus samples where each track has been passed separately to the same model (*After transfer + separately*). Second, those two are set side by side with samples from the same model *before* transfer learning so that tracks have to be synthesized separately as well (*Before transfer + separately + mid patch*). Third, the latter is compared to its large patch counterpart with $p = 286$ (*Before transfer + separately + large patch*). Lastly, also samples from the model trained on all samples from the beginning, i.e. without transfer learning are included. At a first glance, intuition suggests that all those results are quite similar. A detailed comparison with the real data reveals that the models without training on multi-instrument data (i.e. the *Before transfer* models) lack structural precision and contain small yet noticeable artifacts. The two *After transfer* models' results are closer to the ground truth on a visual level. In acoustic perception the *After transfer + separately* strategy clearly outperforms *After transfer + at once* as the latter expectedly expresses the different instruments less clear and tends to sporadic yet disturbing erroneous mixtures of frequencies. The model without transfer learning produces visually similar results for most samples but also fails for a few by producing structural errors. Especially the *Without transfer + separately* variant is likely to produce artifacts. Nevertheless, their perceived audio quality in most cases exceeds the one of the other models.

5.4 Including an Instrument Classifier

In order to improve the generation of realistic and adequate timbre, the inclusion of an instrument classifier in *orGAN*'s discriminator is investigated. Following the approach of using a pitch classifier in GANSynth [17], this is intended to provide additional feedback to the generator. The required architecture has already been described in Section 4.7. Basically, the discriminator outputs a multi-hot vector indicating all the detected instruments in a given sample in addition to its real versus fake classification. It is noteworthy that the instrument classifier is only trained on the real samples, i.e. the loss on the fake samples is not incorporated. Instead, the latter is included in the generator's compound loss (see Section 5.1). Restricting to real data is not only required as otherwise the classifier could be mislead [3]: it also opens the possibility to use the instrument classifier as a simple "in-house quality measure" for the generated samples instead of training such a model externally like in GANSynth [17].

This work investigates the inclusion of the instrument classifier for both stages, before and after transfer learning for multi-instrument play, on the best performing

[3] for instance: if the generator's fake looks like a violin while the ground truth is flute, the discriminator would actually be penalized for "correctly" outputting the label for violin.

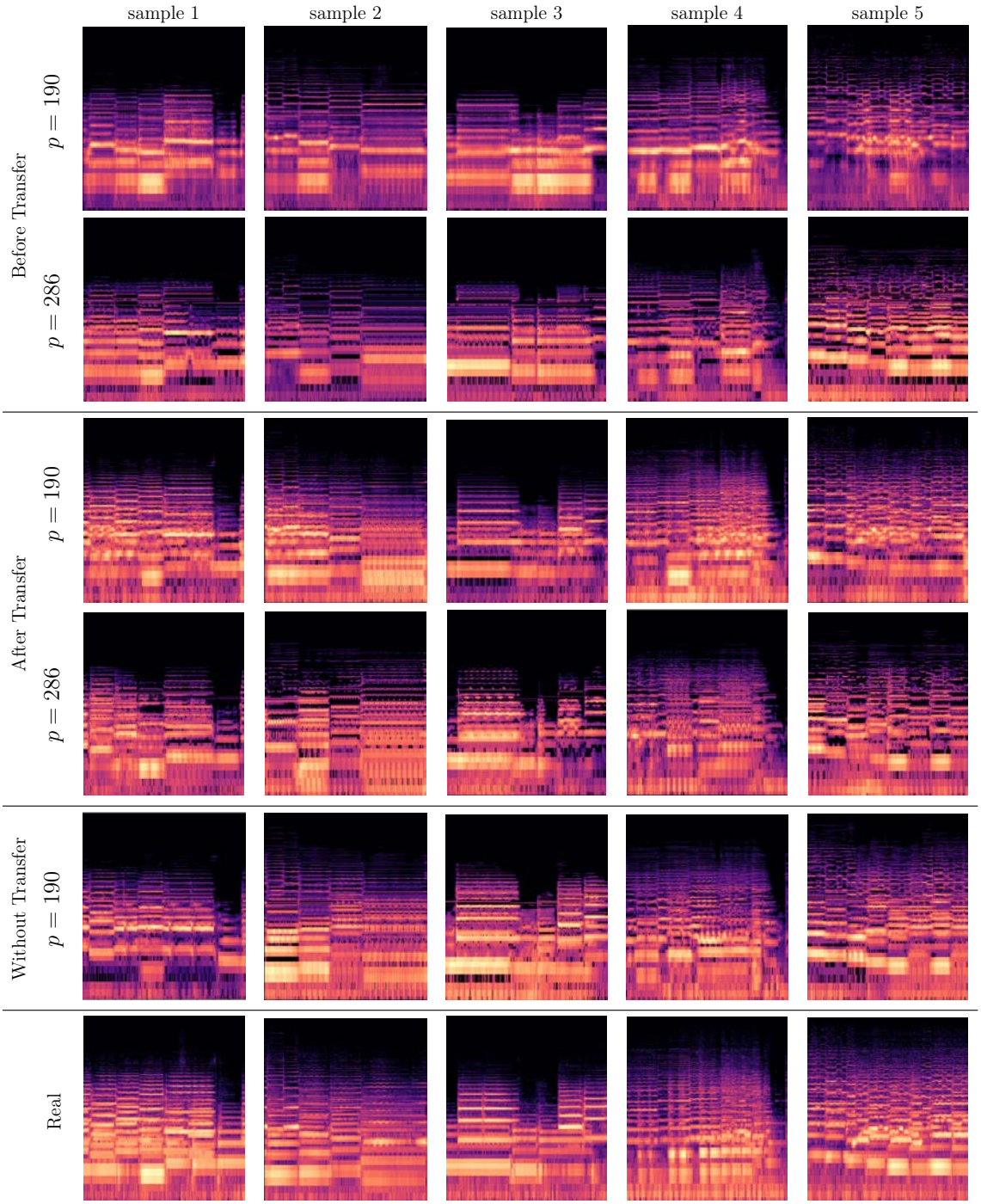


Figure 5.10: Multi-instrument samples before, after and without transfer learning: both models that performed best so far, the MS-SSIM models with $p = 190$ and $p = 286$ fail to synthesize multiple instruments at once without extra training (top two rows). After transfer learning, the $p = 190$ model produces results (third row) that are at least visually close to the real data (last row) while the samples from the large patch variant (fourth row) are missing structural coherence. Training on all samples form the start without transfer learning (second last row) leads to an overall quality similar to the $p = 190$ model with transfer learning.

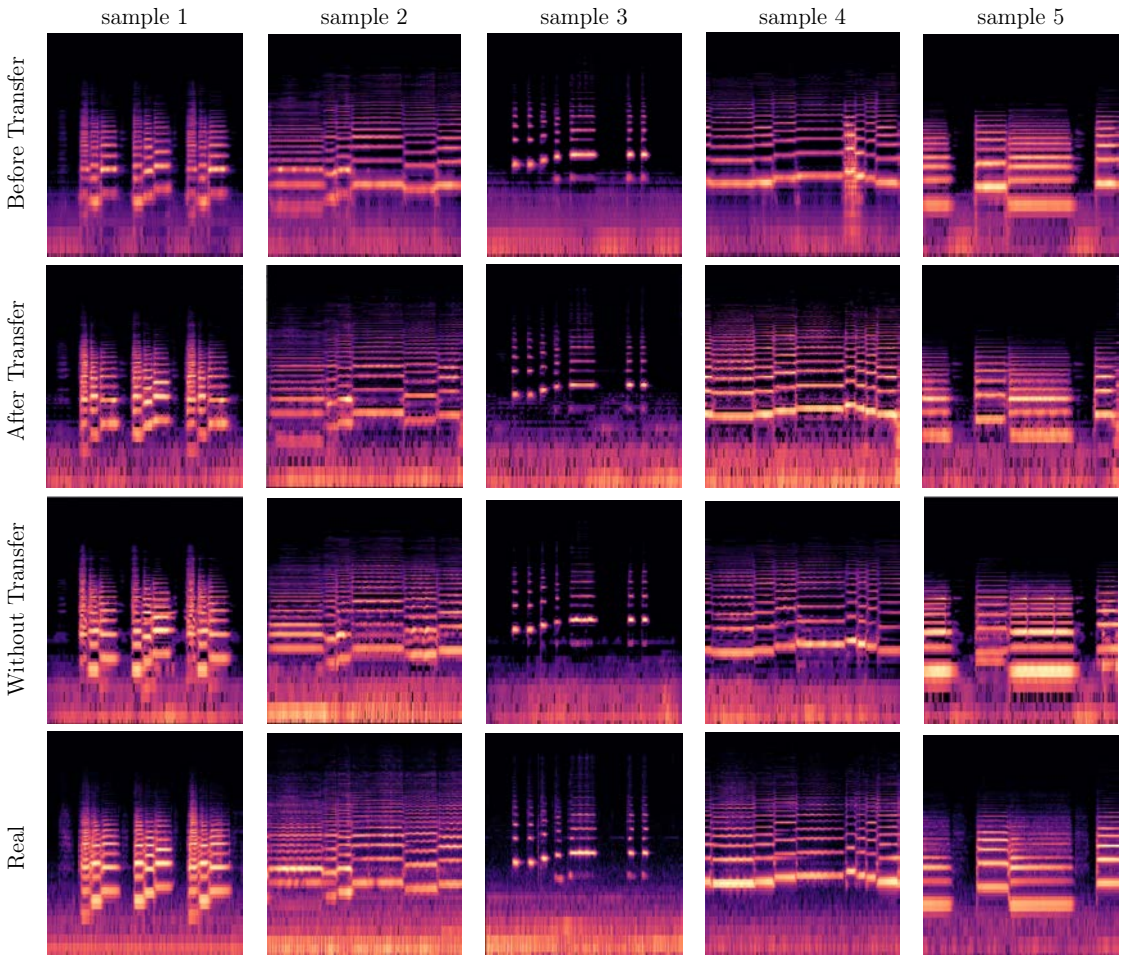


Figure 5.11: Single-instrument samples before, after and without transfer learning on the $p = 190$ MS-SSIM model: after transfer learning including multi-instrument data, the model produces richer and more accurate fake spectrograms, especially when it comes to the contrast between frequency bands as well as on- and offsets. The very same effect can be observed without transfer learning, i.e. when training on all samples right from the beginning of training.

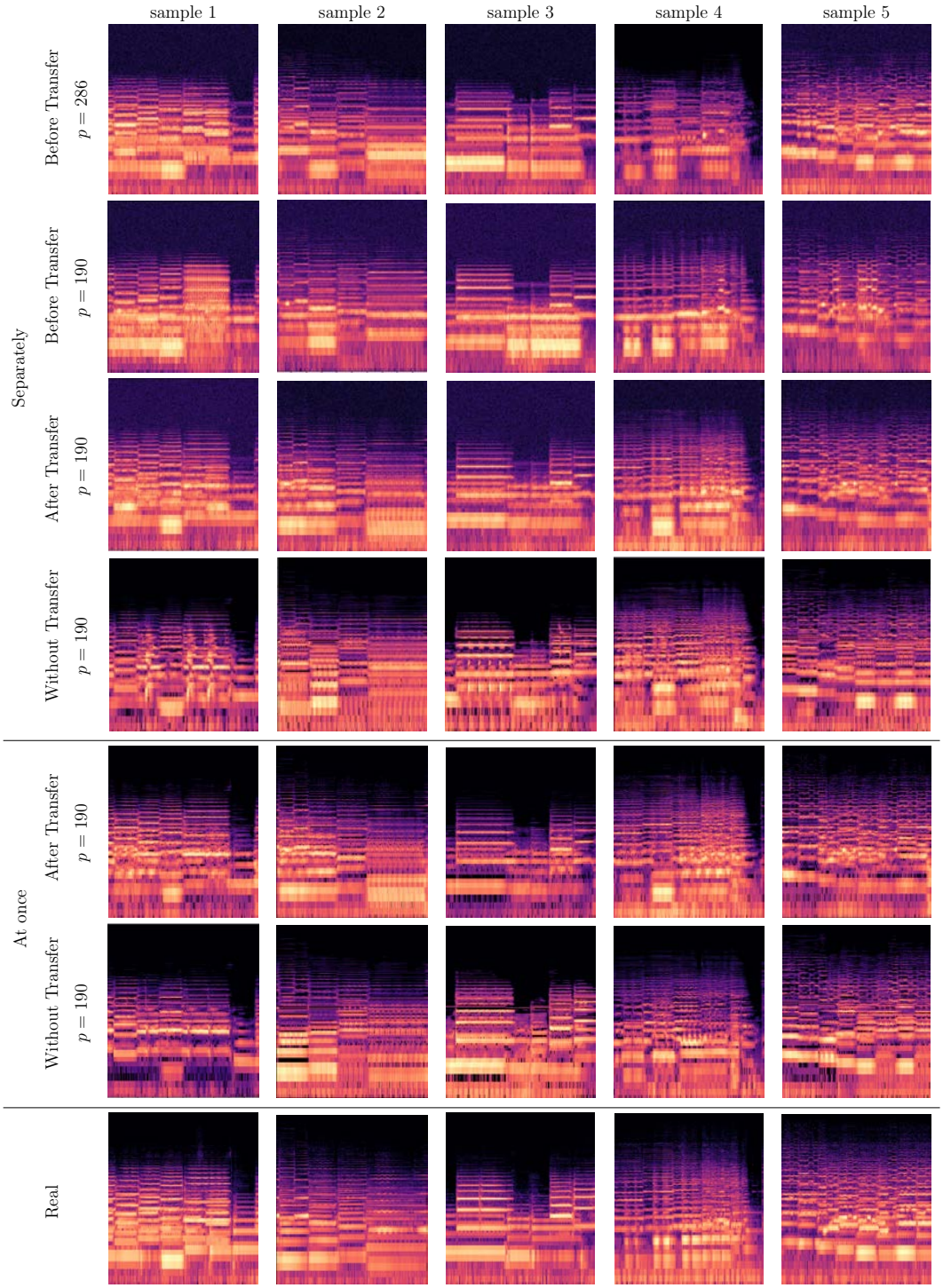


Figure 5.12: Multi-instrument samples that have been faked at once, i.e. by a model producing one spectrogram for all tracks (marked *At once*) versus a model sequentially producing one spectrogram per track (*Separately*). In the latter case, the resulting waveform audio is merged via addition.

model, the MS-SSIM variant with patch size $p = 190$. This results in a comparison of the strategies provided in Table 5.1. There is no *transfer inst-pure* model as it seems highly counterintuitive to omit a previously included instrument classifier at a later stage of training.

Model name	**Training on**	
	single-instrument samples	single- & multi-instrument samples
single-pure	no instrument classifier	-
single-inst	with instrument classifier	-
transfer-pure-pure	no instrument classifier	no instrument classifier
transfer-pure-inst	no instrument classifier	with instrument classifier
transfer-inst-inst	with instrument classifier	with instrument classifier
multi-pure	-	no instrument classifier
multi-inst	-	with instrument classifier

Table 5.1: Different variants of including an instrument classifier in the training of *orGAN*.

From a point of view focusing on the training process, all variants behave nearly identically in particular in terms of the L1 loss and the MS-SSIM and in case an instrument classifier is included it quickly approximates an AUC value of 1. However, there are noticeable differences in a qualitative visual analysis using Figure 5.13 and Figure 5.14: in the *multi* model, i.e. the one that is directly trained on single- and multi-instrument samples, the inclusion of an instrument classifier seems to make almost no differences in the quality of the faked spectrograms which look sharper than those from the *single* instrument models anyway. In this the single-instrument model, the samples generated with *single-inst* appear sharper with clearer distinctions between frequencies and more expressive timbre compared to the *single-pure* variant. This also coincides with the acoustic perception. The improvements seem very similar to the effects multi-instrument training had on the quality of single samples (see Section 5.3, in particular Figure 5.11). Hence it can already be stated that for the single-instrument models an instrument classifier fosters training as it has the same effect as additional training data. Looking at the multi-instrument samples in Figure 5.14, results are visually very close while in the waveform audio those from the *transfer-pure-inst* model sound a little more natural and enable clearer distinctions between instruments compared to the *transfer-pure-pure* variant. Interestingly, for the *transfer-inst-inst* model the measures also indicate a training progress almost identical to the other *transfer* variants but the produced samples contain many artifacts are structural errors. Intuitively, it can be conjectured that as the instrument classifier is trained on more data it becomes so strong that its relative influence on the generator's compound loss function exceeds the one of MS-SSIM

so that the model overlearns timbre and micro structure at the cost of structural coherence. This appears in particular reasonable, as one can observe similar effects in the Vanilla GAN (see Section 5.1, in particular Figure 5.4) where there is only a penalty on textural loss. In parts this can also be observed when comparing the *multi-pure* to the *multi-inst* model. Nevertheless it requires further experiments such as using adaptive weights in the compound loss function (Section 5.1) following an adequate training schedule as it is often done for the learning rate of an optimizer (see Section 4.2.1).

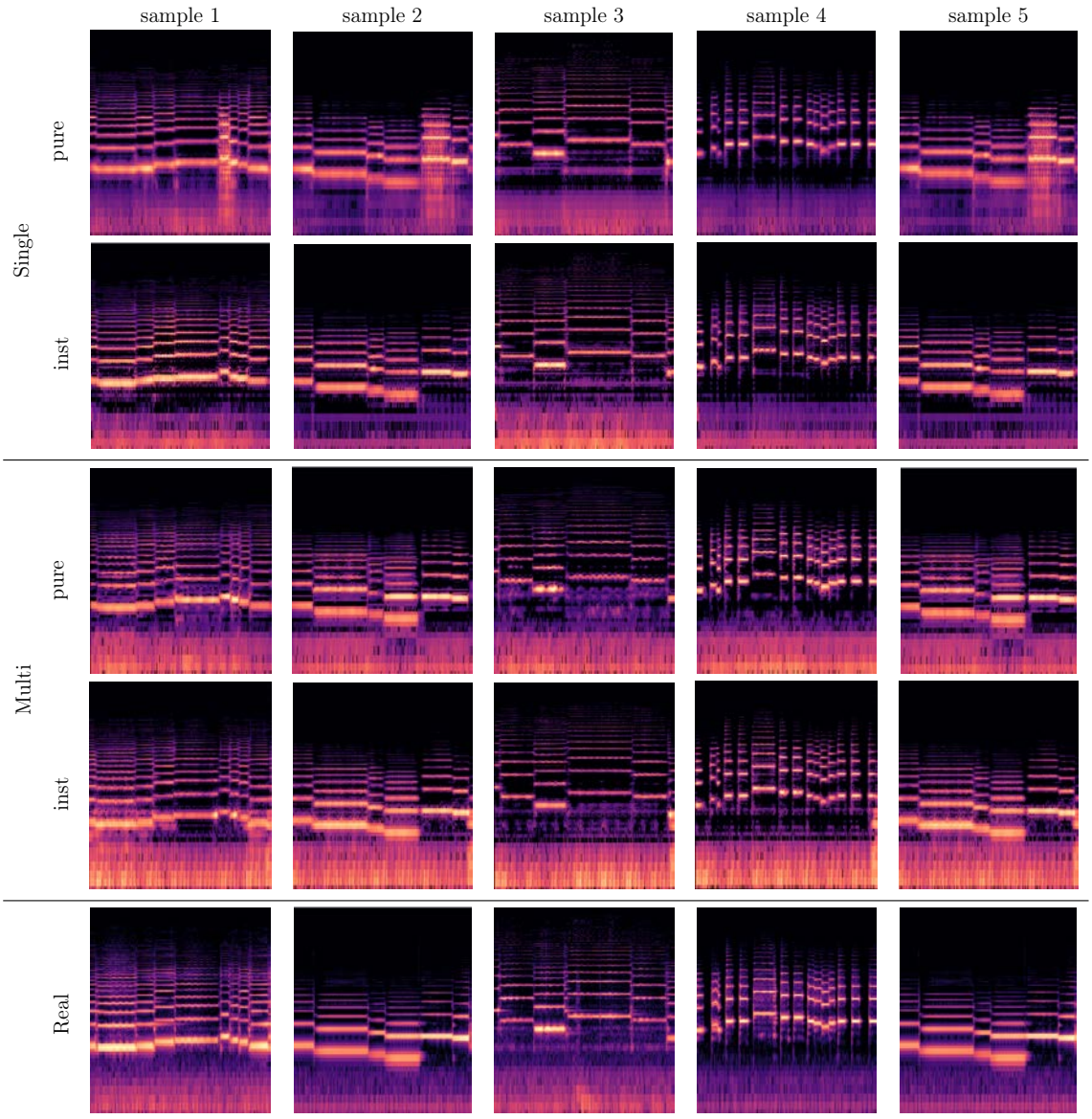


Figure 5.13: Single-instrument samples for different inclusion strategies for an instrument classifier (see Table 5.1): in the single-instrument model, the classifier clearly enhances timbre while reducing blurriness while in the multi-instrument model without transfer learning, no major effects stand out.

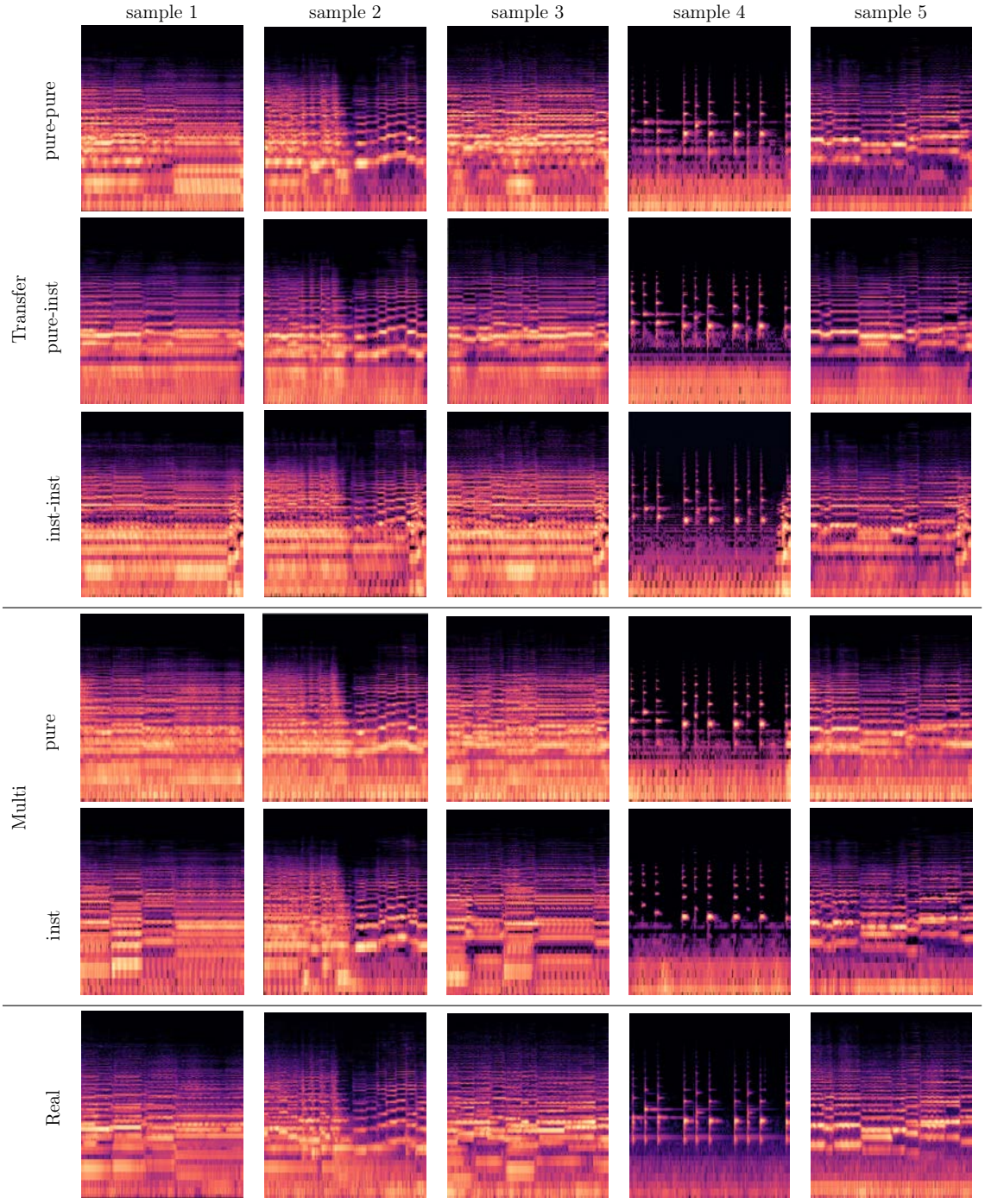


Figure 5.14: Multi-instrument samples for different inclusion strategies for an instrument classifier (see Table 5.1): for *pure-inst* and *pure-pure* the samples are visually very close while first performs better in the acoustic perception. In contrast, the *inst-inst* model also captures the overall structure of each sample well but shows more fine-grain recurring artifacts. Similar can be observed for the *multi* instrument model without transfer learning.

Chapter 6

Evaluation

6.1 Human Perception

The Test Design

Following common practice in the field [17, 72, 12], the gold standard for evaluating this work is human judgment. For this, **486 individuals** recruited from `Amazon Mechanical Turk` as well as from the author's social network are presented samples from the previously held out test set (see Section 3.1). The participants are requested to provide relevant discretized meta data without revealing their identity:

Item	Options
Age	`< 18 | 18-25 | 26-40 | 41-50 | 51-60 | > 60`
Gender	`male | female | diverse`
Student Status	`Undergrad. Student | Graduate Student | neither`
Musical Experience	`Professional | Hobby | neither`
Use of Headphone	`True | False`

Table 6.1: Meta-data items and available values requested from the participants in the human evaluation process.

While the student status is requested mainly for statistical purposes and to filter out cheaters via implausible combinations[1], the items age, gender and musical experience as well as the usage of headphones during the evaluation can be assumed to exert strong relevant influence on the ability to perceive details of music and to identify types of instruments in it. There are no restrictions applied to the group of participants i.e. in particular it is not a representative group.

[1] for instance, if a user selects *undergraduate student* together with *age* $>$ *60* and *professional* musical experience, he or she is likely to select random answers throughout the survey.

Each participant is presented 30 uniform-randomly selected samples and listens to them one after another. Right after listening, each sample is rated on a 5-point Likert scale, i.e. on the choice set

`strongly disagree | disagree | neutral | agree | strongly agree`

for each of the following statements:

(i) The music sounds like generated by human rather than by machine. (Naturalness)

(ii) The timbre sounds as of real instruments. (Timbre)

(iii) The audio quality is good. (Quality)

(iv) The music expresses emotion. (Emotion)

(v) The music sounds good overall. (Overall)

Those items are presented in the same order for each sample. The so far test design matches the most related work, namely [72], to enable a direct comparison.

In parallel, for each sample the participants are asked to select all instruments they think to be able to identify in the given piece of music. For this, binary toggles are presented for all of the instruments occurring in the URMP-Dataset (listed in Section 3.1) grouped by instrument class (i.e. Strings, Woodwinds and Brass) as depicted in Figure 6.1. This selection panel is the same for all samples. The participant does not know about the true number of instruments playing in this sample, i.e. how many instruments to select. The survey design deliberately does not challenge the participants to guess the correct number of instances of the same instrument to simplify the task in favor of more precise feedback. As acoustic instrument classification in general can be assumed too be a very hard task at least for non-professional humans, this in particular necessitates a comparison against recordings from real instruments in order to retrieve some sort of "base error". Once a user has submitted both, Likert scale rating and instrumental classification, for one sample, he or she is not allowed to make modifications to them anymore.

Conducting the Test

The survey is conducted over 150 randomly selected test set samples for each model configuration that is evaluated. Note that the sets of samples for each model can be disjoint but do not have to. The samples of all models are shuffled all together. To compare the different configurations of *orGAN* against conventional approaches, also samples generated from two off-the-shelf synthesizers, `musescore 3`[2] and Ap-

[2] `https://musescore.org`

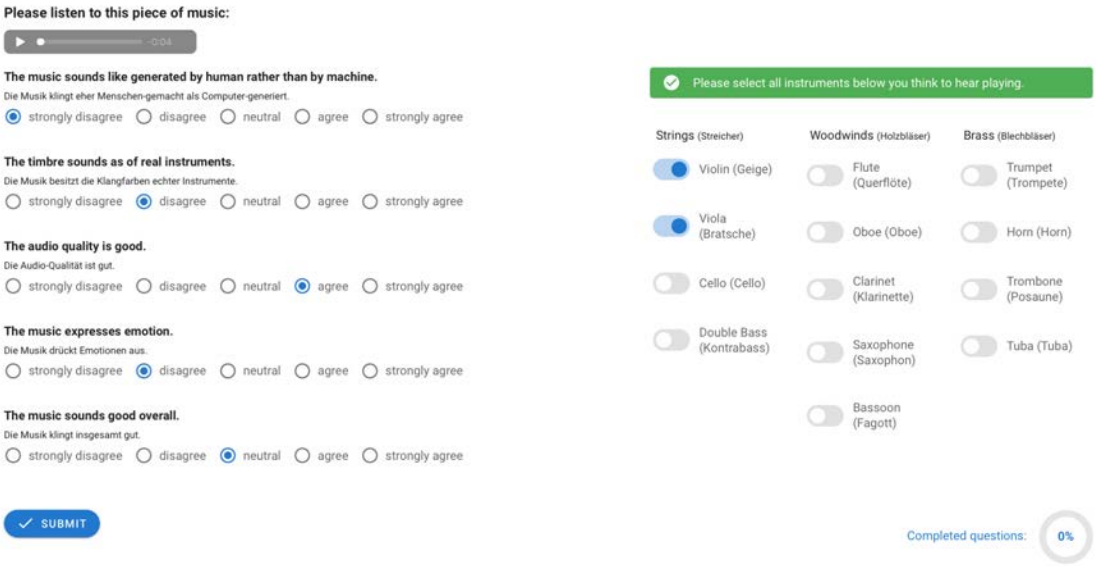


Figure 6.1: The user interface of `audival` with 5-point Likert scales for all criteria (left) a multi-instrument selection panel (right).

ple's state of the art `Logic Pro X`[3] generated from MIDI data from the test set are included in the same way. This is also done in the evaluation of PerformanceNet [72] enabling straight-forward comparison of the results. Note that when applying those synthesizers just the MIDI data is imported and then synthesized with the appropriate instruments. In particular, no effects are applied and there is no manual fine-tuning. To have a solid baseline for interpretations, the pool of samples to evaluate also contains 150 real recordings from the test set split from the URMP-Dataset. In total, the **486 participants** submit **10.067 ratings**[4] for 11 model configurations including synthesizers and the real sample collection. This results in an average of ≈ 6.10 ratings per sample, i.e. $\approx$ **915 ratings per model** configuration. This is on the same quantitative level as the human evaluation of Google's GANSynth [17]. The distribution of the collected metadata of the participants is provided in Figure 6.2.

The whole survey is conducted using the web application `audival`[5] which has been created exclusively for this work by the author. Special features and advantages of `audival` over common survey platforms are that it is tailored to this task, free, enables anonymous usage and allows full customization as well as unlimited access

[3] `https://www.apple.com/logic-pro/`

[4] Not all participants finished the survey and thus there are users who have committed less than 30 ratings.

[5] `https://audival.io`

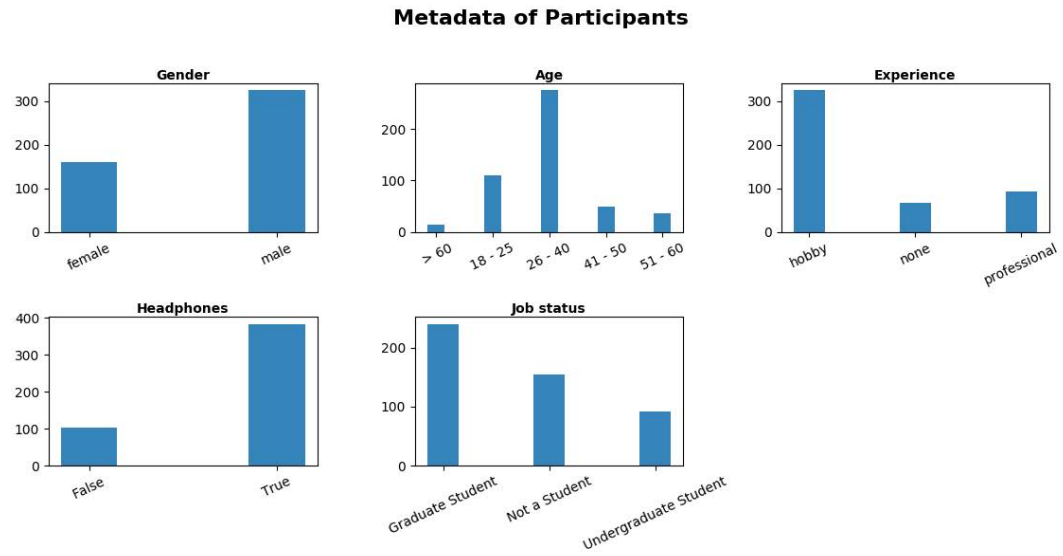


Figure 6.2: The distributions of the collected metadata of all individuals in the human evaluation process over the items described in Table 6.1.

to all raw data. Further, it comes with a yet very basic but nonetheless effective "fake detection" mechanism by automatically rejecting ratings from users who always provide the same answer for at least one type of question as well as ratings for which the user invested less time than the duration of an audio samples (i.e. in this case the user did not even listen to the sample he or she rated). Yet, the implementation of this tool itself shall not be considered part of this thesis.

Results for Perception

For this evaluation, the *orGAN* variants are selected that seemed to performed best in the experiments of Chapter 5. The average results from the Likert-scale ratings are summarized in Table 6.2: it can be seen that the *orGAN* models outperform state of the art synthesizers[6] in terms of realistic timbre, emotional expressiveness and naturalness of their generated music. However, this comes at the expense of quality mainly caused by small artifacts which obviously do not occur in deterministic hand-crafted synthesizers. This also highly correlates with the rating of the overall perception quality. Nevertheless, the two *orGAN* models without transfer learning are able to outperform the synthesizers even in this category, yet by a very small margin. Tackling quality issues is hard as even tiny structural errors on pixel level in the spectrogram can cause a major loss of audio quality. Most applications of generative models focus on image processing and computer vision and are not very

[6] used in basic configuration, i.e. without manual fine-tuning

	Perception				
Model	Timbre	Naturalness	Emotion	Quality	Overall
single-pure-190	2.44	2.34	2.31	2.20	2.24
single-inst-190	2.57	2.45	2.37	2.34	2.38
single-pure-94	2.45	2.38	2.26	2.29	2.28
single-pure-286	2.59	2.40	2.38	2.32	2.36
transfer-pure-pure-190	2.51	2.41	2.40	2.27	2.27
transfer-pure-inst-190	2.58	**2.47**	2.46	2.29	2.41
multi-pure-190	2.59	**2.47**	**2.60**	2.53	2.51
multi-inst-190	**2.65**	2.43	2.58	2.49	**2.54**
logic pro x	2.50	2.31	2.37	2.54	2.43
musescore 3	2.45	2.33	2.45	**2.60**	2.51
real	2.82	2.69	2.64	2.61	2.66

Table 6.2: The average perception of 5s samples from different models on a scale from `0 = very bad` to `4 = very good`. The best performing fake model for each category is in bold typeface. The infix *inst* marks models with instrument classifier in contrast to *pure*. Accordingly, *pure-pure* and *pure-inst* performed transfer learning where one uses an instrument classifier in the second training stage i.e. on single- and multi-instrument samples. The prefix *multi* indicates that the model has been trained on single- and multi-instrument data right from the start instead of using transfer learning. The *single* models have only been trained on single-instrument samples. The number as postfix indicates the patch size used for the PatchGAN.

sensible to fine-grain perturbations and therefore improvements remain a challenge for future work.

Focusing not only on the top scores, but instead comparing the *orGAN* variants among each other, from Figure 6.3 one can see that the *single* and *transfer* models using an instrument classifier outperform their pure counterparts in almost all categories. Further, the single-instrument model with instrument classifier performs similar to the *transfer* model without. This confirms the conjecture in Section 5.4 that including an instrument classifier has the same effect as using more training data. However, this does not hold that clearly when the classifier is included from the very beginning of training: the *multi* models show significantly better performance than others in all categories, in particular in quality and emotion. For them, the inclusion of an instrument classifier seems to have no major effect. Regarding the patch size used in the discriminator, the human evaluation suggests superiority of the 286×286 variant over using the smaller patches sized 94×94 and 190×190. This is in slight contrast to the subjective analysis in Section 5.2.

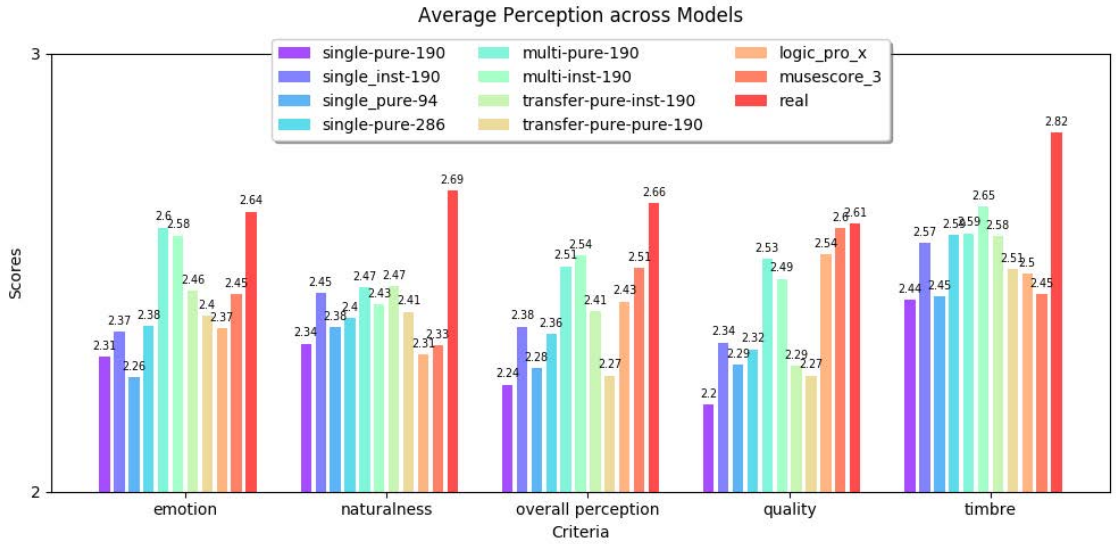


Figure 6.3: The average rating on a 5-point Likert-scale from `0 = very bad` to `4 = very good` for each model configuration and the off-the-shelf synthesizers `musescore 3` and `Logic Pro X` as well as for real recordings from the URMP-Dataset.

Apart from the average ratings over all users, differentiating among raters reveals interesting insights:[7]

first, Figure 6.4 shows that women distinguish better between models than men. One may note, that women also take much more time to rate one sample (see Figure 6.8). Further, from all models they favor the *orGAN multi-inst-190* while men assign slightly higher ratings to *orGAN single-inst-190* and the synthesizers.

Headphone usage has effects as well which are depicted in Figure 6.5: while the real samples are perceived similarly in both situations, the ratings of the fake samples are overall lower when listened to with headphones. It can be suspected that this is due to the reduced environmental noise letting musical impurities and artifacts to attract more attention. Also, headphone usage seems to encourage a clearer distinction between the different models and reveals quality lacks of *orGAN*, especially in comparison to the synthesizers. In particular, with headphones the differences in emotion, timbre and naturalness appear to become more clear so that in particular the margin between ratings of the *orGAN* models and synthesizers in those categories increases.

[7] As the group of raters is neither representative nor has an equal distribution of characteristics, the following results have to be interpreted carefully and are not statistically resilient.

Another influential factor on the perception of music is the age of the listener: as Figure 6.7 shows, the fakes are rated best in comparison to the real data by people over the age of 60. It is well known that the frequency range perceivable by the human ear decreases with increasing age. Therefore it can be conjectured, that in the mentioned group the high-frequency noise and perturbations which are characteristic for *orGAN*'s fakes are not perceived that prominently. In all other age groups, the real samples are rated best overall. While people between 51 and 60 prefer synthesizers over the *orGAN* models, the younger which also invest by far the most time in the rating process (Figure 6.8) assign them high naturalness, emotion and timbre quality. Also in the group aged 41 to 50, in particular the *multi* models perform well and are even rated better than the real data regarding emotional expressiveness. In this context, it should be considered that this group also is the one that invests the fewest time in the rating process. Meanwhile the largest group of participants aged 26 to 40 rates all models similarly which can be caused by the overproportional amount of data for this group.

One of the most interesting yet questionable results is yielded by the distribution of ratings across levels of musical experience visualized in Figure 6.6: from the collected data, it seems like the ability to distinguish real from fake samples decreases with an increasing level of proficiency. The data even indicates that professionals in the field of music (i.e. musicians, singers, producers, dancers and so on) perceive the *pure-inst-190* variant of *orGAN* as more realistic than the real samples. One may note that number of professionals taking part in the evaluation is close to the one of raters without experience in music (recall Figure 6.2) and thus the size of the test group cannot not directly provide an explanation here. One plausible suspicion is that "cheating" clickworkers who select answers in ratings and meta data questions randomly have a higher effect in the group of professionals which is expected to be relatively small when sampling randomly from the real world. Another conjecture is that with less musical experience, users more likely attempt to do a binary real/fake classification by selecting more "extrem" ratings and by this increase the margin. On the other hand, professionals might attempt to do a more fine-grain rating so that the mean rating of the real data is close to those of the fakes. Further, the level of profession significantly correlates with age and, as it has been described in the above paragraph, an increasing age reduces the ability to identify fakes via high-frequency anomalies and noise. In addition, professionals (and "cheaters" who pretend to be) take the fewest time per rating (see again Figure 6.8).

The effect of the job status (undergraduate, graduate, not a student) is not analyzed separately as this is not expected to excert relevant influence on the ratings and further it highly correlates with age anyway.

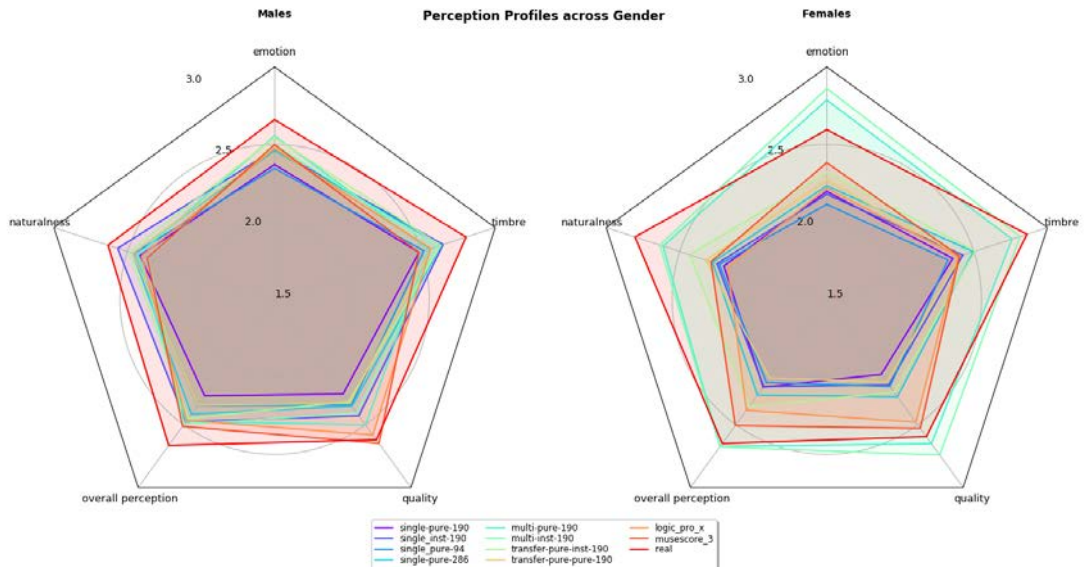


Figure 6.4: The average perception of different models across gender of the evaluating human.

The key insights so far in summary:

- *orGAN* outperforms synthesizers regarding emotion, timbre and naturalness.
- *orGAN* suffers from artifacts and accordingly bad audio quality.
- Without Headphones, artifacts in fakes attract less attention.
- Elderly people rarely detect *orGAN*'s fakes. Young and mid-aged humans can distinguish them well while *orGAN* wins over the synthesizers especially in the younger groups.
- It seems that with increasing musical experience the distinction between fake and real samples gets worse. However, this appears implausible and might be traced back to test effects.

Results for Classification

The instrument classification by the users yields some less informative yet interesting insights as well: an analysis of the confusion matrices for single samples via Figure 6.9 shows that recognizing instruments is a hard task for humans and there is a lot of confusion even for the real recordings. Hence, all the results have to be interpreted carefully. It can be seen that for the synthesizers as well as for the *orGAN* models all kinds of strings and even some winds are likely to be recognized simply

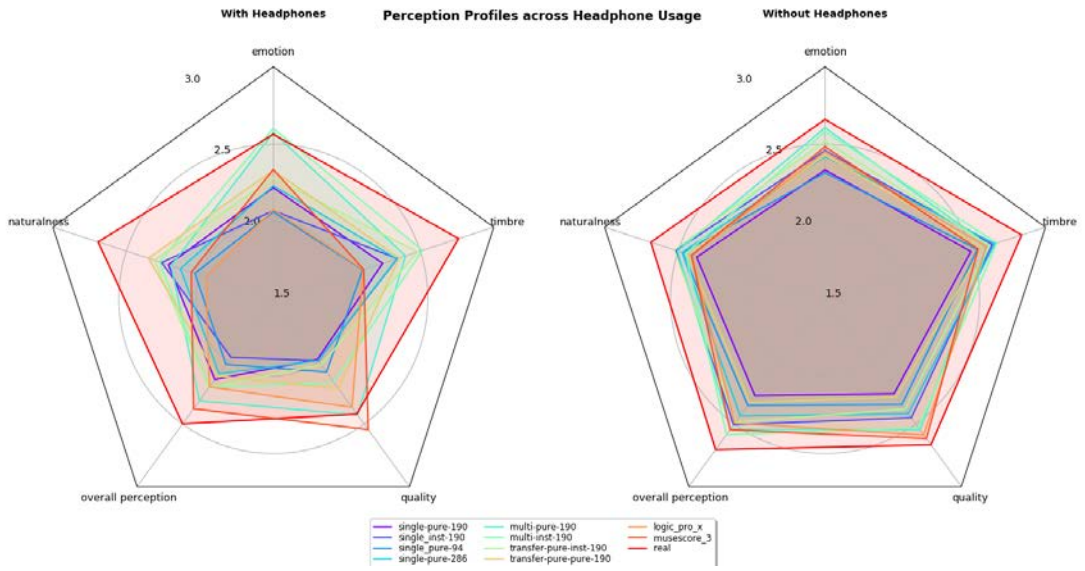


Figure 6.5: The average perception of different models across the usage of headphones during the evaluation.

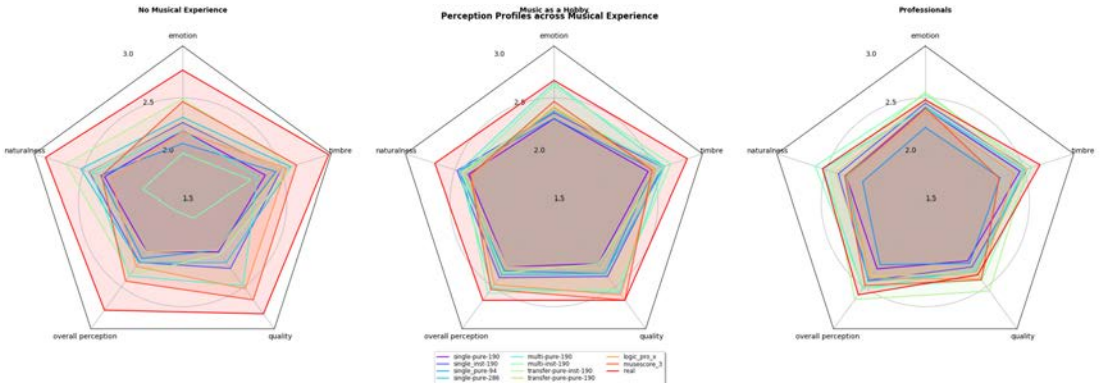


Figure 6.6: The average perception of different models across musical experience of the evaluating human.

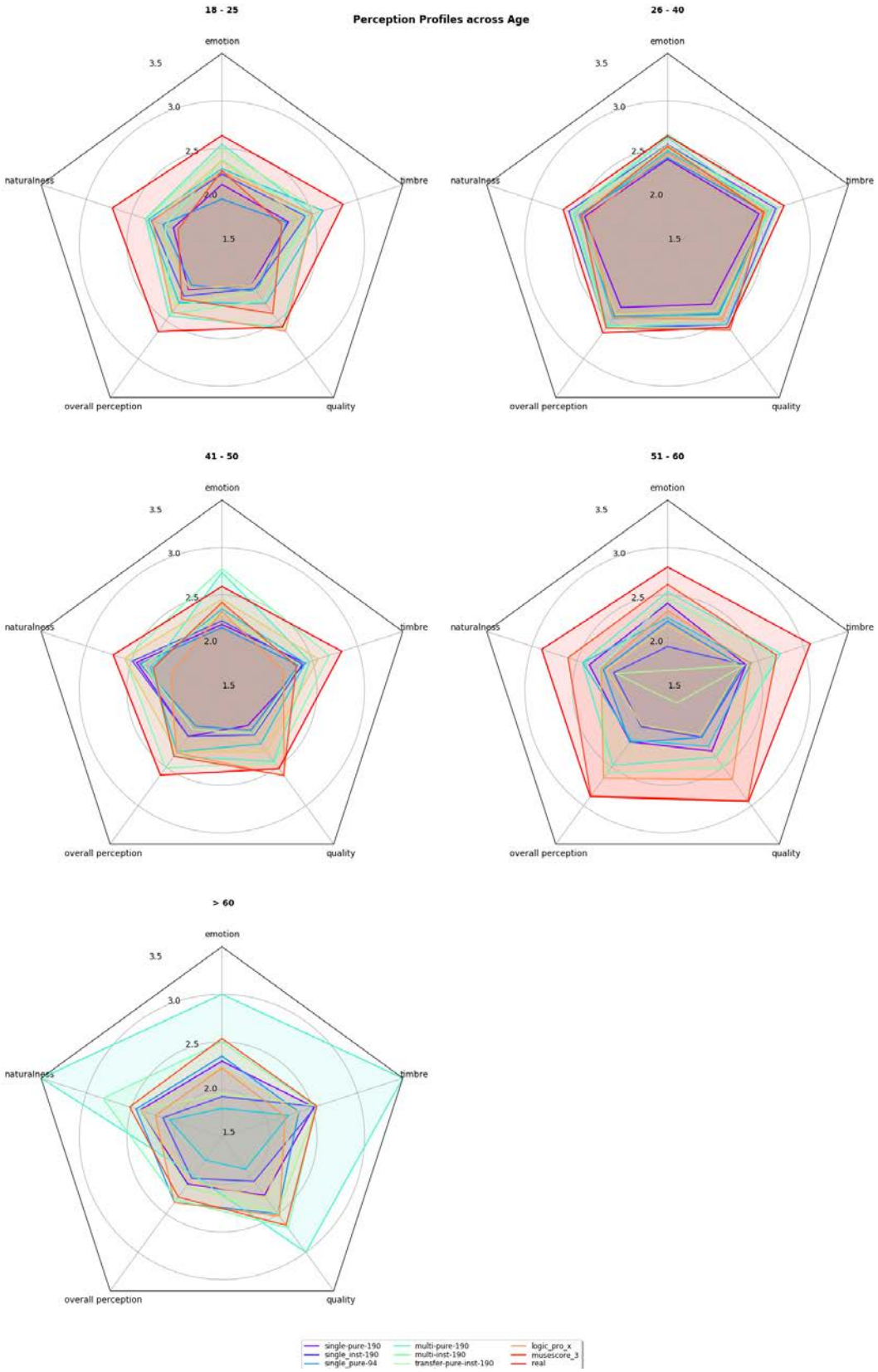


Figure 6.7: The average perception of different models across age.

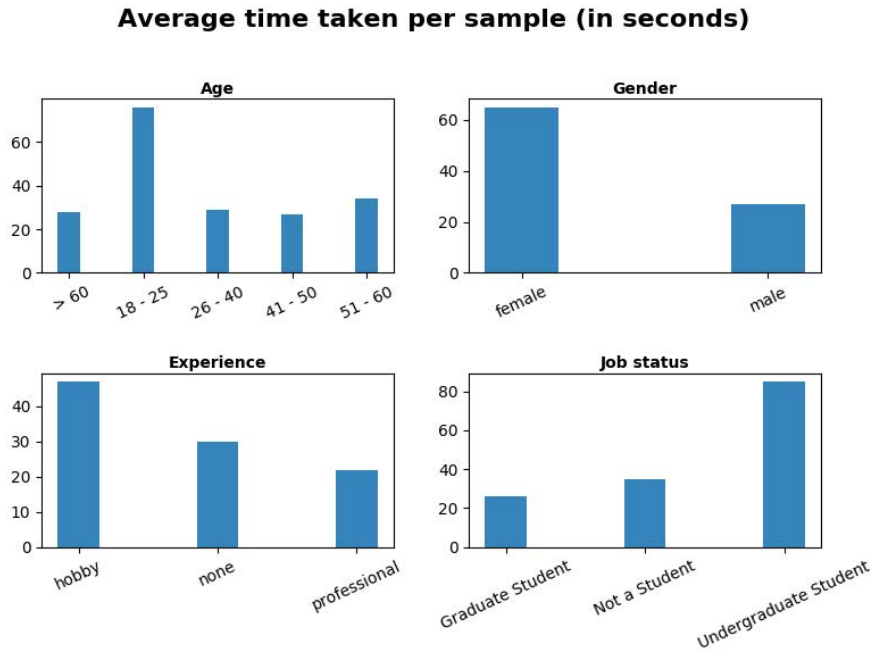


Figure 6.8: The average time in seconds taken for rating one sample.

as a violin. Nevertheless there are little differences and each model comes with its own strengths and drawbacks: the best distinction of violin and viola is achieved by *transfer-pure-pure-190*. For cello, *single-pure-190* outperforms all others. Double bass is faked best by *single-pure-286* while being often confused with other strings. For flute, again *transfer-pure-pure-190* wins even exceeding the results on the real data by far. Regarding all other instruments, the data does not allow meaningful conclusions beside that for all models it is hard to make clear distinctions between closely related instruments. Nevertheless it appears that for many instruments, *orGAN* is superior or at least equal to the synthesizers. Overall, from analyzing the single-instrument samples it seems that there is no clear "best choice" model for all instruments, but instead one has to differentiate. Note that the evaluation here does not necessarily include single samples of all instruments fore each model as the samples are selected randomly.

This is compensated by widening the view to more data and include multi-instrument samples (for which an instrument confusion matrix cannot be plotted easily): as the plots in Figure 6.10 and Figure 6.11 state, for most instruments there is an *orGAN* model roughly en pour with or outperforming the synthesizers. Considering the mean portion of correct classifications over all instruments for each model in Fig-

ure 6.10, many models perform better than the real data[8] where the *orGAN* variants *single-inst-190* and *single-pure-190* perform best. Using Figure 6.11 to compare the mean recognition rate of each instrument to the performance of each model confirms that *orGAN* is indeed superior in most strings, but does not exceed synthesizers in winds such as trumpet, tuba, oboe and bassoon. Nevertheless, recognition rates for all instruments and models are rather unreliable and at an overall very similar level so that this results should not be payed too much attention.

Comparison with existing Models

The design of the human evaluation process described above allows to directly compare the results to those of PerformanceNet [72] and - as it is in included there – to a WaveNet [52] modification proposed by [42]. Relying on the data provided in [72], the comparison to PerformanceNet can only be done on the results for samples from cello, violin and flute and to compare both to WaveNet, only cello can be used. Those are compared to the *orGAN* models with and without transfer learning according to Figure 6.3, namely *transfer-pure-inst-190* and *multi-pure-190*, as well as the worst one: *pure-190*. After converting the results of the other models from a $[1, 5]$ Likert-scale to the range $[0, 4]$, a direct comparison in Figure 6.12 reveals the superiority of all *orGAN* variants over PerformanceNet. While being almost en pour in naturalness, there are clear improvements in emotional expressiveness, timbre, overall perception and in particular in quality. Also when putting the results for cello next to those of the WaveNet based model, *orGAN* again performs best by far as depicted in Figure 6.13. For those cello samples, the *pure-190* model even reaches the level of real data and in the criterion "emotion" even exceeds them. The latter which can also be observed in the previously analyzed Figure 6.11 appears implausible at a first glance and might be traced back to test effects such as the samples containing a more "catchy" melody than others which causes a higher rating in "emotion". Nevertheless, it can be said that the *orGAN* models exceed the standard set by related work.

6.2 Internal Comparison

To conduct a straight-forward "internal" comparison of the best performing *orGAN* models described in Chapter 5, some of the performance measures described in Section 4.7 for the samples of the previously held out test set are reported and analyzed: first of all, the top MS-SSIM values achieved by each model are considered as well as the L1 loss between real and fake spectrograms, which has not been part of the objective of the models investigated here. Related work such as GANSynth utilizes a separately trained pitch classifier to evaluate the quality of their fakes. Following

[8] which again shows the questionable reliability of human instrument classification

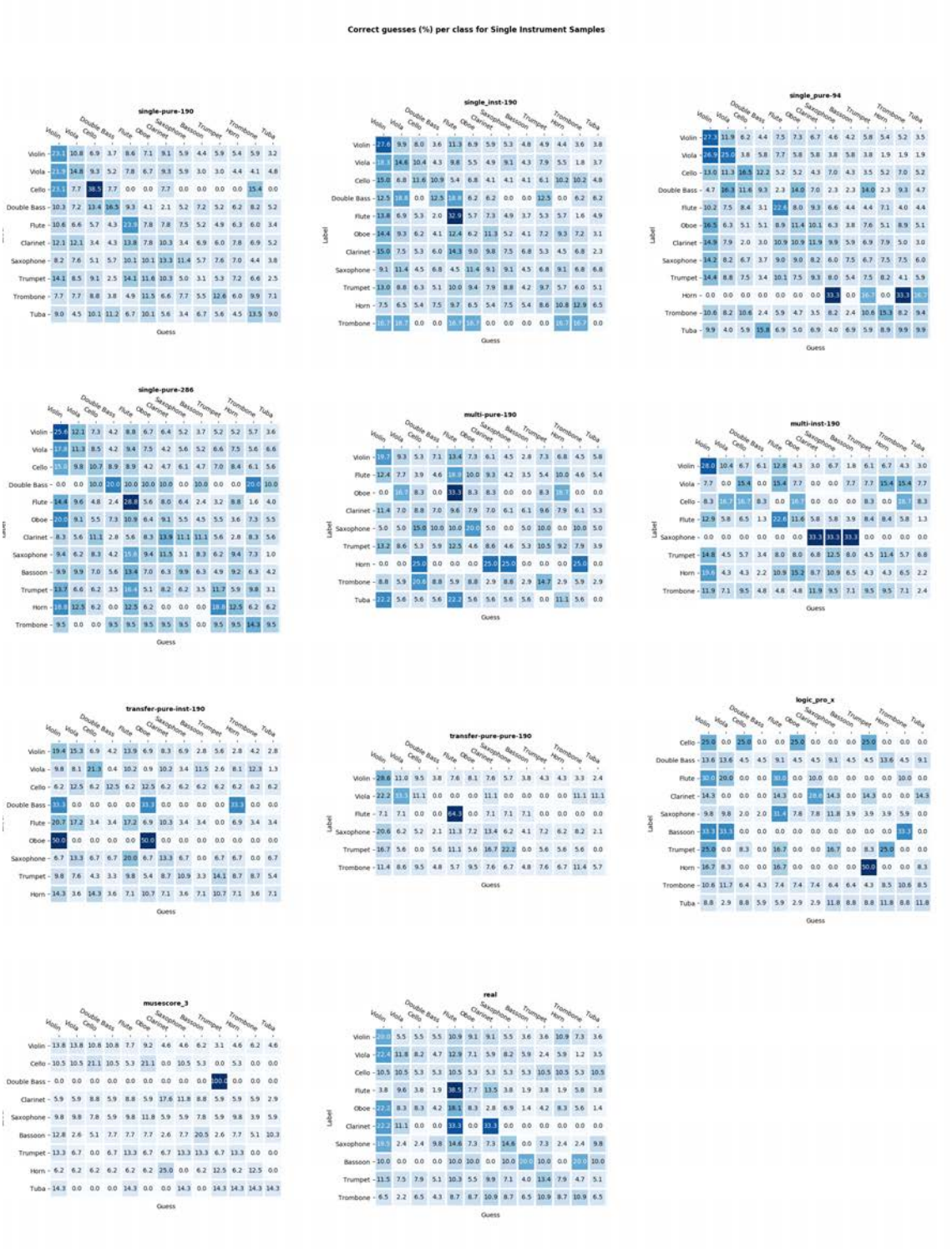


Figure 6.9: The confusion in human instrument classification for different models over single-instrument samples only. Each row in a subplot is the distribution of guesses over all samples from one instrument in the associated model.

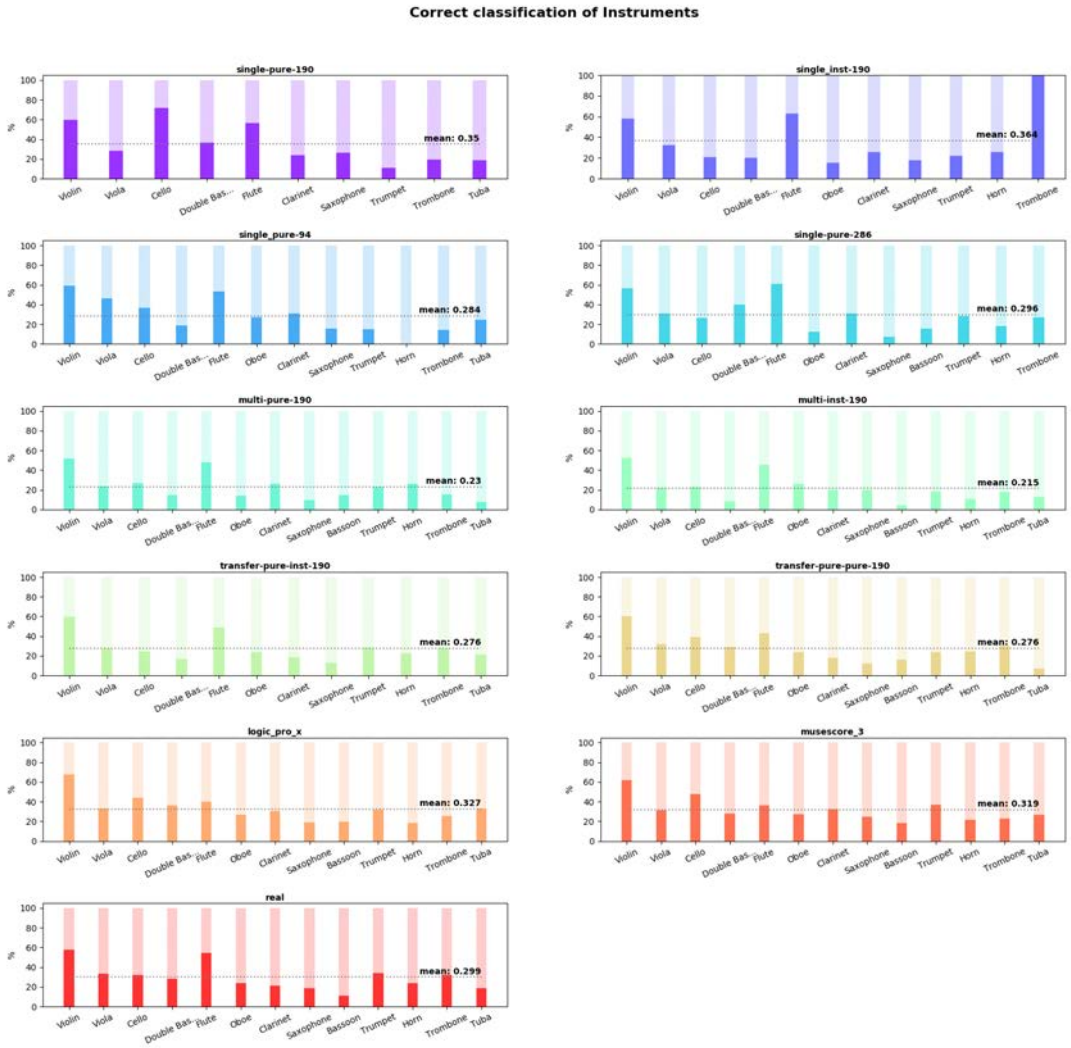


Figure 6.10: The portion of correctly recognized samples per model.

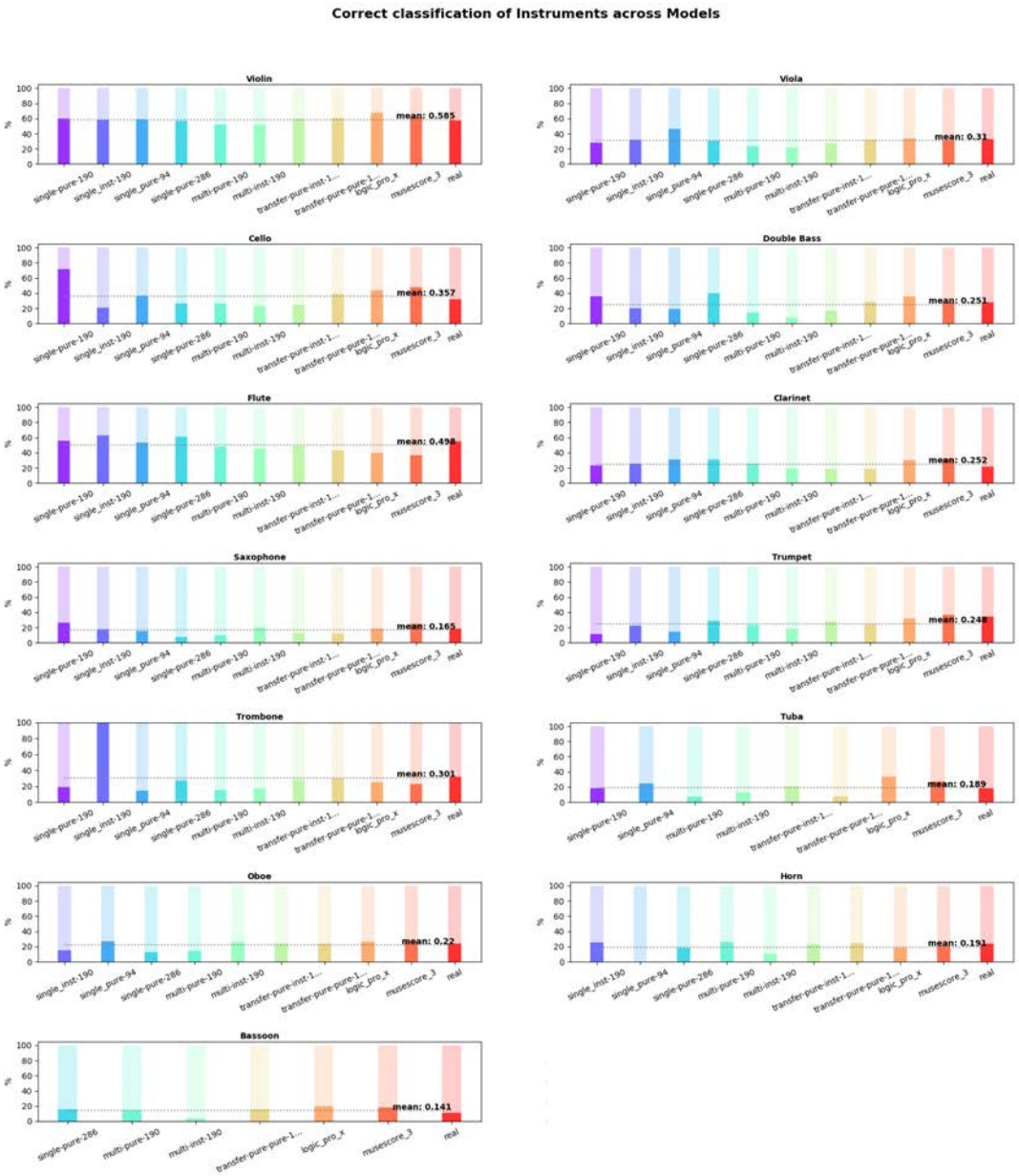


Figure 6.11: The portion of correctly recognized samples per instrument.

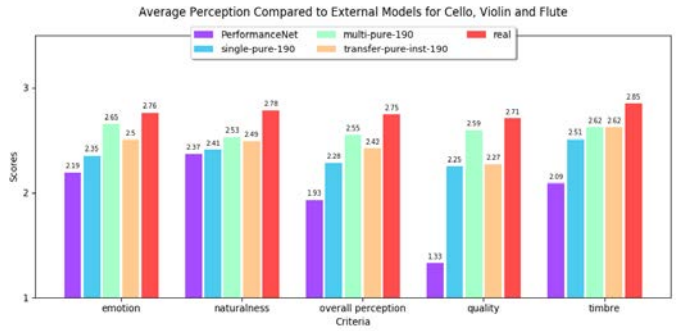


Figure 6.12: Perceptual comparison with PerformanceNet [72].

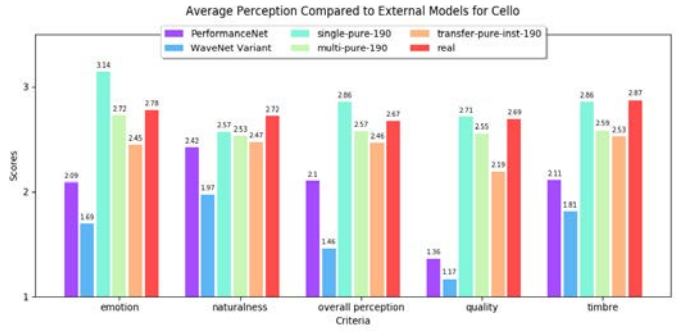


Figure 6.13: Perceptual comparison with PerformanceNet [72] and a WaveNet-based model by [42]

this idea, also the micro- and macro-accuracy as well as the area under of curve of the trained instrument classifier, which is part of the discriminator for some *orGAN* variants, is reported. This approach appears in particular justified as this classifier has only been trained on the real samples and therefore is not tailored to the fake output. Note that of course the same trained classifier, namely the one of the *transfer-pure-inst-190* model, is used for all comparisons here.

Another similarity measure between discrete probability distributions is the *Wasserstein Distance* [58]:

Definition 49 (Wasserstein Distance)
Given two discrete probability distributions represented as equally large finite sets of points $X := \{X_i\}_{i\in I}, Y := \{Y_i\}_{i\in I} \subset \mathbb{R}^d, I \subset \mathbb{N}$, the *Wasserstein Distance* is

$$W(X,Y) := \sqrt{\min_{\sigma} \sum_{i\in I} \left\| X_i - Y_{\sigma(i)} \right\|^2} \tag{6.1}$$

where σ is any permutation of $N := |I|$ elements. □

This has the nice interpretation of, loosely speaking, measuring the minimum effort required to transform X into Y respecting that any permutation of the data points does not affect the overall distribution. Because of its metaphorical formulation as the effort of transforming a pile of soil within a metric space into another one measured in the distance it has to be moved, this is often referred to as *Earth Mover's Distance*. The problem of finding the minimum in expression (6.1) can be solved for instance with linear programming algorithms for small N [58]. For scalability, [58] proposes an approximation of the Wasserstein Distance by projecting each data point onto the unit sphere:

Definition 50 (Sliced Wasserstein Distance)
Given two discrete probability distributions represented as equally large finite sets of points $X := \{X_i\}_{i\in I}, Y := \{Y_i\}_{i\in I} \subset \mathbb{R}^d, I \subset \mathbb{N}$, the *Sliced Wasserstein Distance (SWD)* is

$$\widetilde{W}(X,Y) := \sqrt{\int_{\theta\in\Omega} W(X_\theta, Y_\theta)^2 d\theta}, \quad X_\theta := \{\langle X_i, \theta\rangle\}_{i\in I} \tag{6.2}$$

with the unit sphere $\Omega := \{\theta \in \mathbb{R}^d \,\big|\, \|\theta\| = 1\}$. □

Above, $\langle\cdot,\cdot\rangle$ denotes the inner product. Effectively, with exploiting bilinearity this

	Test Results					
Model	MS-SSIM	SWD	L1	AUC	microACC	macroACC
single-pure-190	**.897**	.235	.023	.995	.869	.829
single-inst-190	**.897**	.339	.021	.998	.875	.849
single-pure-94	.861	.342	**.019**	.998	.870	.806
single-pure-286	.831	.354	**.019**	**.999**	.873	.875
transfer-pure-pure-190	.894	.296	.037	.995	.876	.789
transfer-pure-inst-190	.877	.233	.038	.989	.873	.914
multi-pure-190	.826	**.175**	.044	.990	.876	.861
multi-inst-190	.873	.237	.049	.996	**.894**	**.947**

Table 6.3: Internal tests results for different models: the MS-SSIM, the SWD and the L1-distance between real and fake samples as well as the outcome of feeding the fake samples to the instrument classifier trained along with the *transfer-pure-inst-190* model in terms of the AUC, the micro-accuracy and the macro-accuracy (see Section 4.7 for a clarification on these terms).

results in

$$\widetilde{W}(X,Y) = \sqrt{\int_{\theta\in\Omega} \min_{\sigma} \sum_{i\in I} |\langle X_i,\theta\rangle - \langle Y_i,\theta\rangle|^2 d\theta}$$
$$= \sqrt{\int_{\theta\in\Omega} \min_{\sigma} \sum_{i\in I} |\langle X_i - Y_{\sigma(i)},\theta\rangle|^2 d\theta}$$

In the evaluation of ProgressiveGAN [28], the SWD is computed between real and fake samples at different levels of perception: similar to the approach of the MS-SSIM (see Section 4.7), the image is downsampled iteratively by a factor of two until a size of 16×16 is reached.[9] At each level, random patches are extracted from the real and fake image on which the SWD is calculated and reported for each level separately.

In this work, the same technique is used to compare the different model variants of *orGAN* described in the previous sections in Chapter 5. For compatibility with the factor 2 downsampling, the spectrograms of size 935×1025 are padded with zeros in time and the upper most frequency band which can be considered negligible for this purpose is omitted to obtain a shape of 1024×1024.

A summary of all the measured results is provided in Table 6.3: it can be seen that the two *single*-instrument models with patch size 190 achieve a higher MS-SSIM value than the ones with $p = 94$ and $p = 286$. This is coherent to the findings

[9] This forms a *Laplacian Pyramid* [9].

of Section 5.2. Further, in terms of the MS-SSIM they perform better than their counterparts that are also trained on multi-instrument samples, i.e. *transfer-** and *multi-**. This is probably due to their task being just easier or more precisely: restricting the data distribution to single-instrument samples makes it easier to model. Regarding the SWD between real and fake spectrograms, the *multi-inst-190* model performs best by far, which coincides with the perceptual evaluation above. The L1 distance does not reflect this: here, the *multi-** models are actually the worst performing ones. This can be explained with the *multi* instrument models being more prawn to small structural errors due to the less clear distinction between single- and multi-instrument play despite their superiority in overall perceived quality (look up Section 5.3). Further, when considering those results it should be taken into account that the L1 metric is not sensitive to spatial features as described previously in Section 4.7, in particular in Figure 4.21. When analyzing the results from the instrument classifier, that has been trained along with the *transfer-pure-inst-190* model on the real world data, it can be seen that all models achieve a very similar mirco-accuracy lead by the *multi-inst-190* model. This appears reasonable, as this model has the most of training time with multi-instrument samples and an instrument classifier (which partially even seems to lead to an overlearning of timbre as mentioned previously). The results from the AUC are very inconclusive. The most meaningful insights can be obtained from the macro-accuracy, i.e. the portion of samples, where all playing instruments have been recognized correctly. The results show that throughout all models, the variant with instrument classifier performs better in terms of this measure than its *pure* counterpart. The top score is again achieved by the *multi-inst-190* model which can be justified the same way as for the micro-accuracy.

Chapter 7

Conclusion

In this thesis, based on a Generative Adversarial Network an artificial intelligence model for multi-instrument audio synthesis termed *orGAN* has been designed, built and analyzed:

Provided a stack of pianoroll representations of scores for different instruments where the position within the stack indicates the type of instrument, the model can generate a spectrogram which can be post-processed using the algorithm of Griffin and Lim to obtain waveform audio that mimics a real instrument recording. The model has full control over all musical parameters except timing and therefore takes the role of a human performer capable of *interpreting* scores. Learning the timing aspect is not enabled by design as for training frame-level annotations are required. Therefore, this may be approached by different models. The model is trained on high-frequency data from 13 instruments at once covering a broad range of strings, brass and woodwinds. For this, the URMP-Dataset is used from which 5s chunks of 48kHz audio are sampled. This includes single-instrument samples as well as all possible multi-instrument compositions. The pre- and post-processing has been described in detail together with the needed basics of singal processing. Also drawbacks of the pianoroll in the form of ambiguities regarding the number of instruments, note separation and on-/offsets have been outlined along with its advantage of incorporating useful spatial information.

The model solves a contour-to-image mapping task together with a super-resolution problem as a pianoroll is much smaller than its corresponding spectrogram. Therefore, the GAN utilizes well-known architectures from the field of image processing. In particular, it follows the pix2pix model and uses a U-Net autoencoder as generator conditioned by a pianoroll as input. It is noteworthy that with spectrograms of size 935×1025, *orGAN* operates on much larger "images" than most generative models so far. The discriminator is a PatchGAN together with an auxiliary instrument classifier. The objective function is comprised of the loss of the adversary, the L1 distance as well as the multi-scale structural similarity between real and fake spectro-

grams and the loss of the instrument classifier. As the model is fully-convolutional, it can cope with scores and audio of arbitrary length. In the core of this thesis, the methodological foundations of machine learning tasks, their learning process that uses a gradient descent algorithm, the concept of neural networks as well as basic regularization techniques have been described in depth together with special architectural styles of neural networks such as convolutional layers and autoencoders. A focus has been set on the concept of a generative adversarial network and its variants including cGAN, AC-GAN and PatchGAN. After laying out these concepts step-by-step, building up on this the architecture of *orGAN* has been described. Motivated by the mean absolute error's missing sensitivity for spatial information, the multi-scale structural similarity index has been established as another component of the network's loss function.

Further, a significant number of experiments has been conducted with different variations of *orGAN*: it has been shown that incorporating the L1 loss in the generator's objective speeds up training and reduces blurriness in the faked spectrograms. Using the MS-SSIM instead lead to further improvements, especially in high-frequency details. Both variants outperformed a "vanilla" GAN and a stand-alone generator optimizing the MS-SSIM. The first has been found to lack global coherence while the latter produced samples missing fine-grain local structure. Therefore, it has been shown that the adversarial setting is able to eliminate the need for dedicated refinement architectures like the residual sub-network and the on-/offset encoder in PerformanceNet. Investigations regarding the effect of different patch sizes for the PatchGAN classifier largely confirmed the findings of pix2pix: small patches sized 94×94 are likely to cause tiling artifacts while larger choices of 190×190 and 286×286 perform better on a visually similar level. After starting with models trained on single-instrument samples, the application of transfer learning for multi-instrument recordings has been explored so that the model has been given control over interactions between instruments: while all *orGAN* variants struggle, the variant with patch size 190×190 performed remarkably better than the others. Nevertheless, is did not achieve the same audio quality as faking each instrument separately with the same model.[1] It has been observed that after transfer learning including multi-instrument samples, also the quality of single-instrument audio improved which is probably due to the additional amount of training data. The same effect has been observed for a model trained on both, single- and multi-instrument samples, together from the beginning. This way, samples also achieve a much better quality and have more clear timbres. It also turned out that the usage of an instrument classifier in the discriminator which is trained on real data only fosters training in the same way as more training data does, i.e. making timbre more clear and increasing contrast between frequency bands. This turned out to beneficial mainly in

[1] Improving this in future would in particular address the necessity of applying Griffin-Lim multiple times which slows down the generation process in production dramatically.

transfer learning, a usage across all phases of training yielded less promising results. The latter is conjectured to be due to a too large influence of the instrument classifier in the compound loss of the generator. It is up to further research to tackle this e.g. via adaptive weighting of the instrument classifier loss over training.

For a perceptual analysis, all *orGAN* models have been involved in extensive human evaluation: in ratings for naturalness, timbre, emotion, quality and overall perception of the generated samples together with human instrument classification, *orGAN* outperformed state of the art synthesizers in the first three categories. Quality and overall perception are still largely affected by occasionally occurring artifacts preventing the models to compete with deterministic synthesis strategies. Making improvements here is the most important challenge open for further work. In contrast, the results from human instrument classification are rather inconclusive as this is a challenging task even for professionals so that human raters barely deliver reliable data. Nevertheless it has become clear that there is not "the one" *orGAN* variant which is best for synthesizing all instruments. The *orGAN* variants which still turned out to perform best overall, marked *transfer-pure-inst-190* and *multi-pure-190* both use a patch size of 190×190 and incorporate the MS-SSIM in their objective, but only one applies transfer learning with an instrument classifier included in the second stage while the other one is trained on all samples from the beginning of training. Both outperform PerformanceNet as well as an appropriately modified WaveNet by far. The interested reader is highly recommended to visit `https://students.fim.uni-passau.de/~susetzky/organ/` to explore some audio samples generated with *orGAN*.

Considering all of the above, it can be said clearly that Generative Adversarial Networks are applicable for realistic high-quality multi-instrument music synthesis.

Finally, a few ideas for future work shall be formulated briefly: first, as mentioned above, it remains open to learn the timing aspect and improve audio quality by tracing back and removing sources of artifacts. It would also be possible to include further conditions and contextual information in the generator such as genre, composer or musical epoch and then vary these parameters for a given score after training. Further, *orGAN* is designed to be extensible for learning new instruments. Investigating this ability of extension for a trained model in order to synthesize a whole orchestra including instruments such as drums and percussion completely different from the ones synthesized so far is straight forward, yet interesting. A more challenging task is to apply organ to synthesize vocals or "voice oohs" for different vocal pitches: for this, it appears interesting to use a pianoroll-like representation where the position in the pianoroll stack indicates the vocal pitch. This way, synthesis of a human choir could be explored. This might be taken even further: so far, there are to the best knowledge of the author not many generative models conditioned on multi-dimensional input with spatial information. Using a

stack-like representation like the multi-track pianoroll, also computer vision tasks such as merging fore- and background of different images or combining portraits to group shots where the position of the portrait in the stack indicates the position of the person in the output image seem intuitively plausible. Overall, there is a wide range of further challenges.

List of Figures

List of Definitions

Index

References

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 265–283.

[2] Alain, G., and Bengio, Y. What regularized auto-encoders learn from the data-generating distribution. *The Journal of Machine Learning Research 15*, 1 (2014), 3563–3593.

[3] Arjovsky, M., Chintala, S., and Bottou, L. Wasserstein gan. *arXiv preprint arXiv:1701.07875* (2017).

[4] Arora, S., Ge, R., Liang, Y., Ma, T., and Zhang, Y. Generalization and equilibrium in generative adversarial nets (GANs). In *Proceedings of the 34th International Conference on Machine Learning-Volume 70* (2017), JMLR. org, pp. 224–232.

[5] Badrinarayanan, V., Kendall, A., and Cipolla, R. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence 39*, 12 (2017), 2481–2495.

[6] Bengio, Y. RMSProp and equilibrated adaptive learning rates for nonconvex optimization. *CoRR abs/1502.04390* (2015).

[7] Bengio, Y., Simard, P., Frasconi, P., et al. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks 5*, 2 (1994), 157–166.

[8] Bengio, Y., Yao, L., Alain, G., and Vincent, P. Generalized denoising auto-encoders as generative models. In *Advances in neural information processing systems* (2013), pp. 899–907.

[9] Burt, P., and Adelson, E. The laplacian pyramid as a compact image code. *IEEE Transactions on Communications 31*, 4 (1983), 532–540.

[10] Cauchy, A. Méthode générale pour la résolution des systemes d'équations simultanées. *Comp. Rend. Sci. Paris 25*, 1847 (1847), 536–538.

[11] Cybenko, G. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems 2*, 4 (1989), 303–314.

[12] Donahue, C., McAuley, J. J., and Puckette, M. S. Synthesizing audio with generative adversarial networks. *CoRR abs/1802.04208* (2018).

[13] Dong, H.-W., Hsiao, W.-Y., Yang, L.-C., and Yang, Y.-H. Musegan: Multi-track sequential generative adversarial networks for symbolic music generation and accompaniment. In *Thirty-Second AAAI Conference on Artificial Intelligence* (2018).

[14] Dong, H.-W., Hsiao, W.-Y., and Yang, Y.-H. Pypianoroll: Open source python package for handling multitrack pianoroll. *ISMIR Late-Breaking Demos Session* (2018).

[15] Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research 12*, Jul (2011), 2121–2159.

[16] Dumoulin, V., and Visin, F. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285* (2016).

[17] Engel, J., Agrawal, K. K., Chen, S., Gulrajani, I., Donahue, C., and Roberts, A. GANSynth: Adversarial neural audio synthesis. *arXiv preprint arXiv:1902.08710* (2019).

[18] Gauthier, J. Conditional generative adversarial nets for convolutional face generation. *Class Project for Stanford CS231N: Convolutional Neural Networks for Visual Recognition, Winter semester 2014*, 5 (2014), 2.

[19] Goodfellow, I., Bengio, Y., and Courville, A. *Deep learning*. MIT Press, 2016.

[20] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. In *Advances in Neural Information Processing Systems* (2014), pp. 2672–2680.

[21] Griffin, D., and Lim, J. Signal estimation from modified short-time fourier transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing 32*, 2 (1984), 236–243.

[22] Han, J., and Moraga, C. The influence of the sigmoid function parameters on the speed of backpropagation learning. In *International Workshop on Artificial Neural Networks* (1995), Springer, pp. 195–201.

[23] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 770–778.

[24] Hinton, G. E., and Salakhutdinov, R. R. Reducing the dimensionality of data with neural networks. *Science 313*, 5786 (2006), 504–507.

[25] Hornik, K., Stinchcombe, M., and White, H. Multilayer feedforward networks are universal approximators. *Neural networks 2*, 5 (1989), 359–366.

[26] Ioffe, S., and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[27] Isola, P., Zhu, J.-Y., Zhou, T., and Efros, A. A. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2017), pp. 1125–1134.

[28] Karras, T., Aila, T., Laine, S., and Lehtinen, J. Progressive growing of GANs for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196* (2017).

[29] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[30] Kingma, D. P., and Welling, M. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

[31] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.

[32] LeCun, Y. A., Bottou, L., Orr, G. B., and Müller, K.-R. Efficient backprop. In *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.

[33] Leshno, M., Lin, V. Y., Pinkus, A., and Schocken, S. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks 6*, 6 (1993), 861–867.

[34] Li, B., Liu, X., Dinesh, K., Duan, Z., and Sharma, G. Creating a multitrack classical music performance dataset for multimodal music analysis: Challenges, insights, and applications. *IEEE Transactions on Multimedia 21*, 2 (2018), 522–535.

[35] Li, C., and Wand, M. Combining markov random fields and convolutional neural networks for image synthesis. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 2479–2486.

[36] Li, C., and Wand, M. Precomputed real-time texture synthesis with markovian generative adversarial networks. In *European Conference on Computer Vision* (2016), Springer, pp. 702–716.

[37] Li, F.-F., Karpathy, A., and Johnson, J. Cs231n: Convolutional neural networks for visual recognition. *University Lecture* (2015).

[38] Long, J., Shelhamer, E., and Darrell, T. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015), pp. 3431–3440.

[39] Lu, X., Tsao, Y., Matsuda, S., and Hori, C. Speech enhancement based on deep denoising autoencoder. In *Interspeech* (2013), pp. 436–440.

[40] Lu, Z., Pu, H., Wang, F., Hu, Z., and Wang, L. The expressive power of neural networks: A view from the width. In *Advances in neural information processing systems* (2017), pp. 6231–6239.

[41] Maas, A. L., Hannun, A. Y., and Ng, A. Y. Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML* (2013), vol. 30, p. 3.

[42] Manzelli, R., Thakkar, V., Siahkamari, A., and Kulis, B. Conditioning deep generative raw audio models for structured automatic music. *arXiv preprint arXiv:1806.09905* (2018).

[43] McFee, B., Raffel, C., Liang, D., Ellis, D. P., McVicar, M., Battenberg, E., and Nieto, O. librosa: Audio and music signal analysis in python. In *Proceedings of the 14th Python in Science Conference* (2015), vol. 8.

[44] Mirza, M., and Osindero, S. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784* (2014).

[45] Müller, M. *Fundamentals of music processing: Audio, analysis, algorithms, applications*. Springer, 2015.

[46] Nair, V., and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)* (2010), pp. 807–814.

[47] Neumann, J. v. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen 100*, 1 (1928), 295–320.

[48] Odena, A., Dumoulin, V., and Olah, C. Deconvolution and checkerboard artifacts. *Distill 1*, 10 (2016), e3.

[49] Odena, A., Olah, C., and Shlens, J. Conditional image synthesis with auxiliary classifier GANs. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70* (2017), JMLR. org, pp. 2642–2651.

[50] Oliehoek, F. A., Savani, R., Gallego, J., van der Pol, E., and Gross, R. Beyond local nash equilibria for adversarial networks. In *Benelux Conference on Artificial Intelligence* (2018), Springer, pp. 73–89.

[51] Oliehoek, F. A., Savani, R., Gallego-Posada, J., Van der Pol, E., De Jong, E. D., and Gross, R. Gangs: Generative adversarial network games. *arXiv preprint arXiv:1712.00679* (2017).

[52] Oord, A. v. d., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499* (2016).

[53] Osborne, M. J., and Rubinstein, A. *A course in game theory*. MIT press, 1994.

[54] Pathak, D., Krahenbuhl, P., Donahue, J., Darrell, T., and Efros, A. A. Context encoders: Feature learning by inpainting. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016), pp. 2536–2544.

[55] Pedamonti, D. Comparison of non-linear activation functions for deep neural networks on mnist classification task. *arXiv preprint arXiv:1804.02763* (2018).

[56] Perraudin, N., Balazs, P., and Søndergaard, P. L. A fast Griffin-Lim algorithm. In *2013 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics* (2013), IEEE, pp. 1–4.

[57] Polyak, B. T. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics 4*, 5 (1964), 1–17.

[58] Rabin, J., Peyré, G., Delon, J., and Bernot, M. Wasserstein barycenter and its application to texture mixing. In *International Conference on Scale Space and Variational Methods in Computer Vision* (2011), Springer, pp. 435–446.

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

[60] Raffel, C. *Learning-based methods for comparing sequences, with applications to audio-to-midi alignment and matching*. PhD thesis, Columbia University, 2016.

[61] Ronneberger, O., Fischer, P., and Brox, T. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical Image Computing and Computer-assisted Intervention* (2015), Springer, pp. 234–241.

[62] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *nature 323*, 6088 (1986), 533–536.

[63] Sauer, T. Lecture notes in Einführung in die Signal- und Bildverarbeitung. In lecture, Summer term 2018. Faculty of Computer Science and Mathematics, University of Passau.

[64] Sauer, T. Lecture notes in Learning Theory. In lecture, October 2019. Faculty of Computer Science and Mathematics, University of Passau.

[65] Schroff, F., Kalenichenko, D., and Philbin, J. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015), pp. 815–823.

[66] Selesnick, I. W. Short-time fourier transform and its inverse. *Signal 10*, 1 (2009), 2.

[67] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research 15*, 1 (2014), 1929–1958.

[68] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015), pp. 1–9.

[69] Tieleman, T., and Hinton, G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning 4*, 2 (2012), 26–31.

[70] Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning* (2008), pp. 1096–1103.

[71] Walker, J. S., and Don, G. W. *Mathematics and music: Composition, perception, and performance*. Chapman and Hall/CRC, 2013.

[72] Wang, B., and Yang, Y.-H. PerformanceNet: Score-to-audio music generation with multi-band convolutional residual network. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2019), vol. 33, pp. 1174–1181.

[73] Wang, Z., Bovik, A. C., Sheikh, H. R., and Simoncelli, E. P. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing 13*, 4 (2004), 600–612.

[74] Wang, Z., Simoncelli, E. P., and Bovik, A. C. Multiscale structural similarity for image quality assessment. In *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003* (2003), vol. 2, Ieee, pp. 1398–1402.

[75] Weisstein, E. W. Sigmoid function.

[76] Wiesler, S., Richard, A., Schlüter, R., and Ney, H. Mean-normalized stochastic gradient for large-scale deep learning. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2014), IEEE, pp. 180–184.

[77] Wilson, D. R., and Martinez, T. R. The general inefficiency of batch training for gradient descent learning. *Neural networks 16*, 10 (2003), 1429–1451.