Universidad Nacional Autónoma de México

Faculty of Engineering - Facultad de Ingeniería

Beat Tracking In Musical Recordings With Neural Networks

By:

Magnus Henkel

T H E S I S

In Partial Fullfilment of Requirements for the Degree of

Computer Engineering

SUPERVISOR:

Dr. Ivan Vladimir Meza Ruiz

Mexico City, May, 2019

For my mother.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Magnus Henkel
May, 2019

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my supervisor, Dr. Ivan Vladimir Meza Ruiz, of the Research Institute of Applied Mathematics and Systems of the National Autonomous University of Mexico (IIMAS/UNAM). Without his inspiring guidance, warm encouragement, and endless patience throughout every step of the process, this paper would have never been accomplished.

I also want to express my gratitude to the rest of my thesis committee, including Dra. Jimena Olveres Montiel, M.I. Yahvé Abdul Ledezma Rubio, M.I. Gunnar Eyal Wolf Iszaevich, and Dr. Caleb Antonio Rascón Estebané, for their encouragement, insightful comments, and hard questions.

I would particularly like to thank Satwant Grewal, my long-time friend and soul mate, for his many hours of work, meticulously revising and correcting grammar mistakes and providing criticism.

Furthermore, I want to thank my partner, Gloria Nataly López Rodríguez, who was there for me whenever I reached a low point or when I was glued to my screen almost all nights and who endured the full range of emotions during this journey.

Finally, I am extremely thankful for the empowering riffs and pounding double-basses produced by the metal bands which have marked my personal beat for almost my entire life. They are too many to be mentioned here, but, without any of them knowing it, they have contributed greatly to this work.

ABSTRACT

This thesis reviews audio signal processing methods and machine learning techniques frequently employed in the context of beat tracking. Beat tracking refers to a music information retrieval task which aims at detecting the natural clapping rate in musical recordings, known as the beat. Clapping along to the beat of a musical piece seems to be a simple exercise, given the fact that most humans can achieve this quite intuitively, even without possessing any particular musical skills. Machines, however, cannot easily "pick up the groove" and are forced to go through a complex multi-step process, filtering out redundant information and extracting a set of beat-related features.

Beat tracking might be seen as a highly specialized and perhaps isolated activity in the larger scope of music research and automatic feature recognition, but, quite contrarily, it has to be understood as one of the fundamental and primary tasks for music information retrieval. In most song compositions, the beat functions as a carrier of the basic musical structure and helps to identify such fundamental components as tempo, chords, intention, mood, and genre.

Traditional beat tracking methods depended, to a great extent, on the careful preprocessing of audio signals, using specifically designed algorithms to find relations between events which often do not share common features, evade precise timing patterns, and can be subject to sudden and unpredictable changes. The features providing clues about the beat structure are hard to detect by manually controlling the parameters. Neural networks, however, offer an interesting alternative since they can internally determine the precise nature of these features without human interference, once they are properly designed and configured.

This thesis sets out to deepen our understanding of the ability of neural networks to efficiently extract audio features from training samples of spectrogram data and how these samples can be tailored specifically to the purpose of encoding temporal information and providing context. What is the optimal training sample size and, consequently, how much contextual information

can or should be conveyed? How is context captured at the audio signal level, and what effects does the inclusion of contextual data have on training times and performance? How do audio and network parameters affect each other, and what makes them play well together?

One approach taken in order to answer some of these questions was to seek a point of convergence between audio preprocessing methods, training sample generation, and network design, inspired by numerous publications in recent years. During the experimental part, the optimization process of two neural network architectures revealed important data-related characteristics of the learning process. In a posterior stage of the experiments, the (by this time) enhanced network design allowed the researcher to gain more insight into the question of how contextual information is encoded in the training samples and, secondly, whether a performance increase could be achieved by leveraging this context information.

It was shown that the way spectrograms were segmented into data samples and organized into so-called data partitions had an enormous impact on learning effectivity. Network performance was negatively affected when all data partitions contained samples stemming from the same audio excerpts. Another sequence of experiments showed that networks could indeed take advantage of contextual information embedded in the training samples but also that this was strongly dependent on the particular type of architecture. Trying to influence the network at a more granular level, e.g., by the use of a tolerance window in order to compensate for timing drifts, did not lead to observable improvements. In the majority of experiments, the performance obtained for each category (positive/negative) was found to be highly correlated to the actual number of samples which were produced (and therefore analyzed) per category under a specific configuration. The phenomenon of overfitting, which can be described as a poor ability to generalize patterns, constituted a major impediment to network performance. This situation, however, helped to identify some core factors adversary to the learning process.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Introduction to Beat Tracking

Music has fascinated mankind since time immemorial. It has been a driving factor in human culture since the early beginnings, playing a crucial role in many aspects of social life and functioning as a strong bonding factor in many kinds of human relationships.

Given its long history and immense diversity, it is not surprising that music has not only developed into a means of non-verbal communication (alongside its undeniable fun factor) but also serves as a conveyor of valuable information for scientific research. This thesis will explore a set of audio processing and machine learning techniques to gain insights into perhaps the most important pillar of music and, in particular, rhythm: the beat.

1.1.1 Musical Information Research

In an international and interdisciplinary effort, researchers in areas such as computer science, musicology, psycho-acoustics, psychology, and signal processing collaborate in the field of Music Information Retrieval (MIR). MIR defines a wide set of activities aimed at extending the understanding of musical foundations and their practicality in real-world applications in the context of machine learning and computational intelligence. The Music Technology Group of the Universitat Pompeu Fabra Barcelona characterizes MIR-related activities generally as ones having the aim of automatically generating descriptors "that capture the sonological or musical features that are embedded in the audio signals." These features range from musical facets such as rhythm, tonality, melody, and structure to semantic descriptors and concepts such as musical complexity, similarity, genre, mood, and social tags [1]. The annual Music Information Retrieval Evaluation eXchange (MIREX) conference, which has taken place since 2000 in different cities around the world, lists twenty-three different evaluation tasks on its wiki page[2], including genre, tag and mood classification, audio key detection, query by humming or singing, chord estimation, structural estimation, onset detection, and beat tracking. Participants are invited to compete in any of these areas by uploading their solutions to the MIREX submission system.

1.1.2 What is Beat Tracking?

Beat Tracking (BT) is widely described as the activity of tapping along in sync with the rhythm of music, an ability which most human beings possess, even without any prior musical training or skills. Seemingly an easy task for a person, beat detection with machine learning still faces a number of challenges. A reliable algorithm which is able to correctly identify rhythmic patterns in all kinds of musical genres, similar to a skilled musician, has yet to be developed. It should be noted that the aim of BT is not the analysis or extraction of particular beat structures or complete rhythms; instead, it tries to identify the pulse of the musical piece: an equally paced accent that can vary in frequency, depending on its tempo.

1.1.3 Why is Beat Tracking a Difficult Task?

The concept of beat, or beat time, is inherently ambiguous due to the fact that it is often subject to the individual interpretations of a listener who, often rather vaguely, perceives it as the underlying grid which carries the structure of the piece. Thus, for a human listener, finding the beat and nodding or clapping along is a highly intuitive and natural act, whereas for an algorithm, even deciding at which rate it should follow the pulse is relatively complex.

Current state-of-the-art BT systems are capable of achieving higher accuracies on audio material which exposes a clear and relatively simple structure, as is mostly the case with classic[3] types of Rock, Techno or Popular Music. However, some progress still has to be made in the domain of highly expressive musical styles, such as Classical, Jazz, and Experimental Music, where feature extraction must cope with varying tempi, increased rhythmic complexity, soft or missing note onsets on strong beat times, ornamented beats or syncopations, among others.

[1] `https://www.upf.edu/web/mtg/music-information-retrieval`

[2] `https://www.music-ir.org/mirex/wiki/2019:Main_Page`

[3] "Classic" in this context refers to a simpler and often older type or style versus one that is more experimental.

Expressive performances of musical pieces, as opposed to so-called *mechanical performances*, do not strictly follow the metrics of score time. In expressive performances, a variety of irregularities in terms of timing, tempo, and dynamics can be found. These irregularities can be introduced intentionally as stylistic devices, such as *ritardando* and *accelerando*, whereby the artist slows down or accelerates the tempo as part of the musical phrasing. However, they are more often simply attributable to the fact that human performance does not adhere to machine-like precision.

Since the majority of systems still rely on note onset detection as a primary step for determining beat and phase, feature extraction is required to not only be able to extract this information but also distinguish between useful and redundant data. Making this distinction is particularly hard in polyphonic music where masking effects can occur, resulting from overlapping energy peaks caused by the different voices of instrumentation and where, as a consequence, no significant change in the energy spectrum of the sound signal can be detected.

Another difficulty is the detection of the rhythmical meter, in other words, the best possible interpretation of tempo based on the various pulse levels and which translates to the rate of accents marking the beat.

As Dixon [14] mentions, there is no precise mathematical model of expressive timing or the complexity of musical structure from which timing is derived. In other words, expressive performance is harder to analyze by means of an algorithm, or in the form of rules, since interpretations of performing artists can vary significantly.

Lastly, one practical problem of automated BT should be mentioned, namely, the difficulty of generating or obtaining reliable and well-annotated data. Holzapfel et al. [31] partially attributes the relatively slow progress made by BT systems to the reutilization of the same datasets in many different research projects. Since producing new ground truth annotated data is a time-consuming and difficult task, the existing datasets remain in circulation, exacerbated by the fact that most of them seem to be biased towards easier musical styles. Musical excerpts which are harder to analyze are, therefore, often treated as outliers, and few efforts are made to focus on them specifically, leading to what the researchers call a "glass-ceiling effect," a situation where almost no further progress can be made.

1.1.4 Why is Beat Tracking an Interesting Task?

Many MIR-related tasks, e.g., automatic drum transcription and structural segmentation, require a dependable and efficient extraction of rhythmic features prior to initiating their own processing queue. BT, in these particular cases, serves as a frontend for these higher-level tasks. However, BT can provide useful information at the end-user or application level as well. Obvious examples of these are auto-adjusting instrument effects, machine music, audio repair and time-stretching algorithms, harmonic/percussive source separation, music synchronization, score extraction, audio content analysis, performance analysis, and perceptual modeling. The list of practical applications of MIR can only be growing under the current speed of mobile phone technology and the IOT.

Seen from a more general perspective, machines will be developed to get closer to what humans regard as "consciousness." Whether this artificial consciousness[4] will evolve into a simulated one or rather take the shape of a human-like self-awareness is still fervently debated, but the ability of machines to perceive music will certainly be one of great interest in the future.

1.1.5 Outline of the Approach followed by this Thesis

Part of the reason why beat tracking represents a big challenge at a technical level is the fact that an isolated onset cannot unambiguously be classified as a beat event. To correctly identify those onsets which form part of the beat structure, a concept often hard to pin down, it is essential to analyze the wider context in which the onset is embedded. Many of the methods described in this thesis deal in one way or another with context, be it explicitly, by establishing relations between the observed event and its neighbors through algorithms, or implicitly, by letting a neural network figure out the exact specifics on its own. Audio processing plays a significant role in transforming the raw stream of data, which may contain a high degree of redundancy for the concrete task, into a more refined representation. This is done by carefully preserving the context and highlighting the characteristics of events holding information about the beat structure. This thesis aims at shedding more light on how the rhythmical context of an event is captured and utilized.

Deep neural networks determine internally the exact nature of features on their own account. It is extremely difficult to trace the process of modeling which takes place between hidden layers.

[4]See `https://plato.stanford.edu/entries/mind-identity` for a short introduction to identity versus functionalism theories.

The approach followed in the practical part of this thesis is to provide the learning system with training samples which mask certain parts of the data and label these as context information. Another aspect which will be central to this work is the capacity of networks to infer useful information from data samples which are very similar to each other. Two different database setups will be used to investigate the implications of a less strict separation[5] of highly correlated training samples and its effect on the learning process.

This chapter has dealt with introducing the general concept of the task of BT and has given an overview of the difficulties it poses. In Chapter 2, audio processing techniques will be reviewed, most of them commonly employed in the more traditional, algorithm-based beat tracking approaches but which are, to a certain degree, still relevant to more recent systems using neural networks. Chapter 3 will explain two of the main neural network architectures in depth, the Convolutional Neural Network and the Recurrent Neural Network, both of which can be regarded as highly specialized variants of a general Artificial Neural Network. These architectures will be further explored in the practical part, the subject of following chapters. Chapter 4 is dedicated to the detailed description of the experimental setup, ranging from the specifics of the datasets to the implementation of the system which was designed and programmed for the purpose of this thesis. A comprehensive report on all conducted experiments and their results is compiled into Chapter 5. Finally, Chapter 6 revisits the main findings from the preceding chapter and tries to connect the dots from a broader perspective. This last chapter also underlines some limitations encountered in the experimental process and proposes steps to improve network performance at the technical, as well as the architectural, level.

[5]Separation in this context refers to a data partitioning scheme common in machine learning systems, where data is separated into different data domains, i.e., training, validation, and test subsets.

Chapter 2

The State Of The Art: Traditional Beat Tracking And Beyond

2.1 Definitions and Terminology

The majority of BT systems in the past and even today rely on the detection of significant features in the audio signal in order to evaluate their likelihood of being beat candidates. Commonly, these features are called onsets (or note-onsets) and the assumption is that in most cases beat times either coincide with, or are in some way related to, the occurrence of these events. Before going into more detail on what exactly is meant by *beat*, some of the terminology will be defined.[6]

Considering the envelope of a sound wave in the time domain, there are four main parts that can be identified: *attack*, *decay*, *sustain* and *release*. This is also called the ADSR curve. For the purpose of onset detection, the main focus is with the attack which marks the first interval of the ADSR curve, i.e., when the envelope rises from zero to its maximum. The attack of an audio event can be either soft, that is when the slope rises slowly, or hard, when a sharp rise can be observed. The latter is usually the case with drum sounds or other percussive elements. As it will be shown, a problem arises in the case of softer attacks since these are a lot harder to detect.

A *transient* refers to a short-time, high amplitude sound event which is accompanied by the occurrence of a broadband noise-like frequency spectrum. Generally speaking, transient regions are sections which distinguish themselves by a quick and rather chaotic increase in the signal's

[6]Most of the following definitions appear in publications of Bello et al. [3], Klapuri et al. [37] and Müller [44].

energy. A transient is usually followed by a longer and more stationary sound which carries tonal information.

The term *onset* refers to the precise instant where a transient begins, meaning it marks a specific point in time rather than a period. Feature extraction seeks to gather temporal information about onsets.

Klapuri et al. [37] define three different *pulse levels*: the *measure*, the *tactus* and the *tatum*. These three levels are strongly connected since their time scales are integer multiples or divisions of each other. For this reason, it is often possible to switch between them when tapping along, a fact which adds some complexity to automated BT due to its inherent ambiguity.

The tactus level is associated with the "clapping rate" and, therefore, it can be regarded as the most natural one. It is also referred to as the "beat level" (or quarter note level) and is, conceivably, the preferred metrical level a BT algorithm is meant to detect. "Tatum" means temporal atom and stands for the smallest, reasonable unit of meaningful accents within the beat grid. In a score representation, sixteenth notes are used for its notation. Measure refers to the largest musical scale and it can be useful for analyzing highly expressive music or in situations where musical accents are not closely linked to a regular underlying pulse, at least not in a very obvious way.

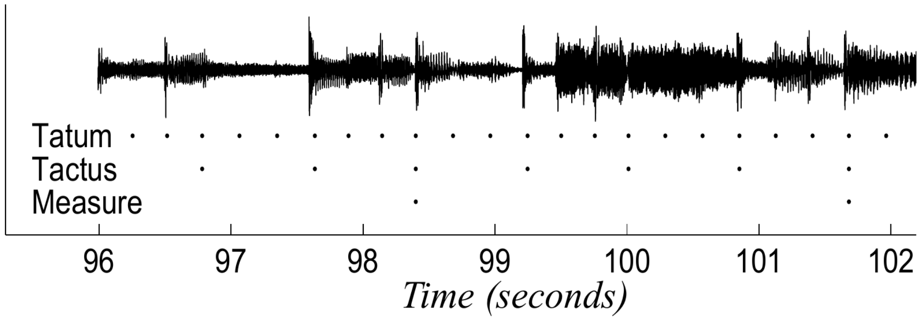


Fig. 2.1 Tatum, Tactus and Measure. Reprinted from Klapuri [36].

From a theoretical point of view, beat intervals are assumed to be regular and, therefore, beat positions are expected be be equally spaced in time. Ideally, at least from the viewpoint of BT, notes and rests are always expected to be found precisely on the grid, as defined by the pulse level, or at subdivisions of that same grid. If this is the case, it is known as score time, whereby a quarter note or a sixteenth rest always has exactly the same duration and relative distance.

However, when it comes to human performance, e.g., live concerts or even studio recordings, this is almost impossible to achieve.

BT is defined by its two parameters, *phase* and *period*, where the phase refers to the position of the beat relative to the grid, and the period is the time between two consecutive beat events in the analyzed sequence.

Another important concept in the context of BT is *tempo*. The tempo is defined as the rate of beats per time unit, generally measured in beats per minute (BPM). BPM proves to be a very useful scale since it facilitates comparison of results between different systems. It is closely related to the concept of period for being its reciprocal, i.e., the longer the period, the smaller the BPM. Detecting the tempo is a preliminary step of BT; once the rate of beats per time unit is clear, the grid of the underlying rhythmical structure becomes more tangible although this does not imply that the time scale is unequivocally correct.

The tempo, when detected at different levels of the time scale, is often represented by a *tempogram* which can give an idea about the coexisting interpretations of the pulse rate. The integer multiple and divisor relationship of time scales is reflected by a tempogram, for instance, in the case of having detected a tempo of 120 BPM, parallel to this pulse rate there might also appear the tempi of 240 BPM, 60 BPM or 180 BPM. When the rate is halved or doubled, sometimes the term *tempo octave* is used, an analogy to the *pitch octave* which is very common in a musical context. Furthermore, references to *tempo harmonics* can be found in the case of integer multiples, and *tempo subharmonics* for integer fractions [44].

In many cases, especially when analyzing polyphonic music, the detection of onsets may be exacerbated by *masking effects* caused by the concurrence of sound events stemming from different voices of instrumentation. In this situation, finding onsets by exclusively looking at energy peaks in the signal might not be enough since all of the voices contribute to these peaks, and conclusions about the relevancy of this particular piece of information in the context of BT can be distorted. This is where the notion of *salience* comes into play. Salience generalizes the targeting of prominent events as beat candidates in that it includes much more information than just the amplitude into the search criteria, e.g., pitch, note density, duration, timbre, among others. This can be understood as the "weighting" of individual accents in the attempt to weed out redundant information.

2.2 Note-Onset Detection

As stated earlier, onset detection tries to preselect beat candidates whereby each of these single events can later be put into a context of recurring patterns which sustain the rhythmic structure. In more than twenty years of research, many strategies have been developed to extract the relevant features while discarding misleading information and ambiguities. In the following sections, it will be demonstrated how a classical pipeline of feature extraction can look like. This pipeline is, for the main part, inspired by the descriptions of Müller [44] and is divided into three main stages: *pre-processing*, *reduction* and *peak-picking*. The goal in this process is to strip down the signal to the point where, ideally, only a relatively regular pattern of peaks remains, referred to as the *novelty curve*. The simplified structure of the resulting novelty curve makes it easier to select certain points in time where beat times are likely to occur. It should be noted that the stages of this pipeline are not necessarily carried out in a linear fashion and that certain steps may well be executed in a different manner or order.

2.2.1 Audio Signal Preprocessing

The division of the signal into different frequency bands or *spectrum strips* is a common technique to circumvent limitations imposed by employing only a single reduction method [3]. This allows for a more fine-grained feature extraction procedure, adds robustness and increases the success rate of some systems. Furthermore, instruments which carry significant beat information can be analyzed within their range of frequency, e.g., the bass drum has its tonal part (the one which follows the attack) in the lower spectrum. Thus, by applying a low-pass filter, this particular information can be targeted more directly while reducing redundant data. The same principle is applicable for sound events in higher frequency ranges. However, it should be kept in mind that the perception of loudness in the human cochlea is not linear but logarithmic. By dividing the signal into several bands or strips, each of them can be worked on separately and different parameters can be applied, including differing time-scales, intensity, etc.

Methods using band-wise processing algorithms often try to exploit psycho-acoustic knowledge, as described by Klapuri [36]. In his earlier experiments, he was able to improve upon the existing methods of onset detection by separating the signal into 21 non-overlapping bands, determining onset intensities for each of them and then combining them the results.

Duxbury et al. [18] has been conducting research on hybrid systems where the signal is sliced into five frequency bands for further targeted processing. The higher frequency spectrum is scanned for energy changes whereas spectral changes are considered in the lower spectrum only. One reason for this approach is that transients in the higher bands show short and well separated energy bursts; in contrast, at the lower end of the spectrum, their decay phase is usually slower and longer. Another problem in the low-frequency band is that the time resolution is considerably lower too due to the fact that note events occur at a faster rate relative to their frequency.

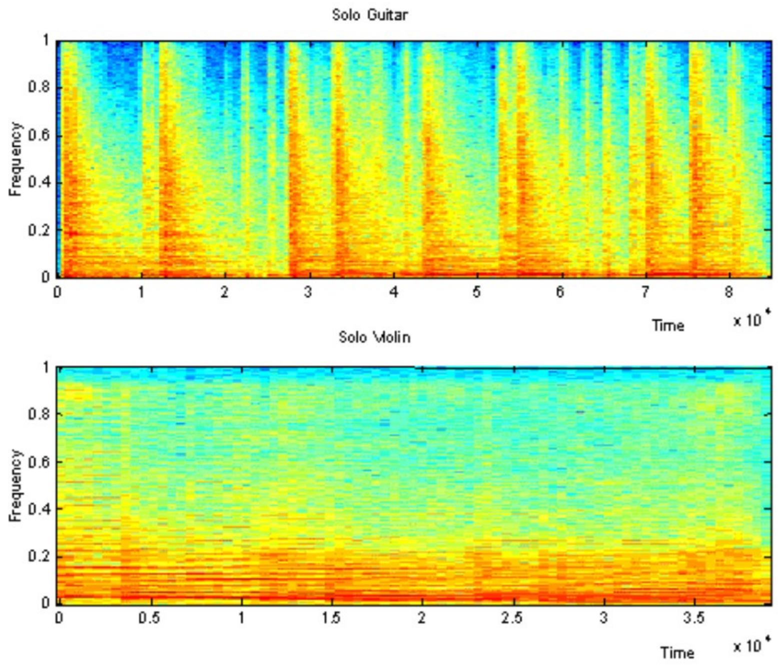


Fig. 2.2 Comparing spectrograms of a guitar and a violin. Reprinted from Duxbury et al. [18].

Figure 2.2 shows why onset detection to a certain extent depends on the presence of well defined transients. Comparing the spectrograms of an electric guitar and a violin shows that for the more rhythmic guitar, transients are perfectly distinguishable because of its fast attack times and rapidly decreasing decay phase. On the other hand, the violin features much smoother transitions, and note changes are more noticeable because of pitch information contained in the signal. Moreover, the decay phase is practically non-existent, effectively maintaining the energy at a constant level. It is easy to see that the soft onsets in the latter case are almost impossible to detect.

2.2.2 Novelty Curves

In order to reduce the signal to a representation with well separated peaks indicating the onset candidates, a feature detection function can be applied. This function is also known as a *novelty function*. After processing the signal in this way, the resulting output will be a *novelty curve*. The word "novelty" hints at the fact that this curve solely contains indicators of change of some musical or physical property in the signal. Two different approaches to obtain a novelty curve will be discussed in more mathematical detail, specifically *energy-based* and *spectral-based*, as they can be seen as some sort of base cases. These sections are mainly based on information found in the excellent book "Fundamentals of Music Processing" by Meinard Müller.

Energy-Based Feature Extraction

The simplest form of deriving the novelty function is energy-based feature extraction. It is based on the observation that percussive voices in the analyzed signal are accompanied by sudden energy bursts. Hence, these methods make use of a local energy detection function which can be described as follows [44]:

$$E_w^x(n) = \sum_{m=-M}^{M} |x(n+m)w(m)|^2 = \sum_{m\in\mathbb{Z}} |x(m)w(m-n)|^2 \quad , \tag{2.1}$$

where x is the discrete time signal, w is a bell-shaped window function which is centered at time zero and shifted over the signal x by n samples, and $m \in [-M : M]$ limits the set of non-zero samples of w.

However, keeping the energy levels for whole transients is not necessary since onset extraction only looks for the starting point of these transients. This redundant data can largely be removed by taking the discrete derivative of the signal. This is done by simply subtracting the energy level of sample $n-1$ from the value at sample n, for all values of n. Afterwards, the signal is left with the slopes only.

As a next step, an operation known as *half-wave rectification* can be applied. Thus, only the positive part which relates to onsets is kept, whereas the negative part is set to zero, effectively eliminating the negative slopes of the signal.

Mathematically, this can be expressed as follows:

$$|r|_{\geq 0} = \frac{r+|r|}{2} = \begin{cases} r & \text{if } r \geq 0 \\ 0 & \text{if } r < 0 \end{cases} \qquad \text{with r} \in \mathbb{R} \tag{2.2}$$

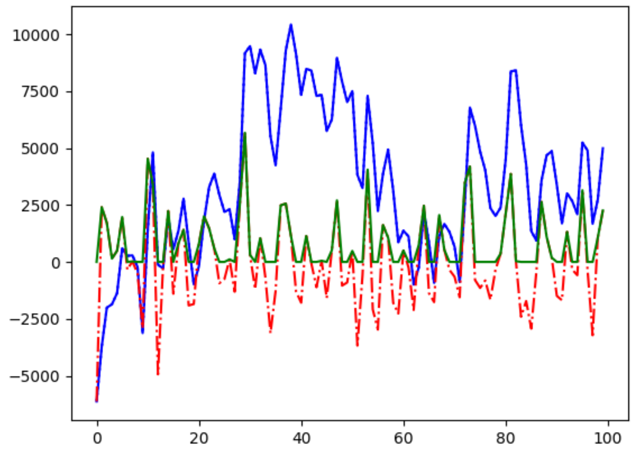


Fig. 2.3 Original signal (blue), discrete derivative (red), half-wave rectification (green)

The final expression of the energy-based novelty function will be:

$$\Delta_{Energy}(n) = |E(n+1) - E(n)|_{\geq 0} \tag{2.3}$$

Most systems adapt the frequency scale logarithmically at this point in order to adjust to the logarithmic behavior of the human auditory apparatus. This is an optional step, however, and is not guaranteed to produce better results. Grosche and Müller [25], for example, propose a novelty function in which they prefer to hold on to the linear scale, arguing that with this approach, higher frequencies get promoted. These higher frequencies, in turn, emphasize the noise-like spectral components which are inherent to transients and make it easier to detect onsets.

Note that if a logarithm is applied, the local energy function will not be a simple difference of energy values anymore; instead, it will correspond to the logarithm of energy ratios [44]:

$$\Delta^{Log}_{Energy}(n) = \left|\log(E^x_w(n+1)) - \log(E^x_w(n))\right|_{\geq 0} = \left| \log\left(\frac{E^x_w(n+1)}{E^x_w(n)}\right) \right|_{\geq 0} \tag{2.4}$$

Energy-based feature extraction works quite well for purely percussive music with hard onsets on beat times but has proven to be more error-prone for polyphonic and highly expressive music, in which case spectral-based feature extraction yields far better results.

Spectrum-Based Feature Extraction

In spectral-based feature extraction, the signal has first to be transformed from a simple waveform representation into the time-frequency domain by means of the *short-time Fourier transform* (STFT). The output of this operation is a spectrogram $X = (X(k,t))_{k,t}$ with $k \in [1 : K] = \{1, 2, .., K\}$ and $t \in [1 : T]$. K denotes the Fourier coefficients and T is the number of frames; $X(k,t)$ stands for the k^{th} Fourier coefficient for time frame t.

The STFT is a variant of the Fourier transform whereby a sliding window function is used to ensure temporal localization of frequencies. The window function is usually chosen to be a discretized Hann function, resulting in the so-called *Hann window* (sometimes wrongly referred to as "Hanning window"). In theory, any other window function could be used but the Hann window proves to be a convenient choice since it mitigates the effect of *frequency leakage*. To understand how the Hann window influences frequency leaking, a review of basic definitions and some of their implications is needed:

The sampling period Δt is obtained by the inverse of the sampling rate:

$$\Delta t = \frac{1}{f_s} \tag{2.5}$$

On the other hand, the frequency of the k^{th} bin is known to be related to the frequency of the input signal by:

$$f_k = \frac{k}{N\Delta t} = \frac{kf_s}{N} \quad , \tag{2.6}$$

where k is the index of a frequency bin, N the number of samples (the length of the window) and f_k the frequency of the k^{th} bin. The reason why frequencies are bundled into *bins* (or buckets) is that a discrete system is not capable of processing or visualizing continuous streams of data; thus, these bins express a given frequency resolution.

The above formula can be used to compute the bin size by finding two successive values of f_k and subtracting one from the other. As an example, given a sample rate of $f_s = 44.1kHz$ and a window length $N = 2^{12}$, the approximate frequency resolution for k = 1 would be $10.7Hz$, whereas for $k = 0$, the result is 0. Thus, the frequency band in this example has a range of

$10.77Hz$. Given a smaller window length of $N = 2^{10}$ instead of $N = 2^{12}$, the result would be approximately $86.1Hz$, a much coarser resolution. One conclusion which can easily be drawn from this example is that frequency resolution depends on the window size, assuming that the sample rate is fixed. A smaller window has the advantage of a better time resolution but includes less samples to analyze; consequently, frequency resolution is reduced. On the other hand, a larger window narrows the frequency range which can be addressed by any frequency bin of index k, but diminishes temporal localization. An extreme case would be a window spanning the whole signal length where $k = f_s$; in other words, all possible frequencies, up to the limit defined by the Nyquist[7] theorem, would be found but without any time localization at all since all frequency information would be averaged over the whole time interval of the signal.

Solving equation 2.6 for any sample index k gives:

$$k = \frac{Nf_k}{f_s} \tag{2.7}$$

The fundamental frequency range of a bass drum lies at about $60-80Hz$ (for the characteristic "thump" sound). In this case, the k^{th} index number of the center frequency, i.e., $70Hz$, would be $k = Nf_k\,/\,f_s = 2^{12}\cdot 70\,/\,44100 \approx 6.5$. However, an index number must be an integer referencing a single frequency value, not a range. This is known as frequency leakage (or spectral leakage); in the case of the given example, the energy leaks between bin 6 and 7. Some frequencies do not show this behavior; for instance, $f_k = 689.0625Hz$ would give the exact bin index 64. Hence, frequency leakage varies depending on the given frequency.

The Hann function is a discretized function which returns non-zero values inside a given range, and zero values outside. Mathematically, it can be expressed thus:

$$w(n) = \begin{cases} 0.5 - 0.5cos(2\pi n/M), & 0 \leq n \leq M, \\ 0, & \text{otherwise} \end{cases} \quad . \tag{2.8}$$

As figure 2.4 shows, this window transitions smoothly from zero to a maximum amplitude and, likewise, from this maximum to zero again. Hence, any function multiplied with the Hann window at time t will be "faded" towards zero at both extremes.

[7]The Nyquist Theorem states that in order to correctly reconstruct a waveform from sampled data points, the sample rate must be equal or higher than the highest frequency in the signal: $f_{sr} \geq 2\cdot f_{sig}$

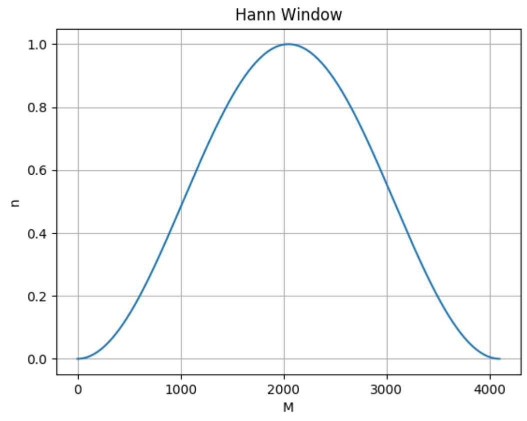


Fig. 2.4 Hann window.

This is beneficial in two ways. In the case that a signal (captured by the window) is not exactly periodic, unwanted audible clicks can occur as a result of this hard transition. More importantly, however, a broadband noise is produced by a sudden signal disruption spreading over a wide range in the frequency spectrum. By fading out the signal at window transitions, the audible clicks disappear, and the leakage bandwidth is constricted to the proximity of the bin frequency with the result that it is dispersed more evenly over all bins [41].

Another technique commonly applied is *logarithmic compression*; instead of the scale being modified, the actual values are scaled logarithmically [44, pp. 125]. A spectrogram, since it usually covers the whole set of audible frequencies (from $40Hz$ up to almost $20kHz$), has a large dynamic range. Thus, values in the lower ranges which contain valuable information for pulse detection might be overshadowed by values in higher regions. Compression helps to make these smaller values gain prominence by scaling them up and, at the same time, scaling bigger values down so that relative distances between peaks decrease.

Let the spectrogram $X \in \mathbb{R}_{>0}$, $\gamma \in \mathbb{R}_{>0}$ and $\Gamma_{>0} \to R_{>0}$. Logarithmic compression can then be denoted as:

$$\Gamma_{\gamma}(|X|) = log(1 + \gamma \cdot X) \tag{2.9}$$

The value of the constant γ defines the degree of compression. Higher values will produce a stronger compression. The ideal γ depends very much on the characteristics of the input signal and the application in mind. As an example, BT seeks only significant onsets for beat estimation, but a task like drum transcription will need to extract every single percussive event

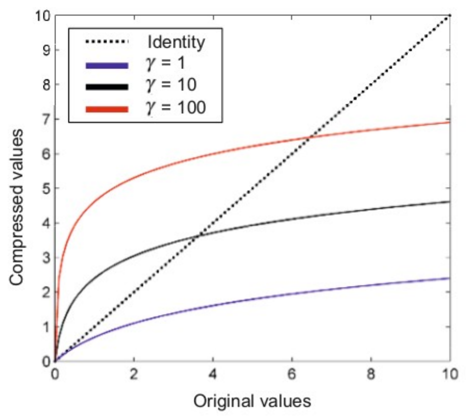


Fig. 2.5 Logarithmic compression with three different γ values. Reprinted from Müller [44].

whether it is part of the beat or not. The drawback of any strong compression is that besides low-energy peaks being amplified, unwanted background noise is increased as well.

The compressed spectrogram (Y) can then be processed in a similar fashion as the energy-based novelty function by applying the discrete-time derivative and keeping only the positive part by means of half-wave rectification. After these steps are applied, a spectrum-based novelty function, also called *spectral flux* (see Figure 2.6), is obtained:

$$\Delta_{Spectal}(n) = \sum_{k=0}^{K} \left| Y(n+1,k) = Y(n,k) \right|_{\geq 0} \tag{2.10}$$

2.2.3 Other Onset Detection Methods

To conclude the topic of onset detection techniques, this section will point out some interesting publications about differing or extended methods on the subject.

Bello et al. [4] combine energy-based onset detection with phase information by building upon the observation that during the steady-state of a signal, the phase information stays constant whereas when a transient is reached, the phase deviates notably.

Zhou et al. [55] work with resonator time frequency images (RTFI) to extract more natural energy change and pitch change cues from the energy spectrum which are then correspondingly fed into an energy-based and a pitch-based detection algorithm.

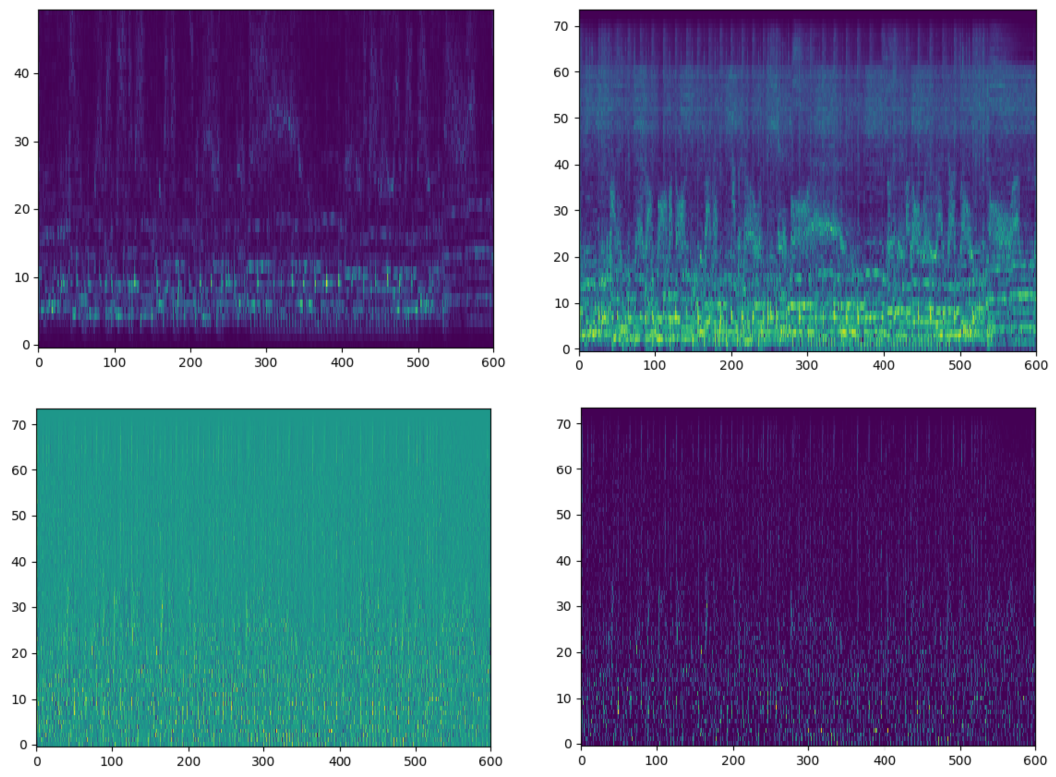


Fig. 2.6 Spectral flux. The spectrogram images show the preprocessing steps applied to a spectrogram in order to obtain the spectral flux. The audio excerpt to produce these images stems from the GTZAN dataset (described in Chapter 4). Top left: original spectrogram; top right: logarithmic scaling; bottom left: discrete derivative; bottom right: half-wave rectification.

Abdallah and Plumbley [1] use independent component analysis to fit a short-term non-Gaussian model to frames of audio data. The hypothesis is that onsets can be detected by uncovering a factor of "surprisingness" in the original audio signal. In a second step, a hidden Markov model is employed to cluster these "surprise" events.

2.2.4 Postprocessing

Postprocessing, like preprocessing, is an optional step taken after obtaining the novelty curve and depends on the reduction method employed [3]. Its primary purpose is the *normalization* and *smoothing* of the onset detection function. Normalization helps to avoid the detection of false positives by reducing the dynamic range of the signal's energy and, thus, making it easier to apply a threshold function during the peak-picking phase (described in Section 2.2.5). Smoothing tries to eliminate noise. A very common way of doing this is computing the local average of the $\Delta_{Spectral}$ (see Equation 2.10) and subtracting the result from the original function. The local average function is denoted as follows [44]:

$$\mu(n) = \frac{1}{2M+1} \sum_{m=-M}^{M} \Delta_{Spectral}(n+m) \tag{2.11}$$

The average is subtracted and, as before, half-wave rectification is employed to obtain the final novelty curve $\overline{\Delta}_{Spectral}$:

$$\overline{\Delta}_{Spectral}(n) = \left|\Delta_{Spectral}(n) - \mu(n)\right|_{\geq 0} \tag{2.12}$$

2.2.5 Peak-Picking

The final step in the process of obtaining the onsets is called peak-picking [3]. This is most commonly done by applying a threshold function to identify local maxima (peaks). Duxbury et al. [17] point out that the technique of a simple threshold is often problematic. The main reason for this is that detection functions tend to be noisy, and high-energy lower band noise can impede the clean separation and identification of peaks when the threshold is not optimally set. Series of repeating onsets of different types over short segments in the signal is another issue. Furthermore, the thresholding mechanism can be impaired by fluctuations in magnitude over longer sections.

One possible, but not always practical (in terms of usability), solution to counter these problems is to set the threshold manually for each signal. A better alternative is given by automatically adjusting the threshold value. This *adaptive threshold*, as described by Bello et al. [3], proves to be a robust mechanism when dealing with highly dynamic signals, mitigating potential masking effects by lowering the impact of very large peaks on lower ones. The adaptive threshold function is calculated using a moving-median filter which is then subtracted from the normalized detection function. Once the onset function is smoothed by the adaptive threshold, all local maxima above zero are considered as onsets.

2.3 Tempo Estimation Algorithms

Once a novelty curve is achieved by a successful reduction of the input signal, an attempt to derive the tempo can be made. Tempo estimation is based on two main assumptions: firstly, the elapsed time between beats is approximately constant during larger sections of the musical piece, and, secondly, these beat positions coincide with onset positions. The principal difficulty of this task lies in the vast amount of musical material which deviates from the ideal scenario of regularly spaced beat times with easily discernible onsets. Even if a steady tempo can be detected, there is still room for ambiguity because of the already discussed metrical levels, which may be sub-optimally detected by a BT system.

2.3.1 Tempograms

A valuable tool for visualizing coexisting tempo hypotheses found in the analyzed signal is the *tempogram*, which can be seen as an analogue to the spectrogram since it tracks the parameter of interest, in this case BPM, over time. Similarly to the spectrogram, higher values are highlighted by the intensity of color, e.g., a dominant local tempo could manifest itself as a lighter horizontal line contrasted by a darker background. Information about multiple different pulse levels are directly reflected by a tempogram. A *predominant local pulse* at 120 BPM, for instance, could be accompanied by its integer multiples (240, 360, etc) or by its fractions (60, 30, etc). Figure 2.7 shows a novelty curve and its corresponding Fourier tempogram, explained in detail in Section 2.3.1.

As an analogue, when talking about "pitch," fundamental frequencies correlate with their overtones also by integer multiples or fractions. These relations are called *harmonics* (see Section 2.1). Since the temporal representation in a tempogram is quite similar in nature, it is common to refer to these multiples and fractions as *tempo harmonics* or *tempo subharmonics*. Following this naming pattern, a *tempo octave* then describes the exact half or double of a given tempo.

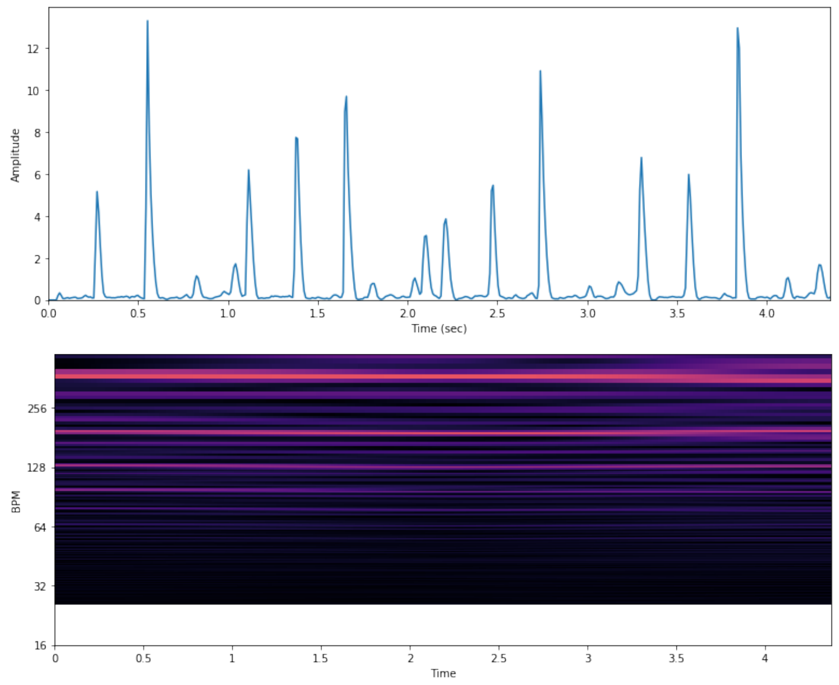


Fig. 2.7 Novelty curve of a drum beat and its Fourier tempogram. Both graphs were created by following the instructions in the Jupyter tutorials provided at `https://musicinformationretrieval.com/` and `https://github.com/stevetjoa/musicinformationretrieval.com`.

In the following section, two widely used methods for obtaining a spectrogram are presented: the *Fourier tempogram* and the *autocorrelation tempogram*.

Fourier Tempogram

As the name already suggests, in order to generate a Fourier tempogram, the Fourier transform (more specifically, the Short Time Fourier Transform (STFT)), is employed. The basic principle of the Fourier transform is to compare sine waves with the signal of interest to extract frequency information. Since the goal of tempo analysis is to extract periodic patterns in the analyzed signal, the application of the Fourier Transform, which utilizes the periodic nature of sine waves, can be exploited to infer the tempo. This is in contrast to the more common use case of attributing the event of a matching frequency in the signal to the pitch. The strongest recurrent patterns between the peaks in the novelty curve point to various tempo hypotheses which, in turn, can give indications of possible beat levels.

A discretized sliding Hann window is used to produce the complex Fourier coefficients for any frequency $\omega \in \mathbb{R}_{\geq 0}$ at frame index $n \in \mathbb{Z}$:

$$F^C(n,\omega) = \sum_{m \in \mathbb{Z}} \Delta(m) w(m - nH) e^{-2\pi i \omega (m-nH)} \quad , \tag{2.13}$$

where $H \in \mathbb{N}$ is the hop size given in samples. A tempogram visualizes locally dominant tempi for a defined time interval, usually measured in seconds. On the other hand, frequency is the inverse of time period T, written as $\omega = 1/T$. To derive the tempo, measured in BPM, from the frequency, the latter has to be multiplied by the seconds in one minute; thus, tempo is defined as $\tau = 60 \cdot \omega$. Knowing these simple properties, it is now possible to translate any period between peaks into a tempo measured in BPM. For example, if the average time period between a couple of peaks is 0.8 seconds, the tempo can be computed by inserting 0.8 into the above formula. This results in a tempo of 75 BPM since the frequency will be $1/T = 1/0.8 = 1.25$ Hz and thus $60 \cdot 1.25 = 75$ BPM.

The tempogram τ^F is obtained by converting the frequency into tempo value, for every instant n:

$$\tau^F(n,T) = |F^C(n,\tau/60| \tag{2.14}$$

Figure 2.8 shows snapshots of three different sine waves which are matched up against the original novelty curve in certain localities. The corresponding Fourier tempogram is shown on the left. The graph at the top right shows the best match of the three sine waves – the peaks in the positive part of the generated sinusoid line up with the peaks of the novelty curve, and the negative parts coincide with the gaps, effectively zeroing out due to half-wave rectification. The obtained Fourier coefficients show a high correlation with the windowed sinusoid in this case. This is reflected in the resulting tempogram by a clear and distinct horizontal line (marked in red) indicating the most adequate pulse level.

The other two cases, (2) and (3), align to some degree as well but much less so. Subfigure (2) shows a sine wave of half the frequency, where approximately every other negative peak of the window function lines up with a positive peak of the signal, canceling out each other. Consequently, these correlation coefficients are at their lowest and result in weaker to almost invisible lines in the tempogram representation. The double frequency sinusoid (3) performs

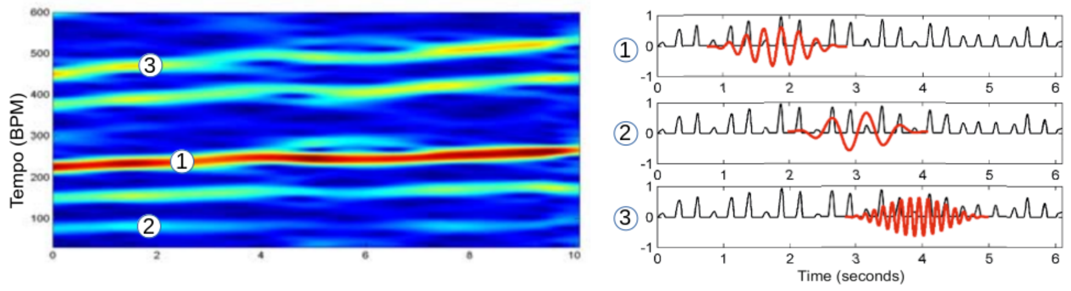


Fig. 2.8 Fourier tempogram. (1) Optimal pulse level; (2) half-tempo; (3) double-tempo. Reprinted and adapted from Müller [44].

significantly better, even though more false positives (peaks occurring inside gaps) are encountered when compared to (1). The Fourier tempogram tends to emphasize tempo harmonics and suppress subharmonics due to the fact that canceling out peaks carries more of a penalty than amplifying zero.

Autocorrelation Tempogram

In the context of tempo estimation, autocorrelation is a technique which detects periodic patterns by comparing a signal with a time-shifted copy of itself. In practice, a small segment of the signal is taken and shifted along the time axis. When similar peaks align, the correlation between them will be high, as already seen with the Fourier tempogram. The distance traveled by the time-shifted segment at this instant is measured and constitutes the so-called *lag* parameter.

For a real-valued, discrete-time signal, autocorrelation $R_{xx} : \mathbb{Z} \to \mathbb{R}$ can be defined as follows [44]:

$$R_{xx}(l) = \sum_{m\in\mathbb{Z}} x(m)x(m-l) \quad , \tag{2.15}$$

where $l \in \mathbb{Z}$ is the lag. Again, a window function $\Delta(w,n) : \mathbb{Z} \to \mathbb{R}$, which in this case contains a copy of the segment, is used to compare the signal against. It can be written as:

$$\Delta_{w,n}(m) = \Delta(m)w(m-n) \quad , \tag{2.16}$$

with $n,m \in \mathbb{Z}$.

As a first step, a short-time autocorrelation $A : \mathbb{Z} \times \mathbb{Z} \to \mathbb{R}$ can be obtained:

$$A(n,l) = \sum_{m\in\mathbb{Z}} \Delta(m)w(m-n)\Delta(m-l)w(m-n-l) \tag{2.17}$$

To be meaningful in terms of the actual goal of tempo estimation, the time-lag representation needs to be converted into a time-tempo representation. This can be done by means of the following formula:

$$\tau = \frac{60}{r \cdot l} BPM \tag{2.18}$$

Here, the frame rate of the novelty curve is assumed to be of length r, hence a shift of $l \cdot r$ seconds corresponds to a rate of $1/(l \cdot r)$ Hz. See [44].

Next, the autocorrelation tempogram T^A is derived from:

$$T^A(n,\tau) = A(n,l) \quad , \tag{2.19}$$

where τ is obtained by equation 2.18 and $l \in [1:L]$.

Figure 2.9 (right) shows a six-second-long segment of the original novelty function superimposed by a sliding window function (marked in red) which is shifted along the time axis by three different lag parameters. The numbered points in the tempogram[8] on the left show that (2) has the highest correlation and therefore represents the preferred pulse level, whereas (1) results in a weaker horizontal line indicating a subharmonic. Graph (3), a slightly phase-delayed copy of the original segment, does not match any of the peaks and, accordingly, shows a very low degree of correlation.

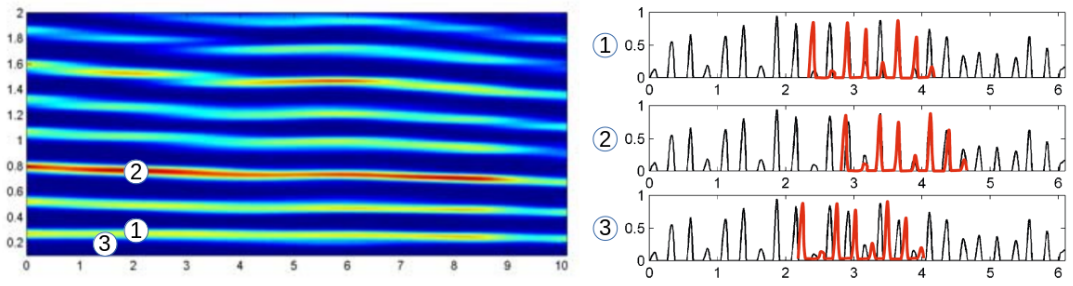


Fig. 2.9 Left: autocorrelation tempogram; right: novelty curve with sliding window; subgraphs on the right: subharmonic (1); optimal pulse level (2); no correlation (3). Reprinted and adapted from Müller [44].

As stated earlier, the Fourier tempogram promotes tempo harmonics while, at the same time, tempo subharmonics remain mostly undetected. Interestingly, the autocorrelation tempogram behaves contrarily in this respect: tempo subharmonics are indeed encountered easily whereas

[8]Note that the lag scale is not yet converted into the more useful BPM scale. The conversion is usually done by resampling or interpolation.

harmonics are suppressed. Intuitively, it can be imagined that the farther the window is shifted along the time axis, the lower the resulting BPM ratio will be (after lag-to-BPM conversion).

So far, it has been demonstrated that pulse information can be gathered and visualized by means of tempograms. However, one important issue still remains, namely, the simultaneous presence of strong indicators for different pulse levels. Since the Fourier tempogram highlights tempo harmonics whereas the autocorrelation tempogram emphasizes tempo subharmonics, it is intuitive to propose combining these tempograms to reduce the ambiguity inherent in the tempo analysis. A *cyclic tempogram* provides yet another method to address this problem. It is described in detail by Müller [44, p. 325].

2.4 Beat Tracking

After reviewing feature extraction and tempo induction techniques, the process of the actual BT will be explored by building on top of the results obtained during these earlier steps. Tempo induction was used to determine the beat period, i.e., the distances between beat candidates, for different tempo hypotheses. Beat tracking, however, requires finding the exact positions relative to the rhythmical grid. In Section 2.1, this was referred to as the beat phase. As stated by Hainsworth and Macleod [26], "to represent the tempo curve, a frequency and phase is required such that the phase is zero at beat locations."

Many different *beat induction* methods have been developed: Dixon [15] uses a system called "Beatroot" which analyzes onsets by means of a multi-agent architecture. The algorithm works by clustering inter-onset intervals (IOI) which occur frequently and can be projected onto a regular grid. Each of these clusters is built by an agent responsible for a specific IOI. The agents compete against each other in making predictions from the current state to following beat positions and aggregating correct estimations. Once the processing is done, the strongest cluster yields the solution for the beat tracking problem.

Peeters and Papadopoulos [46] present a probabilistic framework for beat and downbeat detection. Beat times are considered as hidden states, decoded over beat numbers by a proposed "reverse" Viterbi algorithm. The observations, i.e., chroma vectors and spectral balance, are compared to beat templates and their probabilities are evaluated by *linear discriminant analysis*.

Yet another probabilistic approach is the "reliability-informed" beat tracker by Degara et al. [13]. Beat and non-beat information is extracted by explicitly modeling non-beat states with a Hidden

Markov model. Predictions are then subjected to an automatic quality assessment whereby the measures of accuracy and reliability are estimated by the k-nearest neighbor regression algorithm.

Hainsworth and Macleod [26] proposed a beat tracking method based on particle filters (a sequential Monte Carlo estimation method) where, for each particle, an importance function calculates a probability for the new state to be a beat candidate. This method was based on a similar system developed earlier by Cemgil and Kappen [8], and which was described by the authors as a switching state space model for joint rhythm quantization and tempo tracking.

While these examples only represent a tiny selection of different approaches aimed at tackling automated BT, it is worth looking at one more exemplary technique which aims at the refinement of a previously obtained novelty function: the *predominant local periodicity curve*.

2.4.1 Predominant Local Periodicity Curve (PLP Curve)

In his book, Müller [44] describes the PLP curve as a "local periodicity enhancement of the original novelty function." One interesting aspect of local periodicity is that this procedure does not assume constant tempo throughout the whole piece since the predominant pulse level is analyzed over smaller stretches in the novelty curve. This is indicated by the word "local" in the name of this method. After taking the steps necessary to produce a tempogram for the audio excerpt, the strongest tempo indicators τ for each time position n are extracted by applying a max function. These tempo parameters τ_n are then used to calculate the phase ϕ_n which belongs to the windowed sinusoid at τ_n.

Based on the information of the likeliest pulse level at a certain time and its exact position with respect to the windowed frame it belongs to, an optimal windowed sinusoid K_n can be calculated for time step n. Essentially, K_n tries to fit the original local novelty function as closely as possible, aligning its peaks with the latter. Phase information is crucially important for correct alignment, while picking the strongest local tempo indicator ensures that the majority of peaks are matched. Evidently, the quality of the novelty function, i.e., the property of clearly distinguished and regularly spaced peaks, decides if such an optimal sinusoid can be found. In the case that no tempo information is present or other irregular temporal patterns occur at the locality n, the generation of K_n could potentially fail and tempo misinterpretations are likely to ensue.

The *PLP function* takes these unavoidable information gaps (caused by partly corrupted novelty curves) into account by applying an *overlap-add technique* to subsequent windowed K_n. These sinusoids are accumulated at all time positions n and later undergo a half-wave rectification and normalization to derive the final PLP curve. Rectification and normalization are employed for similar reasons as already seen in earlier sections: preserving the positive part only, countering the effect of constructive interferences when peaks are perfectly aligned and which can lead to excessive outliers, and forcing local maxima to take values close to one. Figure 2.10 shows the improvement achieved by a PLP function over the original novelty curve. In addition to bridging the gaps in the original curve, variations in dynamics are reduced significantly. Moreover, differences in peak height now indicate the level of confidence for any of the beat events: the more a peak approximates the maximum at one, the higher is the likelihood of this event being correct.

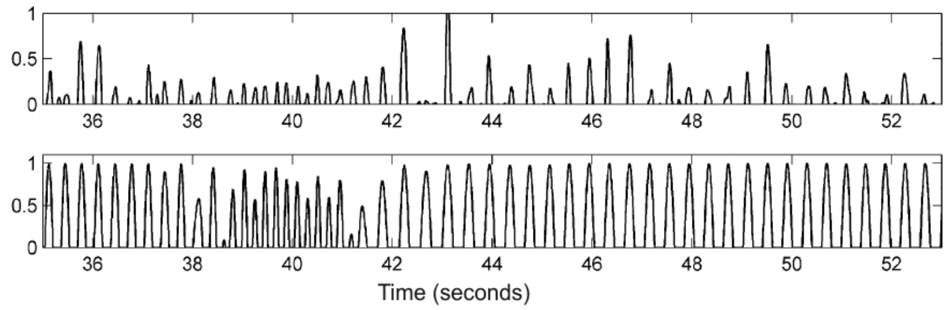


Fig. 2.10 Novelty Function and PLP Curve. Top: original novelty curve; bottom: PLP curve. Reprinted from Müller [44].

2.4.2 Beat Tracking with Neural Networks

Up until this point, the subject of this thesis has been to compare state of the art methods of BT to traditional ones. Although neural networks were first invented more than fifty years ago, it was not until recently that they have been acclaimed as a viable and equally potent alternative for the task at hand. The process of data preprocessing, tempo and beat induction can be substantially different, depending on the actual configuration. Thus, in some cases for example, some fundamentally important steps for traditional BT systems can be skipped altogether. In these cases, the network takes over in controlling some or even all determining factors of the central features that lead to sufficiently close conjectures in the process of BT. However, as will be shown shortly, some researchers have made use of some of the presented methods in earlier sections of this chapter in order to control certain parameters manually or when combining neural networks with traditional BT.

An autocorrelation analysis based approach to BT was first presented by Böck and Schedl [7] at the beat tracking contest at MIREX 2010. Unlike the majority of beat trackers, the proposed system does not possess any onsets extraction capabilities. Instead, a Recurrent Neural Network is trained to conduct a frame by frame classification directly on the basis of the audio signal. Audio segments that serve as input to the network are converted into power spectrograms (omitting phase information) and their relative differences are computed. Taking psychoacoustic knowledge into account, the magnitude spectra is projected onto the Mel scale whereby human loudness perception is approximated. The output of the network, a beat activation function, yields the probabilities of beat events for each frame, ready to be passed onto the peak detection stage. Instead of a simple thresholding of the probabilities, an autocorrelation function is employed to extract the dominant local tempo. This information is consequently used to refine the peak picking mechanism.

Schreiber and Müller [50] present yet another method. It is based on a single Convolutional Neural Network and further strips down the pre- and postprocessing stages. Their completely data-driven single-step musical tempo estimation system requires no additional processing of audio data apart from downsampling the audio signal and magnitude Mel spectrogram conversion. Instead of treating the BT task as a regression problem, Schreiber and Müller [50] opt for the, at first sight, less intuitive alternative of classification, creating 256 classes covering BPM values from 30 to 285. They argue that, with this setup, the system is capable of handling tempo ambiguities and, as a result, the overall prediction stability is increased. To improve training results, data augmentation techniques are borrowed from image recognition systems. Another interesting fact about this system is that it uses parallel network layers with different filter length in an attempt to replicate the behavior of filterbanks in traditional systems.

At MIREX 2014, Böck et al. [6] submitted a tempo estimation algorithm based on neural network and resonating comb filters for determining dominant periodicities in musical pieces. The algorithm does not attempt to deal with varying local estimates but, instead, tries to predict the global tempo with high confidence. At the preprocessing stage, the audio is transfered to a spectrogram representation with overlapping frames and later compressed by using logarithmically spaced filters which have three bands per octave. Additionally, three spectrograms with different frame sizes and their corresponding first-order differences are stacked. The output vector of the network, a recurrent neural network, holds information about the probability of each frame regardless of whether it falls onto a beat position or not. This vector is directly passed into the comb filter which amplifies high probabilities at certain intervals corresponding to lags in the range defined by the filterbank, similar to the autocorrelation function. The accumulated lag values are captured in a histogram where the highest value points to the global beat.

The rhythm tracking architecture by Elowsson [19] represents one of the more complex systems that have been designed by the MIR community. It uses multiple layers of neural networks which build on top of each other and are responsible for different tasks, e.g., beat position identification at frame level, cepstroid[9] identification at frame level, speed detection, and tempo classification. The system is called a "cepstroid invariant neural network," where "invariant" refers to the system's ability to decouple similar rhythmic structures from the tempo of the musical excerpt. This is achieved by first extracting the cepstroid and, in a later step, subsampling the audio of the input vector with the hop size set to a multiple of the cepstroid interval. Essentially, this technique "stretches out" the frame rate to fit the most prominent periodicity which was previously detected. Apart from the division of subtasks between highly specialized ensembles of subnets (the speed subnet alone comprises 60 feedforward networks), a rather sophisticated preprocessing stage makes use of several different strategies, e.g., *harmonic/percussive source separation* and *fundamental frequency estimation* at the audio signal level and the use of specialized flux matrices for building histograms from the data. Furthermore, during postprocessing, the phase of the beat vectors is detected and used to estimate the exact beat positions.

As with the traditional approaches before, the above mentioned beat tracking solutions represent only a fraction of the plethora of neural network based architectures developed in recent years. In spite of the fact that most of these systems keep relying on many of the preprocessing techniques described throughout this chapter, neural networks seem to significantly reduce and simplify onset feature extraction or, in same cases, automate it by moving this step entirely into the domain of the network learning process.

[9]The *cepstroid* is defined by the author as the most salient periodicity of the music.

CHAPTER 3

NEURAL NETWORKS

3.1 Artificial Neural Networks

Artificial Neural Networks (ANN) are loosely inspired by Biological Neural Networks (BNN), e.g., the human brain or brains of any mammal in general.

The fundamental functional unit of the brain is the *neuron* (see Figure 3.1). It has all the features of a regular body cell, but there are some important distinctions. Apart from the cell body, called *soma*, the neuron is able to communicate with other neurons by means of the *axon*, the *dendrites* and the *synapses*. The axon is a long branch serving as an output channel, and it connects to other brain cells, sometimes over relatively long distances. The dendrites form a fine, bushy network of branches surrounding the nucleus and they are the receptors for incoming signals from other neurons. The synapses are the terminal connectors attached to the axon which are tasked with transmitting electrical impulses towards the dendrites of a receiving neuron. The soma is the processing unit, and it is able to send an impulse over the axon that was previously generated by electrical and chemical reactions. To trigger such a signal, the cell nucleus, keeps track of an internal *activation potential* which can increase or diminish depending on the excitatory or inhibitory external stimuli it receives from sensory neurons or other brain cells. Once the activation potential reaches a certain threshold, an electrical impulse is triggered and sent out over the axon, i.e., the neuron "fires."

The more frequently certain neurons are in communication, the more their connections tend to grow, effectively building stronger channels of information. This is what ultimately gives the brain the ability to maintain different brain functions, such as using memory, drawing from previous experience, applying logic to solve a problem or even the more basic control functions such as breathing or walking [11].

A single biological neuron can be connected up to thousands of other neurons forming a huge neural network. In the process of learning, neurons grow new connections and are able to move entire sections of cells to other areas in the brain.

ANNs borrow many ideas from the basic operating principle of a BNN, as described above. It is important to note, however, that ANNs are not meant to replicate the full functionality of a BNN. ANNs are far more limited than their biological blueprints. The human brain is probably the most complex system on Earth – it is a multi-purpose, highly flexible and energy-efficient super computer with an estimated 100 billion neurons[10] and around 600 trillion connections between them [11] [16]. An ANN, in contrast, is designed to solve a specific problem, and it stays confined to this problem once it is built. There are far fewer connections and they do not reconfigure dynamically. Learning is achieved merely by adjusting existing connections by weighting them numerically. Nevertheless, ANNs perform better on a number of tasks for which the human brain lacks speed, endurance, accuracy and storage capacity, among others. Especially when paired with the growing data processing capabilities of modern computer systems, these machine learning architectures offer great potential for tackling challenging problems of the future.

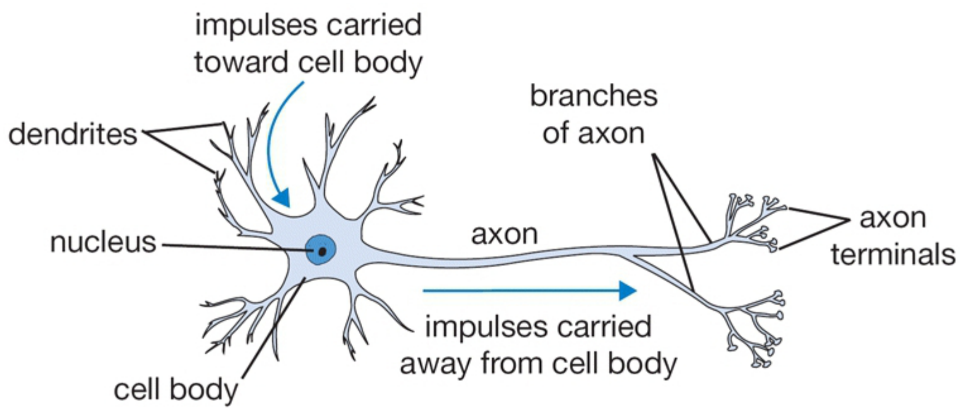

impulses carried toward cell body
dendrites
branches of axon
nucleus
axon
axon terminals
impulses carried away from cell body
cell body

Fig. 3.1 Prototype of a biological neuron. Scientific Figure on ResearchGate. Available from: `https://www.researchgate.net/figure/Biological-Neuron_fig2_322632189` [accessed 15 May, 2019].

[10]When talking about artificial neural networks, neurons are sometimes referred to as *units* or *nodes*. In this thesis, these terms will be used interchangeably.

3.1.1 The Perceptron

The earliest form of a neural network appeared as early as 1943 in a paper published by McCulloch and Pitts. It consisted of a linear model that was able to recognize two different categories. Named McCulloch-Pitts neuron after its inventors, it later developed into the *perceptron*.

In 1958 and 1962, Rosenblatt used the perceptron (essentially a single artificial neuron), for classification of linearly separable patterns. He developed the learning algorithm which empowered the perceptron to learn from input examples and thus delivered the proof of convergence of the learning rule. At this time, of course, instead of being part of an self-optimizing algorithm, Rosenblatt's perceptron was made out of hard-wired circuit boards where operating parameters had to be set by hand. Even though data processing lacked the necessary scale to make the new technology useful then, the case was made that machines could potentially learn by themselves. Amazingly, the perceptron in its original form is still valid as a very basic model for modern multi-layer ANNs [34].

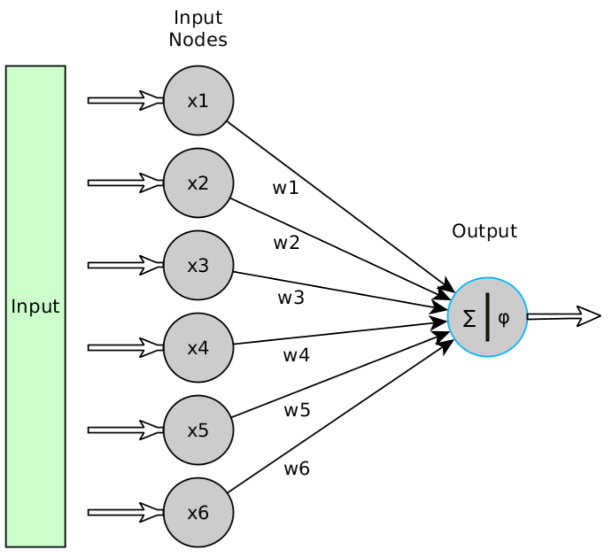


Fig. 3.2 Perceptron receiving an input vector. The output node sums up the weighted inputs and passes them into the activation function ϕ.

The architecture of the perceptron comprises an input layer and an output neuron. The input layer represents the data, i.e., a single training instance at any given time, which is then fed into the perceptron in the form of a vector $X = [x_1, x_2, ..., x_n]$. Every neuron in this layer holds a single value of this input vector and connects to the next layer, which, in this simple case, is the only output neuron. The edges between these layers are the so-called *weights*. The input layer does not do any processing – input nodes only forward their values into the network. The output node multiplies every input with its corresponding weight and does a summation of all these values. The result is then plugged into an *activation function* which decides how the aggregated values will be turned into a single class label. For the perceptron, this was achieved by the use of a *sign function* that maps a floating point number to a set of binary values $y = \{-1, 1\}$.

The equation for the perceptron so far can be written as:

$$f(x_i, w_i) = \sigma\left(\sum_i (w_i \cdot x_i)\right) \quad , \tag{3.1}$$

where the i^{th} weight w is first multiplied with the i^{th} input value x and the result is subsequently passed into the activation function σ.

This model works for very simple scenarios; however, it can be further improved by introducing another component called the *bias*, a constant value which is added to the sum operation just before the activation function. The effect of adding a bias at this point is that the activation function can be shifted horizontally along the x-axis with the result that high values originating from weighted inputs can be attenuated, or, inversely, low input values can be amplified. This concept will become clearer in section 3.1.3 where different activation functions other than the sign function are presented next to their graphs. In essence, the bias is an additional mechanism available to the network by which it can tune the neurons' output. Thus, the key difference compared with tuning the weights is that this adjustment happens inside the neurons and not on the links between them.

With the bias added, the final equation of the perceptron is represented by:

$$f(x_i, w_i, b_i) = \sigma\left(\sum_i (w_i \cdot x_i - b_i)\right) \tag{3.2}$$

Since the perceptron has only one layer which does the actual computations, it is also called a *Single-layer Neural Network* (SLNN). As mentioned before, the perceptron is a linear model; therefore, when training data originates from two linearly separable classes, the perceptron algorithm converges and is guaranteed to find a hyperplane (which marks the decision boundary

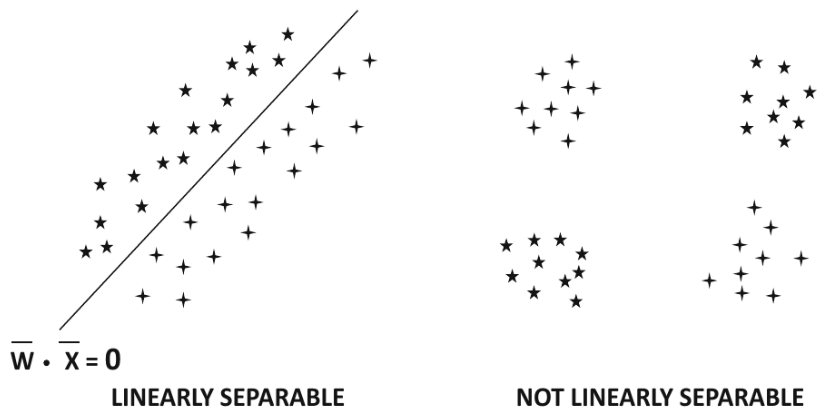


Fig. 3.3 Linearly Separable Dataset. Reprinted from Aggarwal [2].

between these classes). This is in accordance with the Rosenblatt convergence theorem [27]. However, many real-world problems do not confine themselves to a linear problem space. To overcome this limitation, bigger networks with more room for complexity can be constructed by using many of these single-layered artificial neurons. In a *Multi-layer Artificial Neural Network*, every layer, except the input layer, receives inputs from previous ones and passes outputs to the next in line, which in turn receives these as its inputs. On the neuron level, every node in layer n connects to all of the nodes in the next layer $n+1$, thus forming a network of *fully connected layers*. As later chapters will reveal, full (dense) connectivity is not a requirement for all types of layers; however, in the case of simple ANNs, this is the only one. Figure 3.4 shows an example of a *Multi-layer Neural Network*.

Any layers which are neither input nor output layers are called *hidden layers*. This is due to the fact that they are invisible from the perspective of the data on both ends. While the input and output layers are interfacing with the computer program, i.e., receiving and returning data usually in the form of vectors or matrices, the hidden layers in the middle do not interact with the data directly. The number of hidden layers can vary depending on the complexity of the problem to be solved or the number of features which have to be taken into account. It is not possible to predict which part of the problem or feature in the data any of these middle layers are working on since a neural network might split up priorities in a completely different manner from what a human would be able to imagine. One notion is that of every consecutive layer splitting up the information passed on from previous layers in an increasingly integrated way towards the end result. Thus, when considering an image recognition system, for example, the first hidden layer might extract certain features of a face like eyebrows and noses, the second

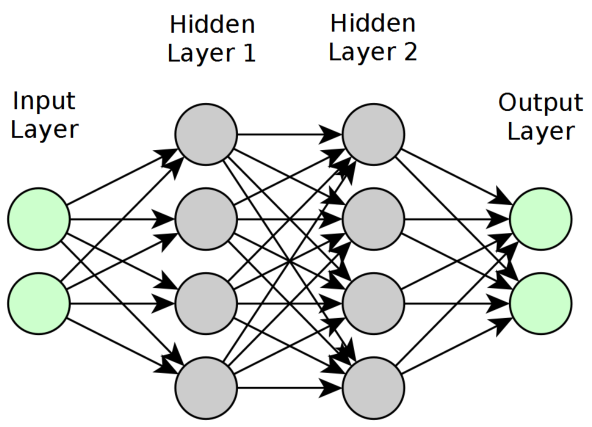


Fig. 3.4 Network of fully connected layers.

layer could be working out how to assemble faces, the third layer might try to detect expressions on these faces, etc. While this idea is certainly a more conceivable way of imagining the inner workings of a neural network, it is unfortunately inaccurate in most cases.

The type of network which has been analyzed so far does not consider any connections in the reverse direction, i.e., from layers closer to the output towards ones that are closer to the input. Since there are no loops, these architectures are known as *feedforward networks*, and they are the most common variant. There are other types as well, e.g., the *Recurrent Neural Network*, which will be discussed in section 3.4.

3.1.2 What does "Learning" Mean?

For a human being, the process of learning is intrinsically related to making errors, repeating all or certain actions and, finally, improving on the task. In *supervised learning*, a neural network learns in a similar fashion, e.g., by first doing some action, then verifying if the outcome of that action matched the expected result, analyzing the error and then making small adjustments to the factors which led to the result. Initially, when the training cycles starts, the adjustable parameters will be chosen at random since the data is unknown and no processing has yet been done. Hence, at the end of the first cycle, the networks' prediction of what the expected data should look like will not be any better than a random guess. With every iteration through this chain of action, however, it gets a chance to tweak the weights and biases nudging them in the right direction in order to be able to make a better guess the next time.

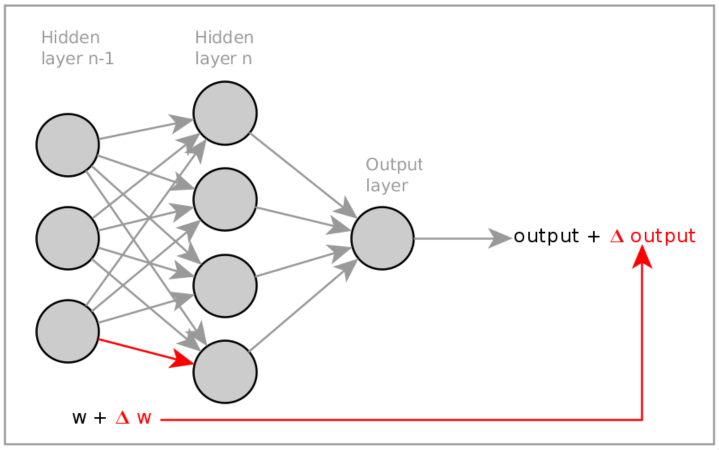


Fig. 3.5 Small changes in any weight will cause small changes in the final output. Adapted from Nielsen [45].

Supervised learning, as opposed to *unsupervised learning*, provides the training algorithm with so-called *labels*. These labels are produced beforehand and independently from the training process. Their format matches the output of the network exactly, e.g., in the case of a binary classifier, the classes might correspond to either a zero or a one. Thus, all labels have to belong to one of these classes too. Training samples and labels are always fed into the network as pairs. This is an obvious consequence of the fact that at the end of the cycle when the network produces a prediction for a particular training sample, this prediction will be compared to the label. The error resulting from this comparison will be fed back into the network and used to update the parameters. When the network achieves to turn the dials in the right direction by revisiting every single weight and bias involved in producing the result, the error at the end of every cycle starts to minimize. After training on hundreds, thousands or sometimes millions of training samples, the network is expected to find the *global minimum* of the error function. At this point, the model is able to make its best predictions not only from the training data itself, but from any new, structurally similar data as well.

This thesis will not go into any detail about unsupervised learning nor any other learning strategies, e.g., *reinforcement learning*, as these are not relevant for the experimental part. At this point, however, it should be noted that neural networks can be trained without providing any labels at all, i.e., "unsupervised." Without any specific labels, the network is forced to set its own priorities by finding patterns which are inherent to the data but possibly invisible to the human observer. The upshot here is that supervised learning is preferable in most situations where the problem is clearly stated, and the classes of the expected outcome are known in advance. On the other hand, if the researcher is trying to reveal new and previously unexpected patterns, unsupervised learning can be a good choice.

3.1.3 Activation Functions

This section will give a brief overview of some of the common activation functions used in modern neural network architectures.

The original perceptron by Rosenblatt used the sign function (see Figure 3.6(b)) in order to assign the output of the neuron to a binary class $\in \{-1, 1\}$ by basically discarding the value and keeping the sign. As with the Heaviside function, the sign function is a step function with the only difference being that the lower limit is set to -1 and not to 0. Both functions are very useful for classifying binary data. Nonetheless, there is one issue with this approach in relation to the network's ability to learn from small improvements: the step function is prone to suddenly flip the output from one extreme to the other even when only a tiny change of a single weight had been made previously. Seen from a mathematical perspective, the step function is not *continuously differentiable*. In MLNNs, choosing differentiable activation functions for the hidden layers is essential since the backpropagated error used for the step-wise updating of weights relies on the activation function gradient [52] [48]. This process will be analyzed in greater detail in section 3.2.2.

To overcome this limitation and to enable networks to achieve better performance gradually by nudging the parameters in the right direction, scientists started to replace the step function with the *logistic sigmoid function*:

$$\textbf{sigmoid:} \qquad \sigma(z) = \frac{1}{1+e^{z}} \quad , \tag{3.3}$$

where z for a single node is defined as:

$$z = -\sum_{j} w_j x_j - b \tag{3.4}$$

Hence, z is the sum of all weighted inputs (minus their biases) that are connected to the node of interest.

To see how the sigmoid function relates to the output behavior of the perceptron, two edge cases will be examined:

- Equation 3.4 returns a large, positive value: $e^{-z} \approx 0$, thus $\sigma(z) \approx 1$.
- Equation 3.4 returns a large, negative value: $e^{-z} \to \infty$, thus $\sigma(z) \approx 0$.

The sign function of the perceptron acts in exactly this way in both edge cases assuming the Heaviside function is used. Very large and very small values *saturate* towards either the constant 1 or 0. Values in the neighborhood of 0, though, show a smooth transition from the lower to the upper boundary, increasing monotonically until flattening out at the extremes.

Figure 3.6(c) shows the graph of the logistic function. Since the range of values stays confined to the interval from 0 to 1, the sigmoid is an example of a *squashing function*, effectively squashing the range of values into a space between 0 and 1. This property enables the sigmoid to be interpreted as a probability estimator for binary classification.

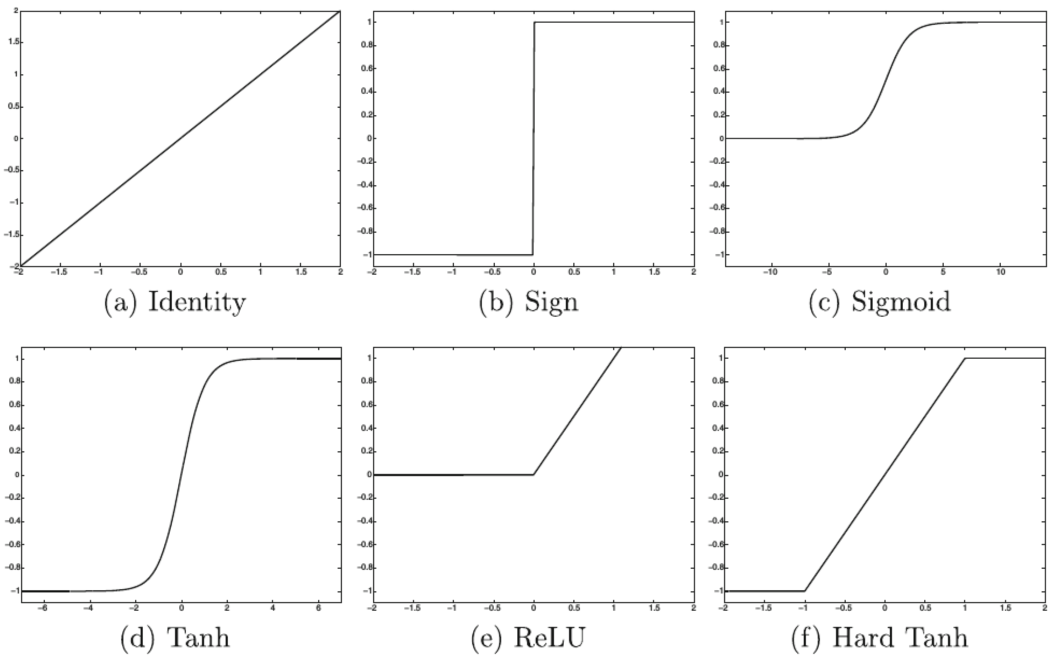


Fig. 3.6 Most common activation functions. Reprinted from Aggarwal [2].

While the logistic sigmoid might be the most convenient choice in situations where it is known in advance that the output lies between 0 and 1 or where a probabilistic outcome is expected, the *hyperbolic tangent function* (Figure 3.6(d)) offers an alternative:[11]

$$\textbf{tanh:} \qquad \sigma(z) = \frac{e^{2z}-1}{e^{2z}+1} \tag{3.5}$$

Comparing the graphs of these two functions, it becomes apparent that the tanh function is, in reality, another type of sigmoid, albeit one that is more stretched out vertically and rescaled horizontally. It could be considered as the non-linear variant of the linear sign function since it also ranges from -1 to 1. Both functions are related by [2]:

$$tanh(z) = 2 \cdot sigmoid(2z) - 1 \tag{3.6}$$

There are several advantages of using the tanh, for instance, Aggarwal [2] and Montavon et al. [43] mention that the symmetric tanh converges faster and that initializing the network with small weights is a more robust procedure compared to the sigmoid.

An example of a non-squashing function is the *identity function* or *unity function* shown in Figure 3.6(a). It is the simplest of all the ones presented in that it just multiplies the input by 1, hence the output is unaltered. This type of activation function is usually used in regression-type networks.

Finally, two piecewise linear functions that have become popular in modern neural network will be reviewed, mainly because of their ease of training: the *rectified linear unit function* (ReLU) and the *hard hyperbolic tangent function* (hard tanh).

ReLU[12] is shown in Figure 3.6(e). In some applications, a saturating function is not the right choice. Computer vision problems and deeper networks face the problem of a *vanishing gradient* whereby the backpropagated error signal has increasingly lower influence over the weights being updated. When an already small gradient is backpropagated, the following gradients which take this first one into account can produce even lower gradients which finally vanish completely causing the training process to halt. ReLU is less susceptible to this problem since it saturates in only one direction (negative values). Furthermore, ReLU is computationally cheaper then sigmoid. This becomes already clear by observing equation 3.7:

$$\textbf{ReLU:} \qquad \sigma(z) = max(0, z) \tag{3.7}$$

[11]In spite of tanh not being a sigmoid function anymore, throughout this work, *sigma* will be kept as an indicator of an activation function in general throughout this work.

[12]ReLU has several variations which will not be discussed in this work.

The hard tanh function is defined as follows:

$$\textbf{hard tanh:} \qquad \sigma(z) = max(min(z,1),-1) \tag{3.8}$$

It is another squashing function (see Figure 3.6(f)) since it limits the outputs from an arbitrary to a bounded range. Like the ReLU function, it is computationally very efficient, even more so than the tanh.

Lastly, the *softmax* function is useful when the output layer is expected to yield probabilities. This function transforms a k-dimensional vector into another vector of the same dimension. All outputs are real values in the range between 0 and 1, and the sum of all *k* values gives 1 [42].

$$\textbf{softmax:} \qquad \sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{k} e^{z_k}} \qquad for \quad j = 1,...,K \tag{3.9}$$

3.2 Training the Neural Network

The idea of gradually improving the network's performance requires a way for it to know what the actual performance is after processing the output of each training sample, i.e., it needs to know the cost of computing the prediction [45]. The *cost function* or *loss function* handles this metric. The cost (or loss) is the error between the prediction and the actual observed value. There are many different ways of calculating loss, e.g., equation 3.10 describes a very common cost function, namely the *quadratic cost function*[13]:

$$C(w,b) = \frac{1}{2n}\sum_x \|y(x) - a\|^2 \quad , \tag{3.10}$$

where w and b are all weights and biases forming part of the network, n is the number of training samples, and y is the label of the training sample x. The letter a denotes the output of the activation function σ, often referred to as the *postactivation* function. Consequently, the input to the activation function, which in this thesis is denoted by z, is called the *preactivation* function. Thus, a of layer l is defined by: [14].

$$a^l = \sigma(z^l) = \sigma(w^l a^{l-1} + b^l) \tag{3.11}$$

[13]Also called the Mean Squared Error (MSE)

[14]To be precise, this equation does not look at single nodes but rather at their matrix representation. σ, the wrapping activation function, is applied element-wise to all matrices.

In Equation 3.10, it should be noted that the cost function is an average over all *n* training samples *x*.

3.2.1 Gradient Descent

Once the training instance (X_i, y_i) has completely run through the network in the forward direction, the loss is computed for every output node and then accumulated. Figure 3.7 depicts graphically how the loss for every output neuron $a_0, ..., a_M$ is computed and then summed up to the aggregated loss by introducing a virtual node serving as an accumulator [47].

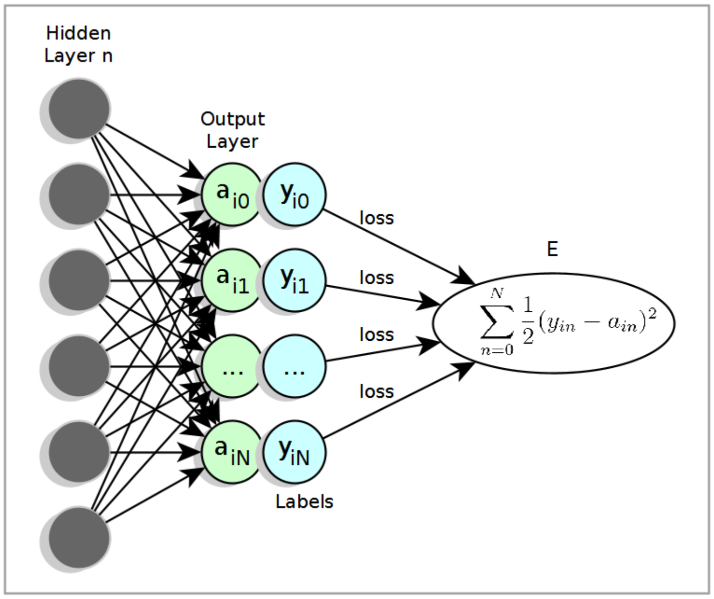


Fig. 3.7 The virtual ∑-node visualizes the aggregated loss of output nodes and labels resulting in the error E.

Effectively, the loss function compares the prediction with the provided class label. It does this by subtracting both entities. As long as the prediction does not match the observed value, the error will be non-zero. Conversely, once the network computes an output closer to the actual data, the loss will approximate zero. Therefore, the goal is to minimize the loss. The technique generally used for minimizing the error of the composite function represented by a neural network is called *gradient descent*.

Updating weights and biases in order to improve the prediction at the end of the next forward pass requires an additional step: the backward pass. During backpropagation of the *error signal* towards every single weight and bias in the network, the *backpropagation algorithm* seeks to determine the degree by which any of these nodes formerly contributed to the overall loss. Gradient descent is applied during backpropagation.

The simplified schematic of Figure 3.8 shows the basic principle of how gradient descent is used to find the minimum of the error function. The abscissa is associated with a single weight in the network, the cost is measured along the ordinate.

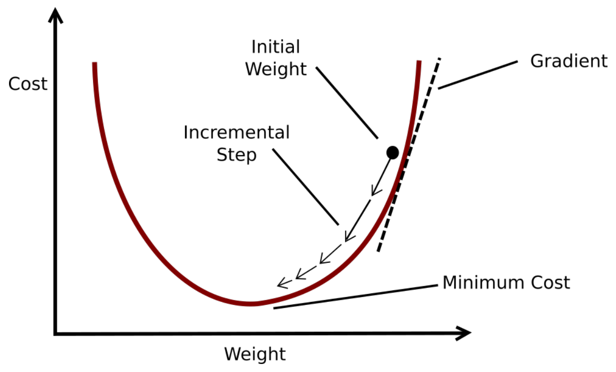


Fig. 3.8 Gradient descent. Reprinted and adapted from Lanham [39].

The initial weight is placed at a random point on the curve, and so, it is clearly not the optimum. In order to minimize the error, this weight has to be moved, one step at a time, down the slope until it reaches the bottom of the curve. The size of these incremental steps is controlled by the *learning rate*, e.g., a high learning rate will result in bigger steps along the curve. The direction in which the weight will be displaced depends on the *gradient*, i.e., a vector of partial derivatives of all the variables involved. In Figure 3.8, the only variable is, evidently, the single weight. In a real setting, however, there will be thousands or even millions of variables. The function in Figure 3.8 does seems to suggest that finding the minimum is merely a question of moving "downwards." While this remains a sensible way of picturing the process in one, two or three dimensions, it is surely beyond anyones capacity of imagination when millions of dimensions are involved. The following section, 3.2.2, studies the algorithm which implements gradient descent in neural networks.

3.2.2 The Backpropagation Algorithm

Let w_{ij} be a single weight somewhere between fully connected layers, leading from node i in layer $l-1$ to node j in layer l (see 3.9).

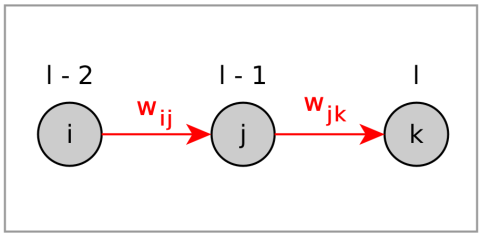


Fig. 3.9 Naming convention of weights and layers.

It is clear that a small change Δ in w_{ij} will affect all other nodes and weights in the chain of connections until it reaches the output layer. This Δ will propagate towards the output and alters its result, which in turn will be reflected in the error after evaluating the cost function.

Thus, the rate of change in the weight is directly proportional to the rate of change in the cost, a fact which leads to the approximate Equation 3.12:

$$\Delta C \approx \frac{\partial C}{\partial w_{ij}} \Delta w_{ij} \tag{3.12}$$

To learn in what way the weight has to be adjusted to minimize ΔC, $\partial C / \partial w_{ij}$ has to be found.[15] However, before weight changes farther back in the chain can be taken care of, all previous parameters on the way, starting with the ones connecting to the output layer, need to be handled first. Therefore, the chain rule is first applied to ΔC with respect to $\partial C / \partial w_{jk}$:

$$\begin{aligned} \frac{\partial C}{\partial w_{jk}} &= \frac{\partial \frac{1}{2n} \sum_j (y_k - a_k)^2}{\partial w_{jk}} \\ &= (y_k - a_k) \frac{\partial}{\partial w_{jk}} (y_k - a_z) \end{aligned} \tag{3.13}$$

It should be noted that the $\sum$ disappears from the above expression. The reason for this is that the derivative is only taken with respect to the j^{th} node with all other nodes being discarded as they will be constants at this moment. In the next step, $a = \sigma(z)$ has to be expanded since a is

[15] To simplify the notation, $y = y(x)$ is, as before, assumed to be the prediction of a particular training sample x. The same is true for a.

not directly dependent on the weight. Again, the chain rule is used:

$$\begin{aligned}\frac{\partial C}{\partial w_{jk}} &= (y_k - a_k)\frac{\partial}{\partial w_{jk}} a_k \\ &= (y_k - a_k)\frac{\partial}{\partial w_{jk}} \sigma(z_k) \\ &= (y_k - a_k)\sigma'(z_k)\frac{\partial}{\partial w_{jk}} z_k \qquad (3.14)\end{aligned}$$

As before, z_j has to be expanded to reveal the independent variable it depends on:

$$\begin{aligned}\frac{\partial z_k}{\partial w_{jk}} &= \frac{\partial}{\partial w_{jk}} \sum_j w_{jk} a_j + b_k \\ &= \frac{\partial}{\partial w_{jk}} (w_{jk} a_j) \\ &= a_j \qquad (3.15)\end{aligned}$$

Substituting Equation 3.15 into Equation 3.14 gives:

$$\frac{\partial C}{\partial w_{jk}} = (y_k - a_k)\sigma'(z_k) a_j \qquad (3.16)$$

To further simplify, all terms depending on the j^{th} neuron in the last layer are bundled into one variable, commonly called δ, which represents the error of the node. The final equation now is:

$$\frac{\partial C}{\partial w_{jk}} = \delta_k a_j \qquad (3.17)$$

Equation 3.17 determines the degree to which any of the weights are contributing to the backpropagated error signal δ by weighting it with the postactivation output of the activation function it is connected to in the previous layer.

For the biases, a similar expression can be derived. The difference in this case is that the last term in Equation 3.14, namely z, will change:

$$\frac{\partial z_k}{\partial b_k} = \frac{\partial}{\partial b_k} \sum_k w_{jk} a_j + b_k = 1 \qquad (3.18)$$

Hence:

$$\frac{\partial C}{\partial b_k} = \delta_k \tag{3.19}$$

Backpropagating the error signal through the nodes of the output layer has proved to be simple since every weight depends only on one activation function so far. However, when propagating the error backwards through the hidden layers $\{l-1, l-2, ..., l-n\}$, it has to be taken into account that all nodes receive inputs from every node in the previous layer because the network is fully connected. The first step in taking the derivative is very similar to Equation 3.13; this time, however, the summation does not disappear due to the above mentioned dependencies.

$$\begin{aligned}
\frac{\partial C}{\partial w_{ij}} &= \frac{\partial \frac{1}{2n} \sum_j (y_k - a_k)^2}{\partial w_{ij}} \\
&= \sum_j (y_k - a_k) \frac{\partial a_k}{\partial w_{ij}} \\
&= \sum_j (y_k - a_k) \frac{\partial}{\partial w_{ij}} \sigma(z_k) \\
&= \sum_j (y_k - a_k) \sigma'(z_k) \frac{\partial}{\partial w_{ij}} z_k
\end{aligned} \tag{3.20}$$

As earlier, with the expansion of the last term in Equation 3.20, z_j can be shown as indirectly dependent on the independent variable w_{ij}:

$$\begin{aligned}
z_k &= \sum_j w_{jk} a_j + b_k \\
&= \sum_j w_{jk} \sigma(z_j) + b_k \\
&= \sum_j w_{jk} \sigma(\sum_i w_{ij} a_i + b_j) + b_k
\end{aligned} \tag{3.21}$$

With the observations made during the expansion of Equation 3.21, the path is laid out for the actual derivative of the last term z_k:

$$\begin{aligned}
\frac{\partial z_k}{\partial w_{ij}} &= \frac{\partial z_k}{\partial a_j}\frac{\partial a_j}{\partial w_{ij}} \\
&= \frac{\partial}{\partial a_j} a_j w_{jk} \frac{\partial a_j}{\partial w_{ij}} \\
&= w_{jk}\frac{\partial a_j}{\partial w_{ij}} \\
&= w_{jk}\frac{\partial \sigma(z_j)}{\partial w_{ij}} \\
&= w_{jk}\sigma'(z_j)\frac{\partial z_j}{\partial w_{ij}} \\
&= w_{jk}\sigma'(z_j)\frac{\partial}{\partial w_{ij}}\sum_j (a_i w_{ij} + b_j) \\
&= w_{jk}\sigma'(z_j) a_i
\end{aligned} \tag{3.22}$$

Substituting this partial result into Equation 3.20 gives:

$$\frac{\partial C}{\partial w_{ij}} = \sum_k (y_k - a_k)\sigma'(z_k) w_{jk} \sigma'(z_j) a_i \tag{3.23}$$

Finally, the terms of Equation 3.23 can be regrouped depending on the layers involved:

$$\frac{\partial C}{\partial w_{ij}} = a_i \sigma'(z_j) \sum_k (y_k - a_k)\sigma'(z_k) w_{jk} \tag{3.24}$$

All terms with a single subindex k can now be merged into the expression δ, as seen before. This is logical because the index indicates that the error is associated with a certain layer. By the same logic, this also works for terms with a sole subindex j. In fact, since the error from nodes in layer l is propagated towards layers $\{l-1, l-2, ..., l-n\}$, these last layers can also be integrated into the δ:

$$\begin{aligned}\frac{\partial C}{\partial w_{ij}} &= a_i\sigma'(z_j)\sum_k (y_k - a_k)\sigma'(z_k)w_{jk} \\ &= a_i\sigma'(z_j)\sum_k \delta_k w_{jk} \\ &= \delta_j a_i \end{aligned} \tag{3.25}$$

Lastly, the equation for the δs of biases in the hidden layers is derived in exactly in the same way as already seen in Equation 3.18 and 3.19.

$$\frac{\partial C}{\partial b_j} = \delta_j \tag{3.26}$$

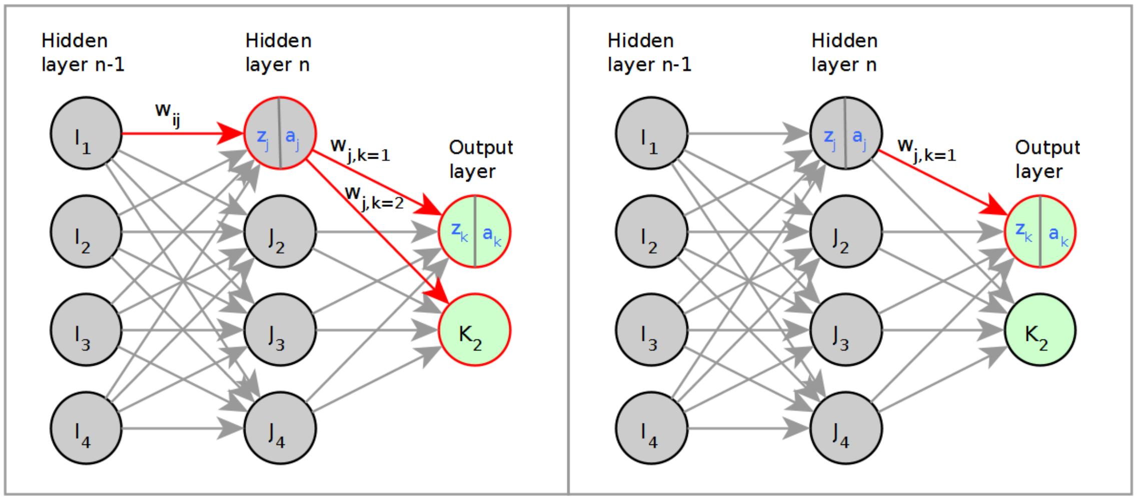


Fig. 3.10 The subfigure on the left shows how tweaking the weight w_{ij} between hidden layers l_{n-1} and l_n affects all nodes in the output layer. Therefore, in Equation 3.20, the summation cannot be discarded since all k_n with $n \in \{1,2,...,N\}$ have to be taken into account. In comparison, when applying backpropagation to one of the output layer nodes (subfigure on the right) only this one node is affected. Thus, in Equation 3.13, which deals with the latter case, the summation is omitted.

3.3 Convolutional Neural Networks

The *convolutional neural network* (CNN) is an extension of the regular neural network architecture in the sense that both share some characteristics while others are specific to the CNN

[35]. CNNs are mostly used in image processing, e.g., for classifying objects, faces or animals, but they are not limited to this. In fact, this more recent type of architecture can be used, more generally, in time series or even audio analysis with spectrograms, etc. CNNs are designed for solving pattern recognition problems; particularly, they need the ability to learn abstract features in high-dimensional input data such as images or video, both in supervised and unsupervised learning.

CNNs are, like regular neural networks, composed of layers which pass a result in matrix form onto subsequent layers. They use a loss function as well as backpropagation in order to update node parameters. Image classification and recognition is a complex task and can be very memory intensive. The content of images is often ambiguous due to shadows, angles, rotations, occlusions, noisy background, etc., and a fully connected network on its own can struggle to model this amount of information. CNNs use mainly three techniques for handling image data and reducing data overhead: *convolution*, *non-linearity* and *pooling*. Figure 3.11 shows a diagram of a basic CNN.

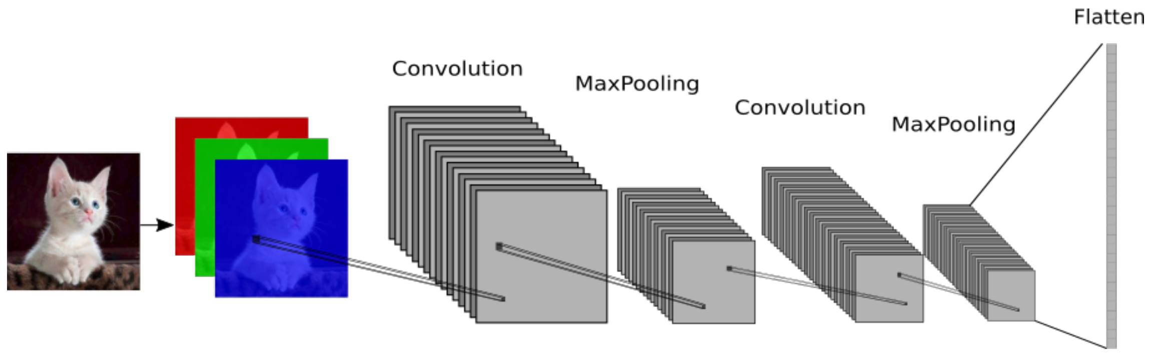


Fig. 3.11 An example of a convolutional neural network. Typically, the last layer of the CNN is flattened and followed by an ANN (not shown in this figure).

3.3.1 The Convolutional Layer

The convolutional layer uses a series of *filters*, also called *convolutional kernels*.[16] These filters (illustrated in Figure 3.12) are convolved, one at a time, with a matching patch of the input array by moving the filters in a sliding window fashion over the entire matrix. Specifically, the convolution operation is accomplished by taking the dot product between the filter and the input patch. Each filter is initialized with random values at the beginning; however, in a similar

[16]In this thesis, filters and kernels will be used interchangeably.

manner to the already discussed weights of fully-connected layers, they represent the learned parameters of the convolutional layer which get adjusted during backpropagation.

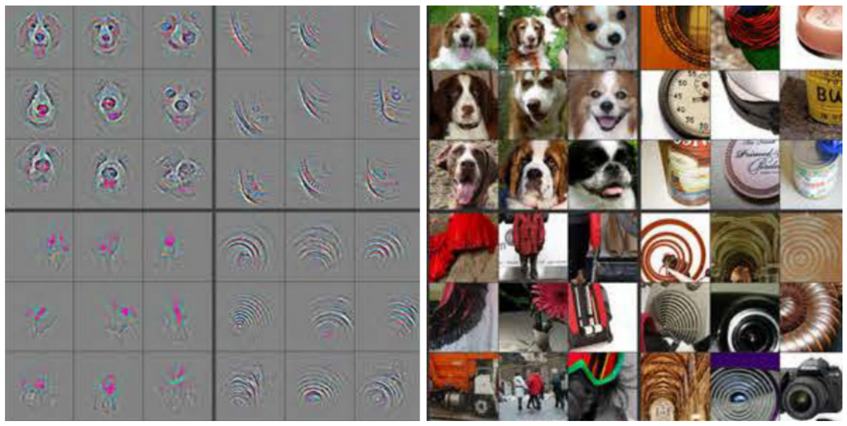


Fig. 3.12 Visualization of filters inside a CNN. Reprinted and adapted from Zeiler and Fergus [54].

To illustrate, Figure 3.13 shows how a filter (in green) is applied to an input array with dimensions of $4 \times 4 \times 3$. The first two relate to the *height* and *width* whilst the third dimension expresses the *depth*[17] (this last dimension corresponds to the *RGB channels* of the image). The filter is required to always match the depth of the input array; therefore, the filter matrix is replicated three times although with different values due to the already mentioned random initialization.

The result of any position the filter visits is stored in a new matrix, the so-called *feature map* as shown in Figure 3.13 in orange. It is two-dimensional only since it merges the depth channels together into one dimension. This feature map is the result of a convolution operation with a filter, and, therefore, the total number of feature maps equals the number of filters in the queue. These feature maps are stacked on top of each other resulting in a 3D *output volume* which is then passed on by the convolutional layer to the next layer. Figure 3.14 illustrates this transformation.

The number of unique positions in which a kernel can be allocated inside the input matrix during convolution is defined by the step size with which it moves forwards (and also downwards when it reaches the end of the current row). The example in Figure 3.13 uses a step size of 2 for the kernel; it moves two positions to the right and also two positions down.[18] This step size is

[17]This depth parameter should not be confused with the depth of the neural network, e.g., the total number of input, output and hidden layers.

[18]The downward movement can be inferred from the remaining empty fields in the feature map.

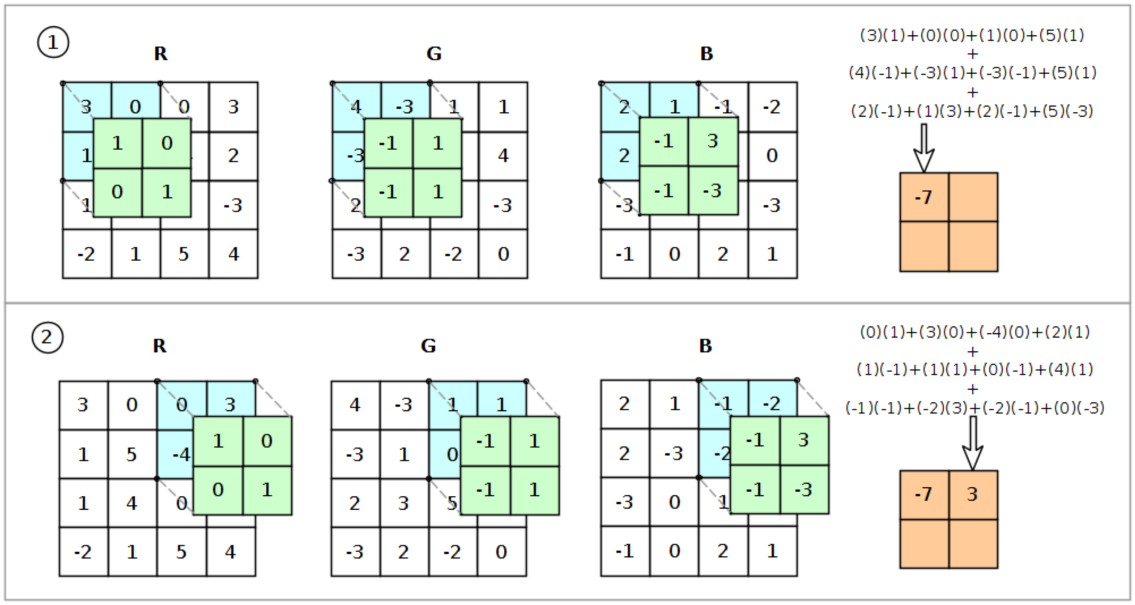


Fig. 3.13 Kernel (green) sliding over 3-channel input matrix with stride of 2. The dot product is taken over all channels and stored in the feature map (orange).

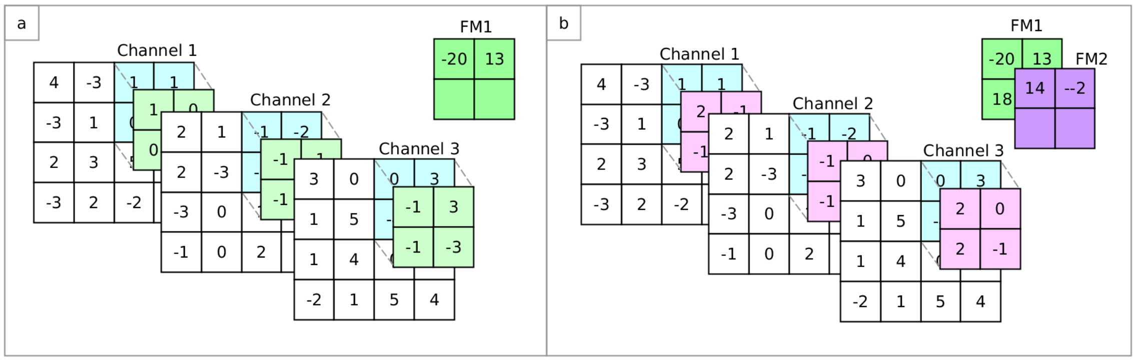


Fig. 3.14 Application of two different filters and the resulting feature maps (second position, stride 2). The stacked feature maps (FM1 and FM2) form the growing output volume of the convolutional layer; (a) snapshot of the creation of the first feature map; (b) the second filter is applied and the feature maps are stacked.

referred to as the *stride*. Common values for the stride are one or two – higher values are rare for reasons described later in this chapter. The horizontal stride is not necessarily the same as the stride in the vertical direction.

Figure 3.13 also shows that filter values do not change during convolution. The same set of filter parameters are shared across all positions, a technique unsurprisingly referred to as *parameter sharing*. One important reason for parameter sharing is that performance is improved since the number of nodes is significantly reduced.

However, parameter sharing also serves another purpose: a filter that has learned to detect a certain feature, e.g., a horizontal or a diagonal edge, might detect a similar feature in other places as well. This assumption is very reasonable because the rows and columns of pixels in the same neighborhood are highly related and derive their meaning from the surrounding context. A feature detection mechanism looks at larger patches of these data points to extract shapes, edges and intensities. Thus, filters which are useful in a certain area are probably useful in other areas too [22].

Filters are usually chosen to be several orders of magnitude smaller than the width and height of the input array of the image. As a logical consequence, filter parameters do not connect to all other nodes in the input array. Instead, they connect to their input matrix counterpart only, at a certain point in time. When the filter is shifted to the next position, some of these previous connections become inactive. This leads to what is called *sparse connectivity* (also sometimes referred to as *local connectivity*).

Sparse connectivity, together with parameter sharing, help to save computational resources by reducing the number of parameters. This does not mean that overall connectivity of the network is poor since there is the possibility of stacking convolutional layers, i.e., when designing CNNs, convolutional layers are often followed by more convolutional layers.[19] Figure 3.15 visualizes how a second layer connects indirectly with a wider subset of nodes from the original data array. Figure 3.15(a) shows a one-dimensional matrix of the original data (white squares in the middle). A 4-unit kernel (orange squares) is convolved along the x-axis with a stride of 1, producing in every step a value of a 4×4 feature map (green squares). The sequence of highlighted red squares marks the patch of nodes that is targeted by the filter in the current operation. This target area, which corresponds to the size of the filter, is called the *receptive field*.

[19]As a side note, between convolutional layers usually a non-linearity layer, e.g., ReLU, is applied, see also 3.3.3.

The resulting feature map, as part of the output volume of the current convolutional layer, is later passed on to the next convolutional layer. The upper right diagram, Figure 3.15(b) depicts the feature map as the new input (in green) to this second layer. A different filter, this time with size 2×2, is applied with stride one, and a feature map (yellow squares) is again created, now with a receptive field of two. The last diagram, Figure 3.15(c) highlights the first unit of this last feature map in blue. By following all paths, starting from the node's direct connections and going farther back through the previous layer, the local branch of connections can be projected onto the original data array. This local area, as shown by the five highlighted red squares in the bottom layer, comprises the *effective receptive field* of the initial node.

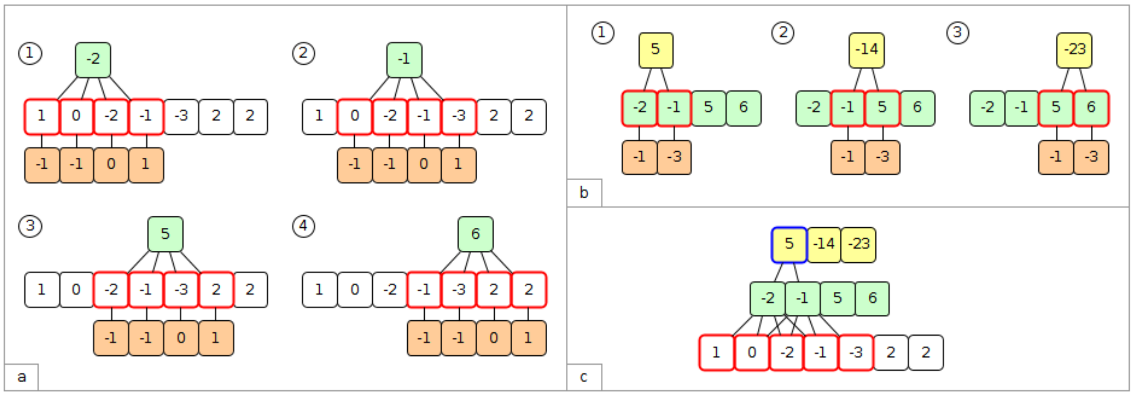


Fig. 3.15 (a) A filter of size 1×4 sliding from left to right over the input array (1×7) with stride 1, producing values of a new feature map (green); (b) a second filter (1×2) in the subsequent convolutional layer is applied to the feature map produced by the previous layer, resulting in a new feature map (yellow); (c) the yellow node with a value of 5 and its effective receptive field (bottom layer, highlighted in red).

Up to this point, the dimensions of the image matrix and the kernels used in the diagrams and examples have been selected in a way such that no part of the kernel surpasses the edge of the matrix. What would happen in the case of a 16×16 pixel input matrix, a 5×5 filter, and a stride of 2? Evidently, the filter would not fit without overlapping; neural network frameworks might handle this situation in different, possibly unpredictable, ways. To avoid ambiguities, it is preferable to set these *hyperparameters*[20] within boundaries that do not permit any overlap.

One problem that arises from the convolution as described above is that the process diminishes the spatial size of output volumes between layers [35]. This happens because the filters are constrained inside the image matrix (or feature map from the previous layer). In the case of

[20]Hyperparameters are parameters which are not learned by the network. They have to be set before training starts.

valid convolution, the output volume shrinks at least $f-1$ pixels (f being the filter size) or more, even when a bigger stride is set. *Zero-padding* the input matrix provides a countermeasure to this effect and preserves the output dimensions, e.g., a padding of 1 means that there will be an additional row or column of zeros on the left, right, top and bottom of the original matrix. These additional rows and columns are then included by the filters in their convolution operations. Apart from valid convolution, two other cases can occur: *same convolution* adds just enough zero-padding so that the original input size stays unchanged, and *full convolution* uses the maximum padding where at least one valid row or column is included.

Identifying the hyperparameters for the input size w, filter size f, zero-padding p and stride s, the spatial size of the output volume can be calculated with the following formula:

$$\frac{(w-f+2p)}{s}+1 \tag{3.27}$$

As an example, in Figure 3.15 a), with input size of 7, filter size 4, stride 1 and zero-padding 0, Equation 3.27 gives: $(7-4+(2)(0))/1+1=4$, which is exactly the size of the resulting feature map (in green). If the same example used a zero-padding of 1 and a stride of 2, the result would be $(7-4+(2)(1)/2+1=3.5$. Since this is not an integer the configuration would not be valid.

3.3.2 The Pooling Layer

Data reduction between layers helps to reduce the computational resources which are needed, improves performance and filters relevant information. However, doing so inside the convolutional layer can curb design flexibility and have other undesired side effects, e.g., "washing away" relevant information present at the edges of the input matrix [33]. The *pooling layer* extracts the task into its own domain – its only purpose is to downsample the output volume while still preserving the significant features gathered by the filters previously. Pooling can be accomplished in different ways, e.g., by taking the average of subsets of input data, L2-norm pooling or applying the MAX operation to these subsets (which is the most common). An example of *MaxPooling* is demonstrated in Figure 3.16.

Similar to kernel windows in convolutional layers, a MaxPooling window strides over the input matrix and produces a pooling slice. Although, in the example shown in Figure 3.16, the stride and kernel size are chosen to avoid overlap, *overlapping pooling* is sometimes used as well, e.g.,

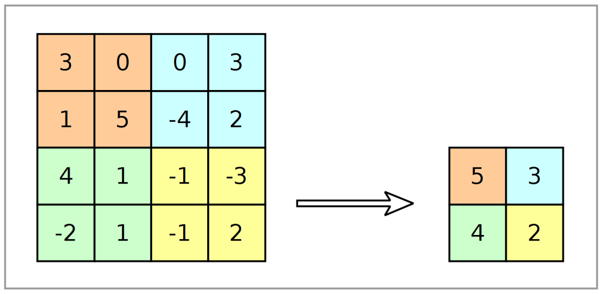


Fig. 3.16 MaxPooling. A sliding window of 2×2 is moved across the input matrix with stride 2. Only the largest values are kept and transferred into the new output volume.

the design of the competition-winning AlexNet by Krizhevsky et al. [38] used a stride of 2 and filter size of 3. Larger filter sizes are considered impractical since they are too destructive [33].

3.3.3 Non-Linearity Activation

The non-linearity activation function is sometimes considered as an extra layer in its own right. Since, in the case of CNNs, non-linearity is regularly applied to the output volumes of convolutional layers, non-linearity activations are often represented as part of these layers.

Krizhevsky et al. [38] found that by using the non-saturating ReLU non-linearity, training speed was improved over the traditionally used saturating tanh function. They argued that their team would not have been able to train such a large neural network (AlexNet) with saturating neuron models. In general, since taking the dot product locally and storing the results in a different matrix is a linear operation, the stacking of multiple convolutional layers would, in practical terms, collapse these layers into a single layer if no non-linearities would be introduced between them.

3.4 Recurrent Neural Networks

A Recurrent Neural Network (RNN) is another type of neural network architecture that addresses an important and inherent limitation of all feedforward neural networks, namely, the inability to include temporal data. Recurrence in the context of neural networks means the revisiting of previously executed functions in order to relate past events with present ones. Feedforward networks pass information forward in only one direction, i.e., any output from layer l feeds into $l+1$ and from there to $l+2$ until the output layer is reached. In contrast, RNNs have loops which allow the output of later layers to be used as input to previous layers. Although sequential data, e.g., time series, may be encoded in the input vectors of feedforward networks and interpreted by using a sliding windows technique, this remains a rather static approach where the placement of the window defines different moments in time [28]. A RNN, on the other hand, does not need to encode temporal information into the input vector since it can find this information on its own. Contrary to feedforward networks, RNNs are not deterministic, i.e., given a piece of information, they do not always produce the same output. Instead, depending on the temporal context, the output can change. Applications include chatbots, natural language processing, sentiment analysis, video and audio classification, handwriting recognition, etc.

3.4.1 Simple Recurrent Neural Network

A Simple Neural Network (SRN) can be thought of as one having a short-term memory. State can be saved to a special layer consisting of context neurons. The next time the network receives input, these context neurons fire and pass their state back to the original hidden layer of neurons, which then receive a copy of their former state. The Elman SRN (Figure 3.17) was introduced in 1990 as the first prototype for modern and more complex RNNs.

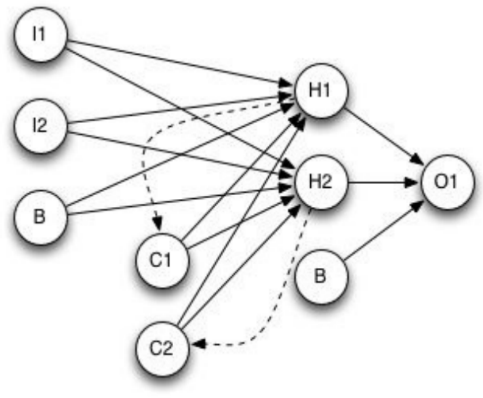


Fig. 3.17 Elman Recurrent Network. Each of the two hidden-layer neurons in the middle is joined to a context neuron (C1 and C2) by a direct, unweighted connection. Both context neurons are fully connected to the hidden layer. Reprinted from Heaton [28].

SRNs work sufficiently well with the most recent state (short-term memory) and could theoretically be trained to recall many steps back in time. However, Hochreiter [29] and others [30] [5] have found that in practical applications, the vanishing (or exploding) gradient problem renders the learning process too slow. Newer RNN architectures have replaced SRNs nowadays, most notably the *Long Short-Term Memory* (LSTM) and the *Gated Recurrent Unit* (GRU). Lipton et al. [40] provides a more in-depth discussion on traditional RNNs.

3.4.2 Long Short-Term Memory

Long Short-Term Memory RNNs (LSTMs) were introduced in 1997 by Hochreiter and Schmidhuber [30] to improve the network's ability to handle long-term temporal dependencies. This is achieved by extending the SRN with a layer of *memory cells* [40]. These special nodes are composed of three multiplicative units called *gates* which control the internal state of the cell.

The state is maintained over successive time steps by a feedback loop: after processing a data point x at time t, a self-connected node, which is part of the memory cell and holds the current state, feeds this state back into itself at time $t+1$. Simultaneously, a new data point x_{t+1} is passed as input to the cell. The weight of this recurrent edge has a fixed value of 1 assigned to it that never changes. This loop, especially important during backpropagation, is called the *constant error carousel* (CEC).

The recurrent call of the memory cell at different time steps is often visualized as an "unrolled" network layer, as can be seen in Figure 3.18 where the same cell is shown at time $t-1$, t and $t+1$.

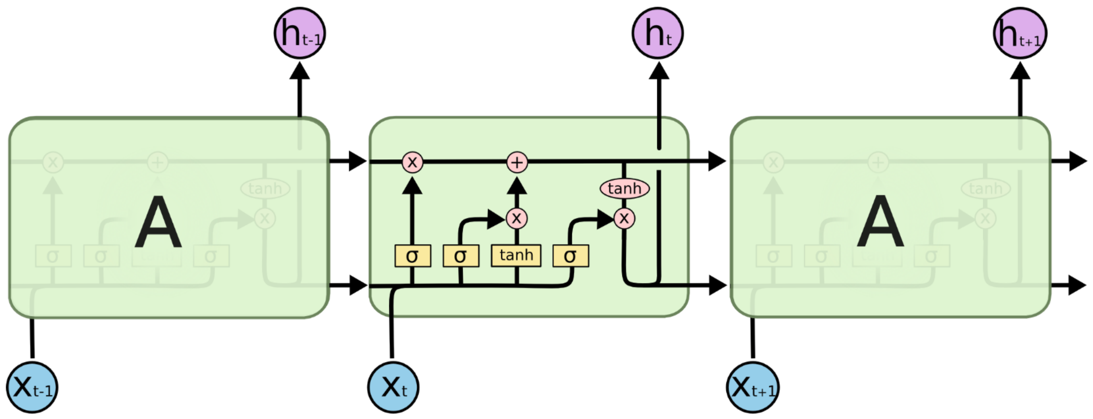


Fig. 3.18 Unrolled layer of an LSTM memory cell. At every time step t, the memory cell receives the input x_t and its own output h_{t-1} of the previous time step $t-1$. The state runs as a horizontal line on top of the gates. Reprinted from Goyal et al. [23].

Figure 3.19 highlights the different functional elements at time t. In the upper row on the left, Figure 3.19(a) shifts the focus onto the *forget gate*. Like the other two gates, it receives input from the current data point x_t and the output of the previous time step of the hidden layer h_{t-1}. These two input streams are merged and then squashed by an activation function (usually a sigmoid).

Let f_t be the output of the forget gate.[21] Then:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \quad , \tag{3.28}$$

where W is the input-to-state weight matrix, and U is the state-to-state recurrent matrix [10].

Figure 3.19(b) demonstrates how the *input gate* i_t (left, sigmoid function) controls the *input node* $\tilde{C}_t$ (right, tanh function). The gate multiplies the weighted input $x_1 + h_{t-1}$ by a number between 0 and 1. Values close to 0 inhibit the data flow of the input candidates $\tilde{C}_t$; conversely, values closer to 1 open up the gate and, as a consequence, $\tilde{C}_t$ will be more strongly represented in the next state C_t.

The equations for the input layer gate and the input layer nodes are:

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{3.29}$$

and

$$\tilde{C}_t = \tanh(W_c x_t + U_c h_{t-1} + b_i) \tag{3.30}$$

Figure 3.19(c) illustrates how the state C_t is updated by the forget gate and the input gate. While the forget gate acts on the old state by scaling it up or down, the input gate first decides internally as to the degree in which the current input is deemed important and, accordingly, forwards these bits of information to be added up with the new state.

Equations 3.28, 3.29 and 3.30 can now be combined to produce the new state:

$$C_t = f_t \odot C_{t-1} + i_t \odot C_t \tag{3.31}$$

where $\odot$ denotes the Hadamard (element-wise) matrix multiplication.

The last element, as shown in Figure 3.19(d), is the *output gate*.

[21] Although the object of analysis in this section is a single memory cell, the components of this cell are organized in layers, similar to what has been discussed in earlier sections. Due to this, all operands in the following equations are assumed to be matrices.

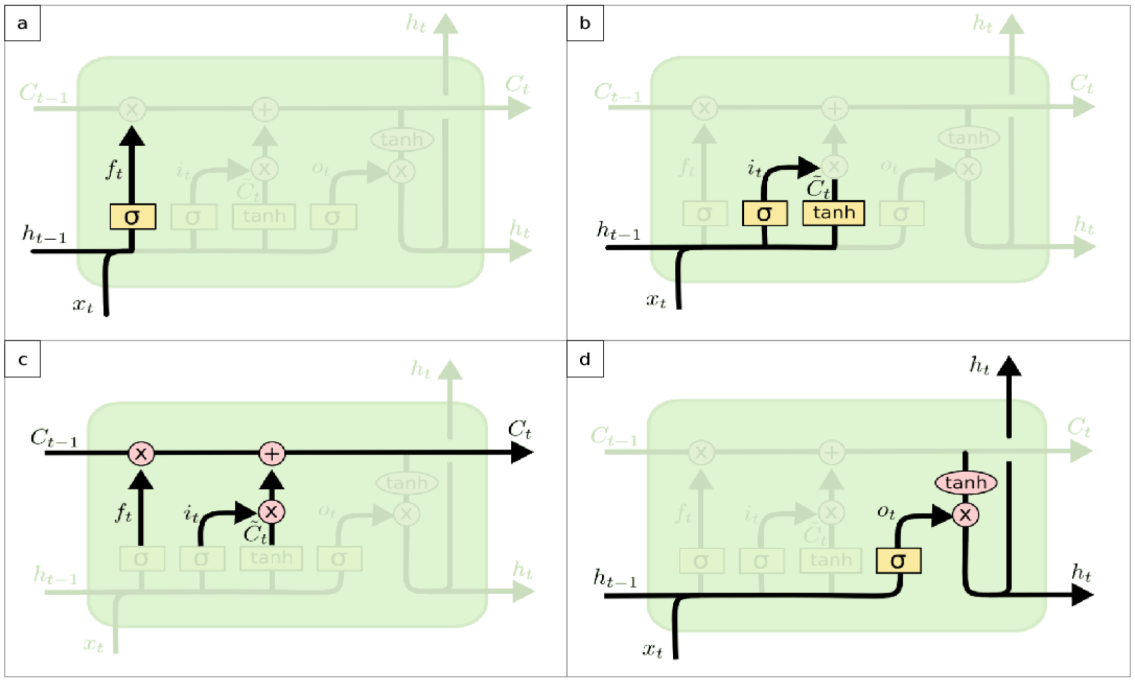


Fig. 3.19 (a) Forget gate; (b) input gate; (c) updating the state; (d) output gate. Reprinted from Goyal et al. [23].

The same principle is applied as with the forget gate and the input gate, namely a sigmoid function scales the output according to the learned parameters (the weights of the output layer nodes) by multiplying the new state. However, since the new state vector is the (element-wise) sum of two other functions, it must be squashed again by, in this case, a tanh function to ensure that the output stays within valid boundaries.

Thus, the output layer gate is defined as:

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \quad , \tag{3.32}$$

The state C_t transitions through the time steps in a continuous loop. As mentioned before, the weight of this recurrent edge is always assigned a fixed value of one. This is important due to the fact that during backpropagation, this constant counteracts the negative effect of the vanishing (exploding) gradient. At this point, it should be recalled that the vanishing gradient occurs when employing activation functions which saturate. This effect is caused by small weights which are multiplied over and over again during backpropagation and which eventually stop learning when their value approximates zero and saturation starts kicking in. The farther away these weights are from the output layer, the slower they learn until they finally "die away." Analogously, the exploding gradient occurs through repeated multiplication of large values. Evidently, this only happens when using activation functions that do not saturate at one.

Shifting the attention back to the state, the CEC (which is the recurrent connection inside the memory cell and over which state travels from one iteration to the next) prevents the degradation of the gradient during long backpropagation cycles. The key here is the fixed unity value of its weight since a multiplication with this weight will never decrease (nor increase) the gradient. Thus, keeping the vanishing gradient problem in check in this way enables the network to span a significantly larger temporal context.

The network can trap the error inside the state for as long as it needs to by simply closing the gates and barring the error from being forgotten or merged together with new input (and previous state). During backpropagation, the error can then be conveyed back to the instant in time where the network needs to allocate it [30] [40].

At its first inception in 1997 by Hochreiter and Schmidhuber [30], the forget gate was not yet part of the design of the LSTM. It was added in 1999 by Gers et al. [21] with the publication of their paper *"Learning to Forget: Continual Prediction with LSTM."* The authors argued that LSTMs, in spite of being able to solve complex long time lag tasks, often failed on learning "very long or continual time series that are not a priori segmented into appropriate training subsequences with clearly defined beginnings and ends." In essence, the LSTM networks needed a way to tag new input sequences on its own since these were not always provided. It seemed that "learning to forget," or more precisely, being able to gradually fade out certain bits of information, was hard or impossible to achieve without a dedicated mechanism for this task. One of the core problems of the earlier design was that the internal state tended to grow in an unbounded fashion without a reset switch. As Lipton et al. [40] points out, the forget gate also proved to be very useful in continuously running networks.

The introduction of *peephole connections* (Figure 3.20) was another improvement added to the LSTM architecture shortly after by Gers and Schmidhuber [20]. These are weighted connections between the gates and the CEC inside the same memory block. Their purpose is to give the gates a way of peeking into the actual state of the CEC which they would otherwise control blindly. The function of the gates is to shield the CEC from unwanted inputs during the forward pass and unwanted error signals during the backward pass. To prevent these peepholes from being used as channels for updating the state, a time lag is introduced during backpropagation.

Further insights on how peephole connections are implemented can be found in the above mentioned publication.

3.4.3 Gated Recurrent Unit

The *Gated Recurrent Unit* (GRU) was proposed by Cho et al. [9] in 2014, successfully simplifying the original LSTM architecture (with the forget gate included). Where the latter had four layers (forget, input, output and state), the most significant achievements of the GRU was to combine the forget and input gates into a single *update gate* and, furthermore, to merge the cell state with the hidden state. Equations 3.33 (reset gate) and 3.34 (update gate) are defined as:

$$r_t = \sigma(W_r x_t + U_r h_{t-1}) \tag{3.33}$$

and

$$z_t = \sigma(W_z x_t + U_z h_{t-1}) \tag{3.34}$$

Equation 3.35 expresses the update candidate $\tilde{h_t}$ for the current t:

$$\tilde{h_t} = \tanh(W_h x_t + U_h(r_t \odot h_{t-1})) \tag{3.35}$$

The activation output of h can then be computed:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h_t} \tag{3.36}$$

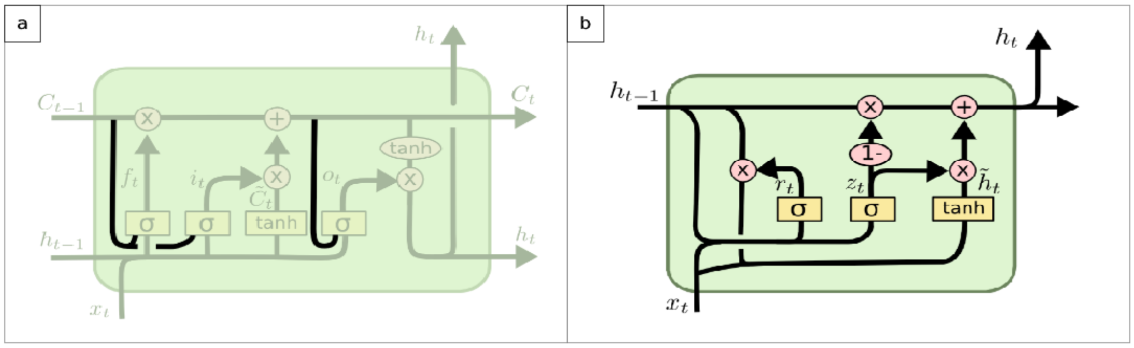


Fig. 3.20 (a) Peephole Connections LSTM; (b) Gated Recurrent Unit. Reprinted from Goyal et al. [23].

Figure 3.20b shows that $\tilde{h}_t$ is dependent on two inputs: x_t and h_{t-1}. The reset gate r_t intercepts h_{t-1} and therefore controls the proportion of the old state in relation to the new input in $\tilde{h}_t$, thus allowing past events to be dropped. The update gate has a similar effect on the state as the memory cell in a LSTM network: old state can be given a high priority while suppressing new events and vice versa (note the complement $1-$ in the diagram).

3.4.4 Bidirectional Long Short-Term Recurrent Neural Networks

In 1997, at about the same time when the LSTM was designed, Schuster and Paliwal [51] published their paper *"Bidirectional Recurrent Neural Networks."* The idea was to embed a wider context into the training of recurrent networks, i.e., taking into consideration not only events that the network had seen already at time frame t, but also the ones it would see in the future. Their first approach was to train two independent networks using datasets with their time axis inverted and to combine them later. Unfortunately, these networks which were trained on the same data were not considered independent anymore and it was unclear how to combine them. The solution was to split the memory cell into two independent cells: one producing *forward states* doing the forward pass in the positive time direction and the other advancing in the negative time direction obtaining the respective *backward states*.

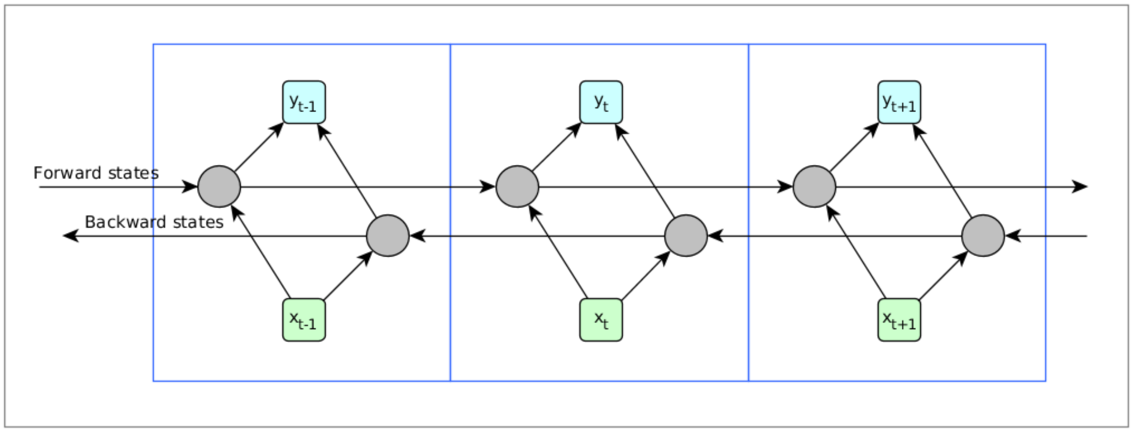


Fig. 3.21 Three time frames of an unrolled Bidirectional Recurrent Neural Network. The network receives input x_T and passes it to the hidden layer (middle). The hidden layer operates in different directions along the time axis, looking at past and future events.

The network layers operate completely separated from each other since there are no connections between them. If the backward oriented layer was removed, the network would resemble a regular RNN. Hence, the backpropagation algorithm (which in the case of RNNs is the

Backpropagation Through Time algorithm) can generally be employed as well to bidirectional RNNs with minor changes applied to it. Further details on these modifications can be found in the original publication [51].

After the appearance and successful implementation of bidirectional RNNs and LSTMs, it is almost self-evident that efforts were made to combine these two. In 2005, Graves and Schmidhuber [24] compared different RNN architectures and evaluated the results. Their benchmarks, obtained from networks carrying out a task of framewise phoneme classification, indicate that *Bidirectional Long Short-Term Memory* RNNs outperform unidirectional networks in terms of speed and accuracy.

CHAPTER 4

IMPLEMENTATION

The following section describes the system which was developed to train two different neural network architectures, a Convolutional Neural Network (CNN) and a Recurrent Neural Network (Bi-LSTM), for the task of automated BT. The goal was to compare their performance by taking an ample variety of musical genres, from classical to popular music with differing degrees of difficulty, and feeding this data into the networks. A brief description will be given of both datasets on which the networks were trained on. Furthermore, the data partitions will be explained as well as the preprocessing of the audio data in order to obtain suitable training samples. Programming-specific details and the concrete implementation will be discussed at the end of this chapter.

4.1 The Datasets

For this project, two different annotated datasets (often used for beat tracking tasks) were selected, and these form the basis for the training and evaluation process of both networks. Firstly, the SMC MIREX beat tracking dataset; it was originally compiled by Holzapfel et al. [31] who described it as an "evaluation dataset comprised of difficult excerpts for beat tracking." The second dataset, known as the GTZAN, was collected by Tzanetakis and Cook [53] with the help of students and researchers all over the world[22]. The SMC MIREX consists of audio fragments of polyphonic, highly expressive and mostly classical music, whereas the GTZAN dataset contains a variety of popular music styles including hip hop, jazz, metal, classical, reggae,

[22]According to the publishing website marsyas.info. Music Analysis, Retrieval and Synthesis for Audio Signals (MARSYAS) is an open source software framework, with the emphasis on MIR tasks.

rock, country, blues, etc. The first dataset is a suitable choice for exploring the limits of beat tracking and to get a better idea of the degree to which performance depends on the particular data. The reason for including the second dataset was the wide variety of styles available, and the fact that most excerpts rely upon a clearer and beat-based rhythmic structure (which makes onsets easier to detect). Preprocessing was easy for both datasets in that annotation times could be extracted without much effort. The GTZAN had the added convenience of using a JSON based annotation format, making all related information (such as metadata and different beat levels) available as a dictionary after being parsed.

	SMC MIREX	GTZAN
Audio Excerpts	217	1000
Length	2h 25m	8h 20m
Difficulty	hard	normal
Styles	Classical	Mixed
Test	50	50
Training	167	950

Table 4.1 Datasets

4.1.1 Partitioning of Datasets

Any dataset used in the supervised training process of a neural network and its derivation of the final model has to be partitioned into different subsets. Each of these sets serve a special purpose at the different stages of *training*, *validation*, and *testing*. The most obvious one is the training dataset, and it should have become clear from previous sections how the training process works.

The validation set is used during training as well but, instead of running through the network, it is used for occasional evaluation. However, this evaluation is not unbiased since the networks "sees" this data sometimes during training. It is still useful to do this evaluation because it simulates how the model acts upon new data. The use case here is that by observing results on validation data, insights can be gained about the hyperparameters, e.g., the number of nodes of hidden layers and how they can be adjusted to improve results. Another useful feature of validation data is *early stopping* whereby the training can be stopped automatically when, based on validation data, it is detected that no significant improvements were made over a certain number of *epochs*. An epoch describes a complete run-through of the whole training dataset. Usually, successful training relies on completing several epochs where the actual number depends on a multitude of problem specific conditions.

Once the model is fully trained, the test dataset is used to evaluate its performance on completely new data. It is important that the test set is not leaked into the training process before this final evaluation to ensure that it is unbiased. Only an unbiased evaluation guarantees that the model is useful for genuinely predicting new and previously unknown data outside the laboratory environment.

In this project, the dataset was divided up into training, validation and test as follows:

Training set	Validation set	Test set
$\approx 72\%$	$\approx 18\%$	10%

Table 4.2 Dataset Partitions

4.1.2 Frame-Based versus Excerpt-Based Datasets

In a first approach to define the data partitions, these partitions were created at frame level. Frame level in this context means that before the training process starts, all frames are extracted from their respective audio excerpts from which they originate and pooled into a single set. The data partitions are then created from this set, effectively disappearing audio excerpt boundaries. An important effect of this approach is that in spite of archiving a clean separation without overlaps (no leaking), neighboring frames, which are almost perfect copies of one another, are spread across all partitions. Furthermore, it is reasonable to assume that in most songs, a certain degree of repetitiveness over long sections is present. Consequently, similar (although not identical) frames would also be distributed over all partitions without necessarily being neighbors. One possible advantage of this design is the fact that data partitions equally represent all genres. Additionally, biases towards outliers among audio excerpts, which are characterized by very uncommon compositions, are avoided in the partitions.

The second approach to data partitioning is excerpt-based (song level). The difference to the frame-based partitioning is that the splitting of the original dataset is done at the audio-excerpt level. As a consequence, a dataset is exclusively composed of frames belonging to a certain fraction of audio excerpts but not others. The expected advantage of this partitioning scheme was that frames originating from the same excerpts and therefore resembling others very closely are confined within their own data partition.

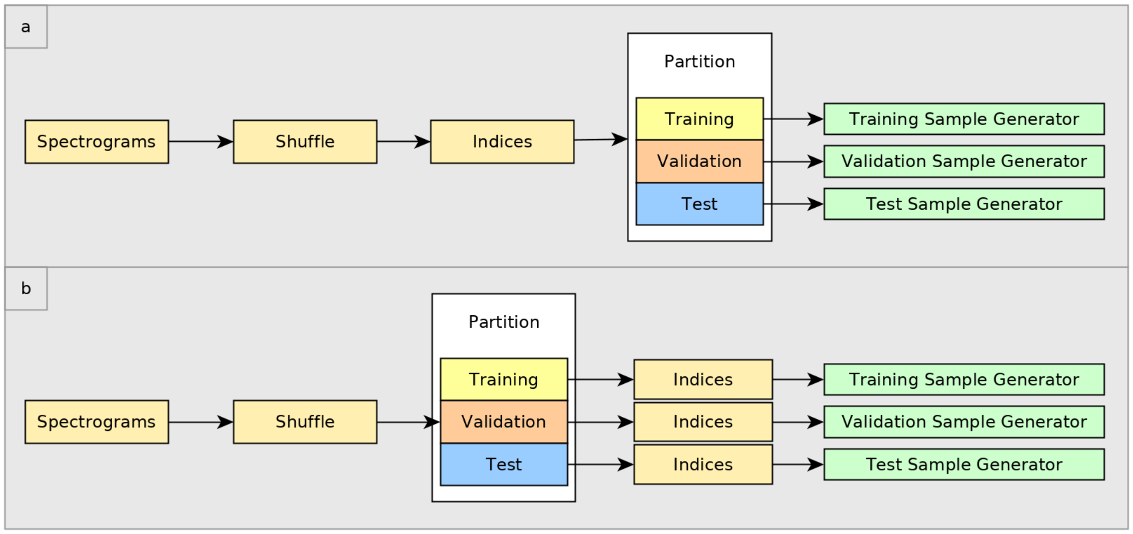


Fig. 4.1 Data partitioning overview. (a) Frame-based partitioning scheme; (b) excerpt-based (song-based) partitioning scheme.

4.1.3 Final Evaluation Dataset

A final evaluation data subset was set aside before the experiments were started. While the song-based data partitioning scheme resulted in rigorously separated training, validation, and testing sets, where highly correlated frames remained restricted to only one of these data domains, the frame-level partitioning scheme was regarded as a possibly weaker segmentation approach. Before initiating the first series of preliminary runs, the potential negative effects of frame-level partitioning could not be anticipated. For this reason, the final evaluation dataset, comprising exactly 50 full audio excerpts of both SMC and GTZAN, was reserved to guarantee the availability of an unbiased dataset at all stages in the process.

4.2 Preprocessing the Data

The step of preprocessing the audio can often be greatly reduced in the case of training neural networks as compared to the traditional methods, where the signal has to be passed through a long chain of preparatory operations (see 2.2.2). The reason is that networks, presumably, detect important features by themselves. Nevertheless, some preliminary steps were necessary before feeding the data into the network. Since the network is unable to interpret the audio data directly, the latter was first converted into a graphical representation: a spectrogram. The spectrogram conversion itself comprises several steps in a chain of different operations which are controlled by a varying set of parameters. Furthermore, training chunks had to be generated to match the input dimension of the input layer and offering enough flexibility to allow for progressively fine-tuning the contextual information which they can provide.

4.2.1 Spectrogram Conversion

For converting the audio signal into a spectrogram, a few parameters need to be set first: *hop size*, *frame size*, *sample size*, and, in the case of using a Mel filterbank (see Figure 4.2), the number of *filter bands*.[23] Hop size, briefly mentioned before in the context of the STFT in Chapter 2, refers to the distance between starting points of consecutive frames. The setting of this parameter usually allows for a significant overlap of frames. The frame size is another variable by which the STFT can be influenced. Thus, a bigger frame size increases frequency precision, and, specifically, if more samples are analyzed, a higher frequency resolution is achieved. The sample rate is given by the audio signal which is used as input to the converter, usually at rates between $44.1kHz$ and $22.05kHz$, as is the case of the datasets used in the project. Naturally, the audio fragments can be further down-sampled to speed up processing time and free up disk space. However, a loss of resolution and detail can potentially counter network effectiveness.

Another commonly applied technique in Machine Learning and Audio Processing is the compression of spectral components aligned to the Mel scale. The effect of a Mel filterbank is one of approximating the non-linear human sound perception, which discriminates lower frequencies at a more fine-grained level than it does with higher ones.

4.2.2 Training Samples

A training sample is composed of an agglomeration of consecutive frames, precisely in the same way as the spectrogram itself. Since a training sample is nothing more than a cutout of a small section of the audio fragments' spectrogram, for the purpose of this work it will be referred to as a "chunk" (see Figure 4.3). The dimensions of a spectrogram produced by the audio library[24] are $frequency \times frames$. The frequency axis is divided up into the indices of frequency bins, e.g., Mel filterbank bins or the original bins which, prior to any filterbank, were produced as a result of the FFT. Frames, on the other hand, can be considered as the unit on the temporal axis.

[23]It should be noted that these are not the only possible parameters. For the scope of this work, the above mentioned seemed to be a reasonable choice to the author

[24]For all audio related task, the excellent and comprehensive audio library Madmom is used.

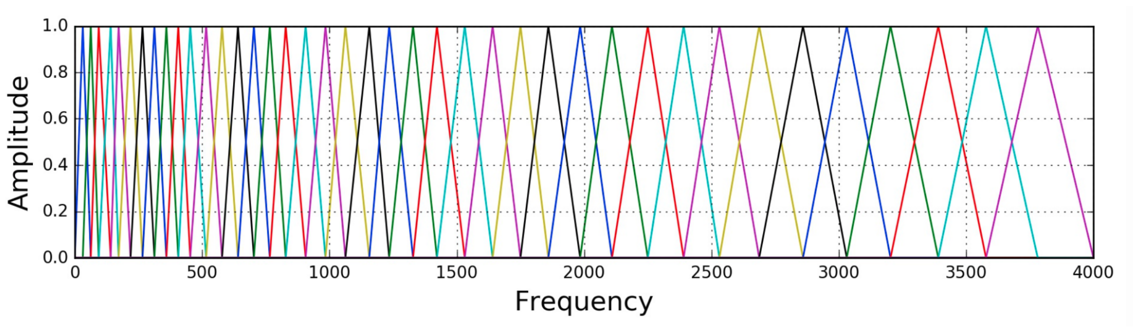


Fig. 4.2 Principle of a Mel Filterbank. A number of triangular filters span from the lower frequency spectrum towards the higher ranges. Each filter is slightly wider than the previous. Retrieved from `https://haythamfayek.com/2016/04/21/speech-processing-for-machine-learning.html` [accessed 15 May, 2019]

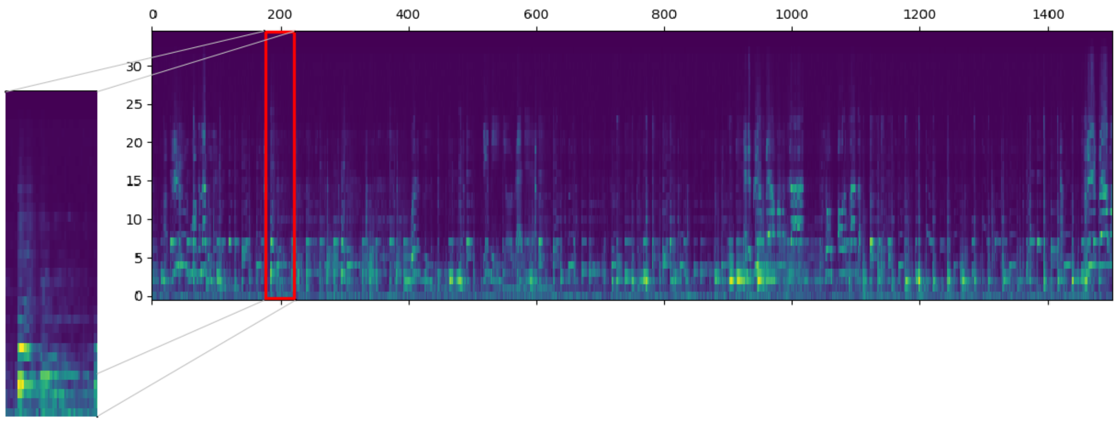


Fig. 4.3 A training sample (or chunk) with center index 200. The chunk expands equally to both sides of the index number by *chunksize*/2. The full spectrogram in this example consists of 1500 overlapping frames.

A chunk always includes the full range of frequencies but only a subset of frames. The example in Figure 4.3 shows a chunk with *chunk size* of 30 taken out of a spectrogram with the size of 35×1500. Thus, the resulting chunk dimension (or shape) is $(35, 30)$.

A second parameter, the *margin*, is used to control context, i.e., deciding which part of the sample is considered to be the actual event and which part is the context in which the event occurs. The margin is subtracted along the temporal axis from both extremes of the chunk and limits the region in the center from where a beat event is detected and interpreted as such. The center region, in the context of this work, will be referred to as the *hot-patch*. Only events falling into the region of the hot-patch will cause the training sample to produce a beat-positive label. Contrarily, if a beat event is detected in one of the margin areas, the label will indicate *false*; however, the event can still contribute to beat detection as part of the context.

4.3 Convolutional Neural Network

The general architecture of the CNN used for this work is composed of the following layers:

- Convolutional (input layer)

$$n \times \left\{ \begin{array}{l} \bullet\ \text{Convolutional} \\ \bullet\ \text{MaxPooling} \\ \bullet\ \text{Dropout} \end{array} \right.$$

- Flatten

$$m \times \left\{ \begin{array}{l} \bullet\ \text{Dense} \\ \bullet\ \text{Dropout} \end{array} \right.$$

- Dense (output layer)

The input layer of the network (a convolutional layer) takes the input shape of the training samples as a mandatory parameter. Subsequent hidden layers deduce dimensions from previous layers. After every convolutional layer, a maxPooling layer is employed to condense the feature maps. MaxPooling is explained in detail in Section 3.3.2.

The first convolutional layer uses 32 filters[25] each with a size of 5×5. For all other convolutional layers, the number of filters are doubled to 64 but the filter size is reduced to 3×3. The effect of this design is that feature detection at first operates at a coarser level (by applying bigger filters) and becomes more granular in later convolution stages. Zero-padding (of type "same convolution") is used only for the last two convolutional layers. Therefore, the first two layers contribute to data compression slightly while the last two keep the layer size constant. In general, design decisions were based on observation and trial runs and adopted for all future runs once the learning process stabilized.

Dropout layers following maxPooling and *dense* layers help to avoid *overfitting*, a condition that occurs when neural networks start memorizing the data instead of generalizing abstract features. A network that is overfitting can predict data it has previously visited almost perfectly with the result that it shows high accuracy and low loss on training data, but increasingly bad results on validation data and, evidently, testing data. The negative effect of overfitting can be significantly reduced by a simple and computationally inexpensive trick: randomly obscuring a certain percentage of connections between layers. Now, these eclipsed neurons cannot be taken into account for any calculations by the network for the current forward pass. This measure counters the negative effect of neurons in the fully connected layers starting to develop co-dependencies between each other. Intuitively, they can be imagined as becoming accustomed to the behavior of their neighbors, consequently adjusting their own behavior in a predictable way and collectively reducing their ability to find abstract patterns. By switching off parts of the network randomly, neurons are forced to achieve similar results through different connections, thus extracting the more general characteristics.

A *Flatten* layer transforms the output volume that originates from a previous convolution operation back into a vector by stacking all matrix columns on top of each other. The reader might remember that a ‘normal’ neural network, also referred to as ANN, receives its input in precisely this form: a vector column representing the matrix data serially.

Dense layers are those that were previously mentioned in this thesis as fully connected layers. "Dense" refers to the level of connectivity between layers. This is in contrast to *sparse* connections in convolutional layers, where not every neuron is connected to every other in neighboring layers.

[25]This paragraph describes the initial setup only, as changes to the evolving architecture were made at a later stage.

The variants of CNNs used in this work use some groups repeatedly. In most of the conducted experiments, the first group (convolutional - maxPooling - dropout) was employed two or three times and the second group (dense - dropout) twice. Results from earlier experiments with fewer layers indicated that the network was not learning fast enough or at all. Thus, the reason for adding more layers (or layer groups) was with the intention of making more room for the network to accommodate a higher level of complexity[26]

The activation function for all layers throughout the network are ReLU functions. Previous training runs used the sigmoid function which seemed to perform slower and was therefore replaced by ReLU. The only exception is the output layer, where the softmax function provides a probability distribution.

In general, layer dimensions were chosen to support a smooth and regular transition from the initial flat representation of the spectrogram's input vector into a increasingly compact cubic shape of output volumes produced by subsequent layers in the forward direction.

4.4 Recurrent Neural Network

The general architecture of the second network, a Bidirectional LSTM, can be summarized as follows:

- Bi-directional LSTM (input layer)
- Bi-directional LSTM
- Bi-directional LSTM
- Dense
- Dense
- Dropout
- Dense (output layer)

The output activation functions for all hidden layers are ReLu non-linearities. The output layer comprises two nodes holding the probabilities for the two types of possible events: beat-positives and beat-negatives. This is achieved by the using the softmax function.

[26]Later experiments, however, showed that too much complexity can also be counter-productive.

The setup described above showed a relatively stable learning progress and was used for most of the conducted experiments with minor modifications at later stages. The bi-directional flow of data, repeated in three different layers of *bidirectional LSTM* layers consecutively, allows for a thorough analysis of temporal features (with the assumption that precisely these features are central for the task). Beat, after all, is a inherently recurrent phenomenon. Moreover, the choice of specifically using bidirectional layers is based on the idea that past and future events, to the same extent, can provide useful information for the evaluation of the current event.

The last layers of the RNN constitute an ordinary ANN, as seen before. Two dense layers are followed by a dropout layer to prevent overfitting, while reducing the layer size from 100 to 32.

4.5 The Beat Tracker with Python/Keras

This section will give an overview of how the program code was structured and detail the relevant mechanisms that were put in place in order to provide both networks with easily customizable training samples. Due to the wide range of experiments which yielded a large amount of data, automatic data recollection was an additional important concern. The system was written in Python 3 using the Tensorflow framework Keras which provides low level network functionality (layer classes, training and evaluations classes, etc.). Audio preprocessing was done with the library Madmom (FFT, spectrogram conversion, filterbanks, etc).

4.5.1 Modules

beattracker.py

The *beattracker.py* module is the entry module. When called directly, it handles command line arguments and sets up the environment. Command line arguments include the type of network (mandatory argument, can be CNN or RNN), the storage environment (store or sandbox), the dataset size (limited_dataset trains a previously defined subset only), the configuration file, whether to overwrite results or resume training based on previous runs, and the option to restart a run with the same configurations (as specified in the configuration file) while creating a new directory for the results.

Based on the provided command line options and the contents of the configuration file that is parsed at the beginning of every execution, other parameters are set by this module, e.g., the dimensions of training samples (which have to be determined dynamically depending on run configurations), the run directory (which is also dynamically set and instantly created as the default location for reports, results, timings, logs and state), dataset split, random states and shuffle settings for data generators, static and dynamic file locations, etc. Some file locations require to be set at runtime as well, depending on the selected set of audio parameters, e.g., hop size, frame size and filterbank settings (Mel).

After partitioning the dataset, data generators are initialized for the three partitions: training, validation and testing. During network operation, these generators provide the batches of data pairs (data sample, label) for their data domain.

Once the environment is prepared, the module corresponding to the architecture is dynamically loaded and layer options are forwarded to it. After bootstrapping the neural network, training is initiated by passing the training generator to the network module responsible for this step.

Once the network model finishes training, the main module *beattracker.py* triggers the evaluation of results, passes a test generator into the evaluation method of the network module and, finally, recollects results and logs for compiling a report with all relevant information.

config_parser.py

With flexibility and best coding practices in mind, configuration options are kept separate from functionality. The *config_parser* is a module that uses recursion to parse a global JSON configuration file and converts the resulting simple dictionary into a dictionary-like data structure, resembling a modifiable and versatile configuration object. Subtrees of this object can be updated, subdivided, addressed and merged together easily, thus allowing for a flexible and consistent handling of configuration options. Command line options are integrated (merged) into the configuration object and accessed from any module later. Network layers, along all layer-specific options, are stored in this object and used to build the network entirely from a file, thus avoiding the need to hardcode the architectural design or any of its options.

As an added convenience, all configurations are stored internally as objects themselves and linked together as an object chain (which is especially useful inside an interactive shell), like iPython/Jupyter. The config_parser supports backlinks, turning the configuration object into a kind of "intelligent" dictionary where every property can look back in leaf-to-root direction. The object chain makes manual inspection a fast and easy task since objects are separated by dots rather than brackets and quotes, as is the case with dictionary items.

file_object.py

Before the beat tracker can be run, audio files have to be converted into file objects and serialized into byte-streams to be stored on-disk as pickle files. A file object contains spectrograms in different frame size resolutions and, after scanning the annotation file which corresponds to the audio, keeps a mapping from beat events to the frames embracing these beat locations. Since file objects have to be kept available during the whole training process (and later during testing), "caching" them as pickle files eliminates the necessity to initialize them from scratch every time they are requested by a data generator.

data_generator.py

The large amount of training samples needed to effectively train the networks can cause problems on some machines when trying to fit this data into memory all at once. Keras provides customizable data generators for a range of use cases, additionally enabling the program to take advantage of the machine's multiprocessing capabilities. A well documented example is the *flow_from_directory* generator, part of the Keras ImageProcessing Library, where training examples are retrieved from a directory as needed. Unfortunately, for the current system, these generators are not a viable choice. Due to the sometimes large overlaps (up to 89% with hop size 441 and frame size 4096), training samples for the beat tracker exhibit major redundancies with respect to neighboring samples. Consequently, storing all samples on-disk, even if admittedly faster, would result in substantially inefficient resource allocation. In simple terms, statically stored training samples would most likely exceed memory capacity or be constrained by a reasonable level of storage efficiency.

The custom made *data_generator* retrieves previously cached spectrograms and generates domain-specific data samples on the fly, each sample based on the current run configuration (hot-patch size, margin, etc.). One reason for the necessity of generating training samples in real-time is determined by the fact that labels for comparable, same-size data chunks are not static but depend on the hot-patch definition, e.g., a smaller hot-patch, as part of a larger spectrogram chunk will less likely label a training sample as beat-positive compared to a bigger hot-patch. Since the bigger hot-patch includes more frames, with each frame potentially being associated with a beat event, the *true* label is dependent on chunk and margin settings (see Figure 4.4).

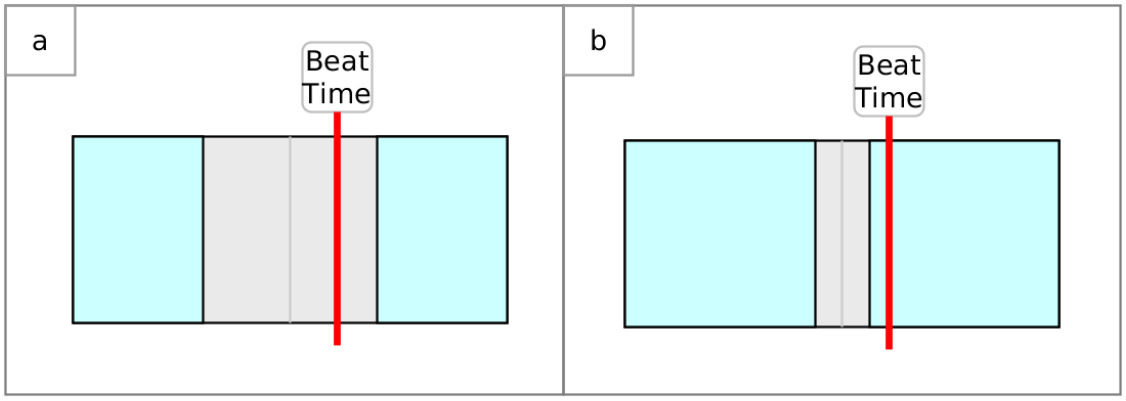


Fig. 4.4 Beat times inside the hot-patch (gray) qualify the sample as positive, otherwise as negative. (a) Narrow margin (b) Wide margin.

The Keras *fit_generator()*[27] is responsible for processing batches of training samples. The *data_generator*, which delivers the data, configures itself from the configuration object and determines the batch size *n*. During an epoch, a generator forwards the full range of training samples that can be drawn from the domain-specific dataset, bundled together as batches. To produce a new batch, it selects *n* frame indices, unpickles the file objects where the frames pointed to by these indices can be found, and cuts out the training samples (chunks) from the file object's spectrograms, centering these chunks around every frame index.

nn.py

The *nn.py* module contains the class *NN*, a superclass from which both the CNN and the RNN networks are instantiated. The *NN* superclass hosts several methods needed by the network subclasses, such as *train()*, *evaluate()*, *print_architecture()*, *save_model*, and a series of utility methods. Functionality that is not shared by the CNN and RNN subclasses is defined, respectively, inside their own classes. This design aims to facilitate clean namespaces and inheritance-based, modular expandability.

custom_callbacks.py

As the name already suggests, this module accommodates all custom callbacks, in particular, the callbacks used by the *train()* method, part of the class *NN*. The following custom callbacks are executed after every epoch:

[27] In statistics, *fitting* refers to approximating a target function.

ModelState: Saves the current state (number of epochs, evaluation results) in case that training is interrupted and has to be restarted.

SensitivitySpecificity: Outputs sensitivity (proportion of actual positives that are correctly identified), specificity (proportion of actual negatives that are correctly identified) and the confusion matrix (false negatives, true negatives, false positives, true positives) to stdout.

HistoryCheckpoint: Stores a dictionary with validation results. Useful for plotting learning progress after training.

TimeHistory: Records elapsed time per epoch. Useful for estimating total training time and runtime comparisons between different network configurations.

Additionally to the custom callbacks, two Keras callbacks are executed after every epoch: the *ModelCheckpoint* saves model and weights whenever a validation improvement is achieved, and the *CSVLogger* writes training/validation loss and accuracy data to a csv file.

helpers.py

This model provides the namespace for a collection of helper functions that can be used across all modules. Examples include the functions for pickling and unpickling file and dictionary objects and creating missing directories silently (error checking included).

preprocessor.py

The preprocessor is run initially, at least once before the first run of the beat tracker main module, and later whenever file object properties change. It serializes all file objects as pickle files and creates a storage path that uniquely identifies the file object properties under which these pickle files are then stored. Once all file objects are built from their raw audio data and pickled, a mappings dictionary is pickled under that storage path. Each mapping provides a sequential number by which every single frame in the combined data partition can later be uniquely identified. When a data generator requires a frame, the mappings dictionary key is used to locate the file containing the requested frame. Then the file path is read from the dictionary values and passed into the generator, which consequently proceeds with unpickling the file.

runner.py

The *runner.py* module is a small external tool that can run a series of configuration files (JSON) from a directory without supervision. When running this configuration dispatcher, no command line arguments can be provided since arguments only make sense when running a program manually. Some of these options are required nevertheless, e.g., the type of network that should be built and trained. The runner can read these arguments, which normally would be specified manually, from the same configuration file. In case one of the files contains an error and prevents the beat tracker from executing, the ensuing exception is caught and execution jumps forward to the next file on the run list.

4.6 System Overview

The general beat tracking process, in the context of this thesis, can be divided into four stages (see Figure 4.5). Only the first two stages of this process, the audio preprocessing and neural network stages, are currently addressed by the developed system.

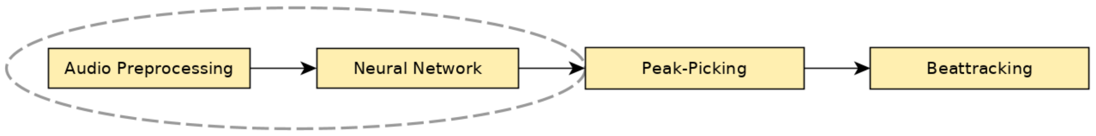


Fig. 4.5 The four main stages of the beat tracking process.

Figure 4.6 shows a detailed overview of these two beat tracking stages, depicting all major processing steps and their integration into the main five processing stages of the developed system. The figure also illustrates how the system is controlled by a global configuration object, which is parsed from a JSON file and consequently used to bootstrap any of these components.

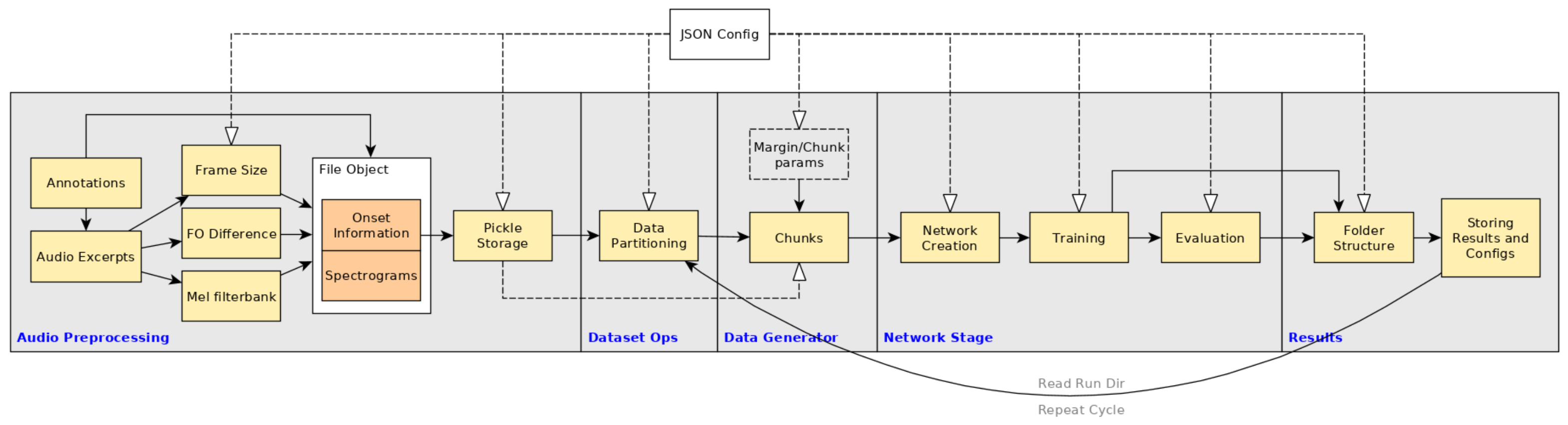


Fig. 4.6 System overview. The execution flow is divided into five processing stages: audio preprocessing (asynchronous), dataset operations, data sample generation (data generator), the network stage, and results. The first stage can be considered as asynchronous with respect to the main execution flow since creating the file objects has to be done prior to executing the beat tracker. For a more detailed look at the dataset operation stage, see Figure 4.1. Once a network configuration has completed the evaluation and the results are stored, the run configurations directory (also called the queue) is checked for any new configuration files to be processed. If new files are present in the queue, a new training process is launched from them.

CHAPTER 5

EVALUATION OF RESULTS

Two main groups of adjustable parameters can be identified for the training process of both the CNN and RNN networks: audio parameters and network hyperparameters. In order to restrict training to a reasonable period of time, the number of these parameters had to be reduced in the experiments conducted for this thesis. The main focus was directed at tailoring audio training samples with differing characteristics which consequently could be used to analyze the relationship between sample length, frame density, filterbank settings, spectrogram reduction methods, etc. and the learning process. It was found that a feasible strategy would be to first build two relatively simple networks and improve them gradually by using a very simple set of training samples. In a second step, when both networks attained a stable and comparable training performance, these networks would remain mostly static, and the tweaking of audio settings would be prioritized for all future experiments.

The only network hyperparameter which was subject to further experimentation in this last stage was the *batch size* for the mini-batch gradient descent (explained below). The audio parameters include the number of *frequency bins*, usage of a *filterbank* (Mel), sample *frame size*, sample *chunk* and *margin* settings (hot-patch configuration), and whether or not the first-order difference was applied to the spectrogram. From a set of preliminary runs[28], the highest-performing parameters were identified and kept for future runs. For all preliminary runs, a downsized[29] dataset (training and testing) was prepared in order to avoid excessive training

[28]In this thesis, the execution of any particular network configuration is referred to as a *run*. A series of runs will be called an *experiment*. The terms "training run" and "test run" are avoided since they imply that the configuration in question is only used in the context of one these domains.

[29]In the case of the GTZAN dataset, the downsized version contained only one fifth of the original number of audio excerpts.

times. The data partition scheme adopted for the preliminary runs was chosen to be at frame level. The difference between frame-based and excerpt-based (or song-based) datasets was explained earlier in Section 4.1.2.

5.1 F-Measure

A commonly used metric for statistical analysis of binary classification is the *F-measure* (also f_1-score or F-score). While there are many more useful metrics for the task of BT (see Davies et al. [12]), for the scope of this thesis only the F-measure was used in the evaluation of results. Sasaki et al. [49] define the F-measure as the harmonic mean of *precision* and *recall*:

$$F = \frac{2pr}{p+r} \tag{5.1}$$

Recall (also known as sensitivity) describes the proportion of correct beat detections[30] in relation to all beats (which should have been detected):

$$r = \frac{c}{c+f^-} \quad , \tag{5.2}$$

where f^- stands for *false negatives*.

Precision describes the proportion of correct beat detections with respect to all beat-positives returned by the classifier (sum of *true positives* and correct detections):

$$p = \frac{c}{c+f^+} \tag{5.3}$$

5.2 Preliminary Runs – CNN

The first phase of experiments was set up to find a simple, viable, and stable network configuration. It was divided into three stages: CNN architectures, frame-based versus excerpt-based dataset partitioning, and RNN architectures. The fixed conditions during all three stages were

[30]Davies et al. [12] note that a detection is usually interpreted as "correct" when it falls into a *tolerance window*. The reason for this is the aforementioned expressive performance of most musical pieces. In this project, the adjustable hot-patch window functions as the element that comes closest to the concept of a tolerance window.

defined by the use of the limited dataset and an unaltered network architecture. In the remainder of this thesis, the preliminary runs are also referred to as phase I.

5.2.1 Mel Bins

The first training runs were aimed at determining the effectiveness of applying a filterbank, in this case, a Mel filterbank. The number of bins, passed in as an argument to the spectrogram conversion function, resolves into the distribution of bins along the frequency axis of the spectrogram. For this experiment, a bin number of 40 was found to be a good middle ground since it preserves a relatively high-frequency resolution while considerably reducing the total number of bins, which amount to more than 70 when the filterbank is omitted. The training process was executed for a total of 20 epochs.

	20/6	40/15	80/35
No Mel	0.90 / 0.83 / 0.86	0.93 / 0.91 / 0.92	0.94 / 0.93 / 0.94
Mel 40	0.89 / 0.82 / 0.86	0.92 / 0.90 / 0.91	0.95 / 0.94 / 0.94

Table 5.1 F_1-scores for the Mel filterbank experiment. Mel 40 indicates that a filterbank with 40 frequency bins was used. The three slash-separated values in each field express f_1-scores for the categories of beat-negatives (0), beat-positives (1) and the weighted average (avg f_1-score) of both categories (in this order).

Three different hot-patches were defined for this experiment. The first hot-patch resulted in a chunk size setting of 20 frames and a margin of 6 frames; hence the notation (20/6) is used. The hot-patch size of 8 can easily derived from these numbers by subtracting both margin areas from the chunk size: $20 - (2 \cdot 6) = 8$. The second hot-patch spanned 10 frames (40/15), as did the third (80/35). From looking at Table 5.1, it becomes clear that, without exception, the f_1-score increased when larger chunk sizes were selected, regardless of whether a filterbank was applied or not. It is also noticeable that the category of beat-negatives scored better than the category of beat-positives in all cases. It should be noted, however, that these categories did not turn out in equal numbers. This can be seen in Table 5.2: beat-negative events represented approximately 60% of the total number of the captured events when used with the smallest chunk, while in the case of 40/15 and 80/35, this number decreased to around 54%. The total number of samples and the numbers per category are referred to as *support numbers*, a topic which will be the subject of a later discussion.

When comparing results obtained in the Mel filterbank experiment, it can be observed that the network, when trained with the unfiltered audio excerpts, performed only slightly better than or

	20/6	40/15	80/35
No Mel	3932 / 2468 / 6400	3449 / 2951 / 6400	3471 / 2929 / 6400
Mel 40	3931 / 2469 / 6400	3437 / 2963 / 6400	3492 / 2908 / 6400

Table 5.2 Support numbers for the Mel filterbank experiment. The three slash-separated values in each field express support numbers for the categories of beat-negatives (0), beat-positives (1) and the total number of generated samples (in this order).

the same as during the training with the filterbank applied. However, training times decreased by more than half when the audio was compressed into Mel bins, as Table 5.3 shows. This suggests that higher-frequency bins did not contribute to the same degree to successful feature extraction as lower frequency bins did and thus can be regarded as redundant data. This finding is supported by the psycho-acoustic knowledge (discussed in Chapter 2) that it is mostly the lower-frequency spectrum which contains information relevant to beat detection[31].

	20/6	40/15	80/35
No Mel	13.2	30.3	64.6
Mel 40	5.6	12.5	26.2

Table 5.3 Mel filterbank experiment: epoch times (in minutes) for three different chunk sizes.

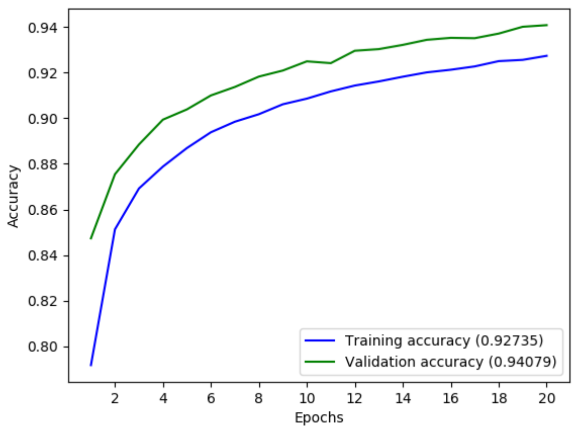
Based on these findings, most later experiments have been conducted with the audio excerpts previously passed through a Mel filterbank.

Figure 5.1 shows that both networks seemingly converged in a rather smooth and regular fashion during training. However, the fact that validation accuracies and losses constantly lagged behind the corresponding training results by a small margin delivered an early indicator that both networks were potentially susceptible to overfitting.

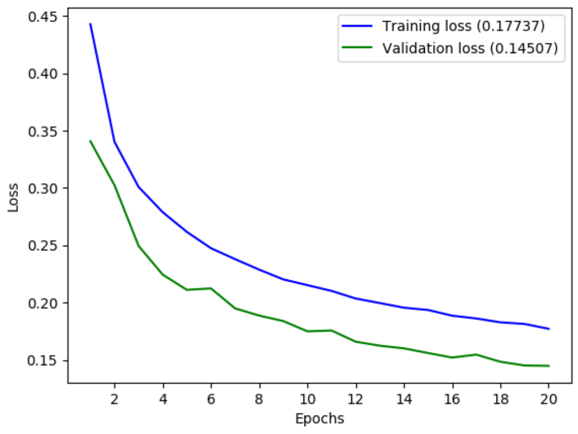
5.2.2 Batch Size

The *batch_size* parameter determines how many training samples are propagated through the network during the forward pass before their accumulated errors are averaged by the loss function in order to initiate a single backward pass. This process is called *mini-batch gradient descent* and represents a highly memory-efficient learning mechanism. This batch-processing

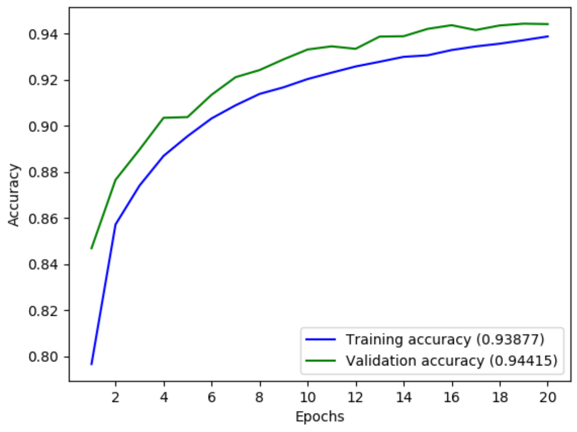
[31]It should be remembered that the Mel filterbank applies a much finer resolution to the lower-frequency spectrum than it does to the higher spectrum



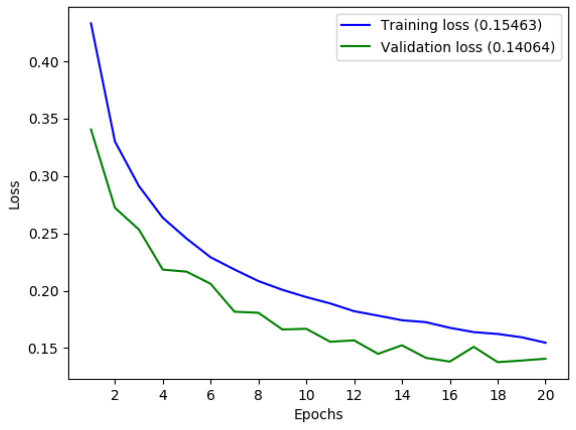
(a) Accuracy – Mel filterbank applied



(b) Loss – Mel filterbank applied



(c) Accuracy – No Mel filterbank applied



(d) Loss – No Mel filterbank applied

Fig. 5.1 Training histories of two networks over 20 epochs. The network in the upper row (subfigures (a) and (b)) was trained with a Mel filterbank (40 bins) applied. The second network, shown in the lower row, was trained without applying any filterbank settings (subfigures (c) and (d)). The difference in performance is almost negligible.

technique results in improvements which are almost at a level with the ones achieved by the *stochastic gradient descent*, an alternative method in which prediction errors are backpropagated at the end of the forward pass of every single training sample.

	Batch size 32	Batch size 128
500 / 245	0.95 / 0.94 / 0.94	0.96 / 0.95 / 0.96

Table 5.4 F_1-scores (0/1/avg) for two different batch size configurations, both with chunk size/margin settings of 500/245.

In this setup, two networks with equal settings were trained, the only difference being the (mini-)batch size, which was set to 32 during the first run and to 128 during the second. One result from the previous experiment was taken into account, namely, that larger chunk sizes yielded better results. Both networks used a chunk_size/margin configuration of 500/245, resulting in a hot-patch of 10 consecutive frames.

When comparing f_1-scores in Table 5.4, it can be seen that larger batches seem to be beneficial in this setup, albeit by a very small margin. The training duration was equally high for both networks, averaging approximately two hours per epoch. The longer epoch durations can be attributed to the larger training chunks, a finding which is consistent with observations made in the previous experiment.

5.2.3 Chunk Size and Margin

100 / 47	200 / 97	300 / 147	400 / 197
0.95 / 0.85 / 0.90	0.95 / 0.88 / 0.92	0.96 / 0.89 / 0.92	0.96 / 0.90 / 0.93

Table 5.5 F_1-scores (0/1/avg) for four different chunks size/margin configurations.

From the experiments conducted earlier, it had become evident that feeding the network with larger chunks had a positive effect on its performance. A chunk with a size of 500 frames allows the network to analyze ten seconds of spectrogram data during a single forward pass (when configured with a hop size of 441 samples at 22.05 kHz sampling rate) since $500 \cdot (441/22050) = 10$ [sec]. Under the assumption that typical song tempi range from approximately 60 to 280 BPM, this size covers between 10 and 30 beat times.

A series of four training runs was conducted to explore the space of chunk sizes between 100 (2 seconds) and 400 (8 seconds), each of them configured with a hot-patch of six frames. Table 5.5

shows how the detection of beat-positives gradually improved when chunk sizes were scaled up while the hot-patch itself was kept constant. It should be noted that in order to speed up the process, training for these runs was restricted to five epochs.

In addition to the chunk size variations, in all four runs, the batch size was set to 256 for the purpose of gaining more insight into how this last parameter influences training times and, possibly, training results.

100 / 47	200 / 97	300 / 147	400 / 197
31.9	65.8	98.1	119.9

Table 5.6 Epoch times (in minutes) for four different chunk size/margin configurations.

By inspecting Table 5.6, it can be estimated that the uniformly spaced increments of chunk sizes and their corresponding epoch durations[32] increase along a linear trend line. Furthermore, the batch size parameter did not seem to influence training times to a great extent in this particular setup: using the results presented in Table 5.3, which showed a linear behavior and were obtained with a batch size of 128 (second row, Mel bins), it is possible to approximate the outcome of a larger chunk size by linearly extrapolating (see Table 5.7). The extrapolated epoch duration for a chunk size of 400 gives approximately 137 minutes, a value slightly higher than the previously obtained value of 119.9.

	20 / 6	40 / 15	80 / 35	400 / 197	400 / 197
Batch size	128	128	128	128	256
Epoch duration (min)	5.6	12.9	26.2	$\approx$ 137.0	119.9

Table 5.7 The first three columns list the results from an earlier experiment (see Table 5.3). The value for the epoch duration in the fourth column (highlighted) is extrapolated. When comparing the latter to the experimentally obtained value in column 5, it can be seen that, in spite of changing the batch size from 128 to 256, the linear behavior between increasing chunk size and epoch duration is maintained.

As described in Section 4.2.2, the hot-patch is the central part of the training sample where any onset which falls in between its boundaries is interpreted as beat-positive. Simultaneously, the network inspects the surrounding margin regions as well, with the important difference,

[32]The concern about training times arises mainly from a practical perspective: networks using larger training chunks improved their performance but also increased training times significantly. A chunk size of 500 (around 2 hours per epoch) was, rather arbitrarily, chosen as a viable limit.

however, that potential beat events are never interpreted as such. The assumption for the following experiment was that by selecting a relatively small hot-patch, the network could now potentially be constrained to focus more narrowly on beat events (since fewer frames form part of the detection window).[33] In spite of not interpreting any events in these margin regions as beat-positives, they, nevertheless, can be expected to provide valuable information to the overall context, potentially enabling the network to analyze periodicities between recurring events along the temporal axis. Under the above-made assumptions, it is, then, logical to ask about the optimal width of the hot-patch window in order for it to capture only event-related information.

Table 5.8 compares two networks with equal settings, only distinguished by the size of the hot-patch. As can be seen by looking at the weighted average of f_1-scores, the larger hot-patch performed better by two percentage points. However, an interesting relation between support numbers and f_1-scores for each category becomes apparent: the total number of beat-positives and negatives are reflected directly by the performances of each category. This seems to suggest that if more training samples of a certain kind are present, the ability of the network to correctly identify this category during testing is also higher.

	300 /140	300 / 147
F_1-score	0.90 / 0.98 / 0.94	0.96 / 0.89 / 0.92
Support (0/1/total)	2641 / 10159 / 12800	9028 / 3772 / 12800

Table 5.8 Performance comparison between hot-patch sizes of 6 and 20 frames.

These results can be interpreted in the light of (at least) two different aspects, the first related to the individual frames themselves and the second to the width of peaks and how they are captured. With respect to the frames, it is easy to see that by widening the hot-patch window, more frames are associated with an occurring beat event. Due to the sometimes large overlaps between frames, an event is usually detected by more than one frame, forming what, in the remainder of this thesis, will be referred to as *detection groups*. When choosing a smaller frame length, e.g., 1024, these groups comprise only a few frames (between two and three in the current setup). Correspondingly, larger frame sizes (with significantly increased overlaps at a fixed hop size) form larger groups, comprising approximately 9 to 10 adjacent frames at a frame size of 4096. Since a training sample is generated at every single frame index, each additional frame added to the group results in a beat-positive sample.

[33] This also depends on the frame size parameter, as described below.

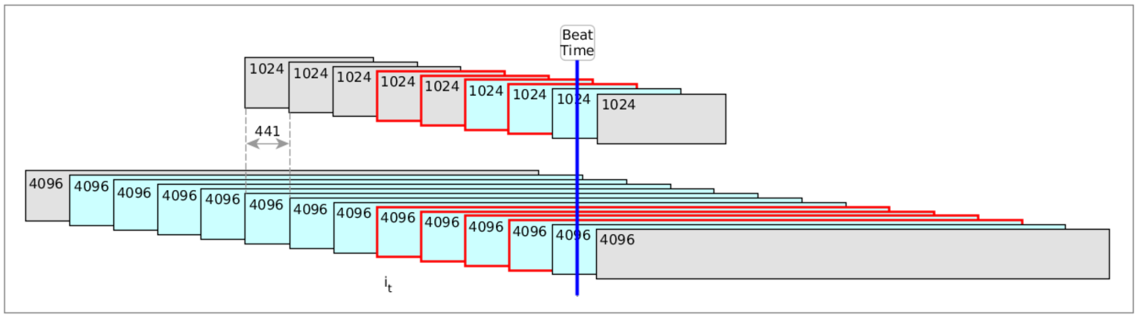


Fig. 5.2 Two networks with different frame resolutions (1024 and 4096) and equal hop size (441) at time instance t. The network with the higher frame length of 4096 captures a beat event twelve times (detection group of 12, light blue), while the one with a 1024 frame resolution captures the event in only three instances (detection group of 3). The frames marked with a red border symbolize the hot-patch of size 4 at frame index i_t (four frame indices before the last frame detecting the event). In an alternative scenario, with the hot-patch configured to be of size 8 instead of 4, the upper network would remain unaffected since only the same three frames would continue to detect the event. In the case of the network at the bottom, however, the number of event-capturing frames within the hot-patch would be extended until reaching the limit defined by the detection group.

Looking at the example in Figure 5.2, an important consequence of differing frame size settings can be observed, namely, larger frame sizes effectively "blur" the hot-patch. As a side effect, the proportion of total beat-positives to beat-negatives increases when either widening the hot-patch, increasing the frame size or both. Thus, a careful adjustment of these sizes could potentially be used as a mechanism to balance the dataset since beat-positive events are often underrepresented when compared to beat-negative events.

Another interesting effect to be considered is what can be regarded as a form of "shifting the networks' attention" to specific parts of the beat events' transient by carefully choosing certain chunk/margin settings and then combining them with the appropriate frame size parameters. Figure 5.3 shows two examples of how the networks' attention can be directed towards the attack phase of the transient or centered around the event.

The second aspect, which was mentioned above, is the average width of the peaks. It is reasonable to assume that a network will have difficulties recognizing beat events when the detection window, which, as described earlier, is defined by the combination of hot-patch and frame size configurations, is set too narrowly. To gain more control over the amount of transient information captured by a single training chunk, the width (in ms) of the hot-patch can be

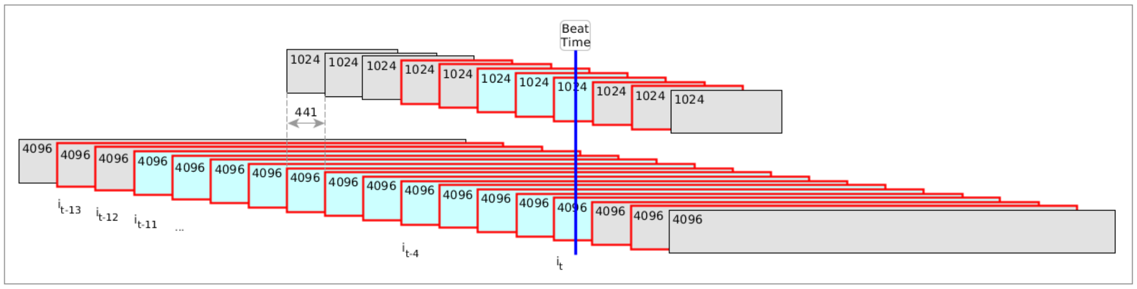


Fig. 5.3 Two networks with a hot-patch of three frames which has been shifted forward one position per step over the time span from t to $t-14$. The network on the top (with frame size 1024) has a centered view on the beat event. In this situation, with a relatively small frame size, the hot-patch size could be increased indefinitely, yet the centered focus around the beat event would always be maintained. In contrast, the network shown at the bottom (with frame size 4096) shifts the focus almost entirely towards the transients' attack due to the relatively small hot-patch and a significantly larger frame size.

calculated with the following expression:

$$W_{HP} = \frac{hs(g+2hp-2)}{f_s}[ms] \quad , \tag{5.4}$$

where g is the group size, hp is the hot-patch size (in frames), hs is the hop size in samples, and f_s is the sample rate in kHz. The hot-patches from both network configurations presented in Table 5.8 can now be computed by using Eq. 5.4 and the conventions established in Table 5.10:

	300 /140	300 / 147
Frame size	1024	1024
Hot-patch length (ms)	396	119

Table 5.9 Computing the hot-patch width for the previous experiment (see Table 5.8).

Frame size	Detection group
1024	2
2048	8
4096	12

Table 5.10 The size of detection groups in this experimental setup can vary by one frame; e.g., a network configured with a frame size of 1024 produces detection groups of two or three frames, depending on their exact position. To eliminate ambiguities, these conventions are used for any computations.

5.2.4 Difference Spectrogram

In this experiment, a single network was supplied with two different sets of spectrogram data. The first set was preprocessed by applying logarithmic compression and a Mel filterbank (like all previous spectrograms), and the second was generated by additionally taking the first-order difference. This last step strips the signal of most of the information characterizing the transient and leaves only the rising slopes indicating the onsets. As can be seen from Table 5.11, the network which was trained with difference spectrograms falls slightly behind when compared by their f_1-scores.

Difference Spectrogram	Logarithmic Filtered Spectrogram
0.91 / 0.93 / 0.92	0.93 / 0.95/ 0.94

Table 5.11 F_1-scores (0/1/avg) for two networks trained with differently preprocessed spectrograms. The *logarithmic-filtered spectrogram* was produced by applying logarithmic compression and was filtered with a Mel filterbank. The *difference spectrogram* underwent the same preprocessing steps as the first, but, additionally, the first-order difference was taken.

This outcome suggests that the network, at least in this specific configuration, relies to some extent on transient information which goes beyond the bare onset time.

5.3 Performance of Frame-Based Experiments

During the first phase of experiments, a total of 21 different network setups were tried, all of which were trained, validated, and tested with the limited frame-based dataset. Performances, measured by their avg f_1-score, were found to be relatively similar, ranging from 0.86 to 0.96 with an average value of 0.92 (0.86, in fact, was the only outlier). Comparing the f_1-scores at the category level showed that they performed almost equally well, resulting in average values of 0.94 for beat-negatives and 0.91 for beat-positives.[34]

These results did not only appear unusually high given the particular test conditions (limited dataset, wide range of different parameter settings), but they also exposed only minor differences between both categories, despite a relatively high disparity in numbers of beat-positive and negative samples. The implausibility of these results, combined with the previously made

[34]The avg f_1-score should not be confused with the average taken from the results of all runs. While the first refers to the weighted average of f_1-scores from both categories computed after a single run, the latter is simply the mean value of a specific metric obtained in all runs, e.g., accuracy, loss, or f_1-score.

observation that validation steps consistently scored higher than their corresponding training steps, gave rise to the supposition that these networks had overfitted to the training data.

In order to verify whether overfitting had taken place or not, all networks trained during the preliminary phase were evaluated a second time with the separate testing dataset. This second evaluation was done mainly with the partitioning scheme in mind: since the frame-based partitioning scheme shared very similar samples across all domains, it was reasonable to assume that in case the networks started to memorize features during training, these features could potentially be applied "successfully" to the testing dataset too, i.e., by remembering the previously analyzed "clones" from the validation or training set.

	F_1-score beat(–)	F_1-score beat(+)	F_1-score avg
Frame-based	0.94	0.91	0.92
Song-based	0.73	0.69	0.73

Table 5.12 Averaged f_1-scores from a selected representative set of training runs (no extreme outliers). The f_1-scores for any of these runs were obtained by first evaluating preliminary runs with the frame-based and later with the song-based partitioning scheme.

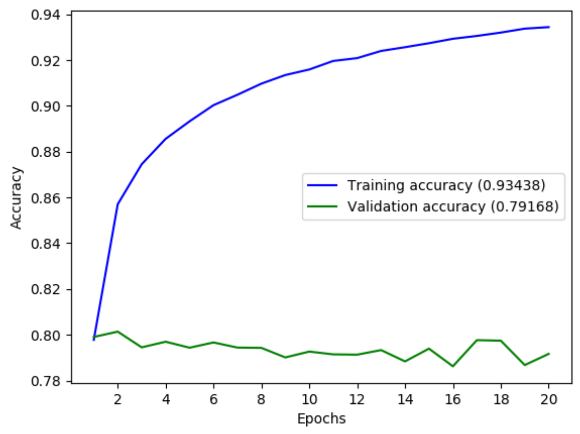
It can be seen from Table 5.12 that the avg f_1-score lost approximately 20 percent points when tested with the excerpt-based (or song-based) partitioning scheme. This significant performance loss indicated that, indeed, the networks had overfitted. For the purpose of gaining a better understanding of how these networks would perform when trained with the song-based partitions and, more importantly, being able to realistically interpret the results obtained so far, another experiment was set up.[35] These later runs retained the same network architecture and configurations employed in the previous runs, with the exception of the chunk size, which was varied, and, naturally, the use of a song-based partitioning scheme. Table 5.13 shows the results.

50 / 20	100 / 45	200 / 95
0.81 / 0.72 / 0.77	0.80 / 0.72 / 0.77	0.81 / 0.75 / 0.79

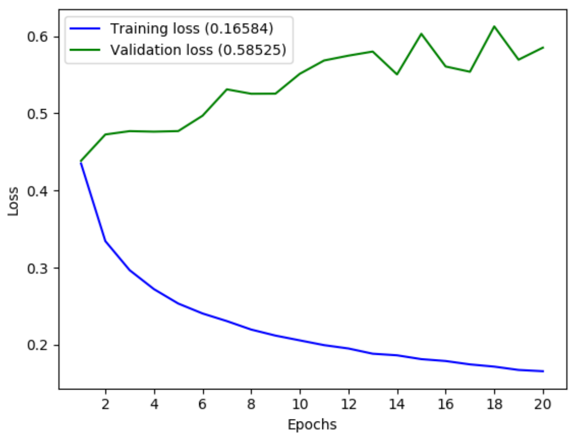
Table 5.13 F_1-scores (0/1/avg) for three overfitted networks trained and tested with the song-based partitioning scheme.

When looking at the training histories of all of these networks shown in Figure 5.4 (a)-(f), it is easy to see that performance degraded very quickly (visibly after the first epoch). Interestingly,

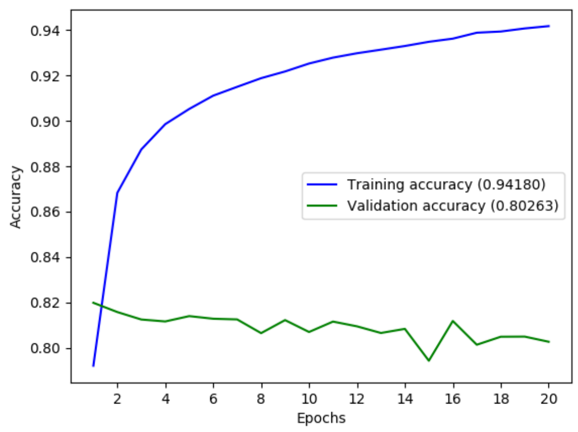
[35]Only three runs were necessary to confirm that continued training and testing of these preliminary networks would not deliver any new insights.



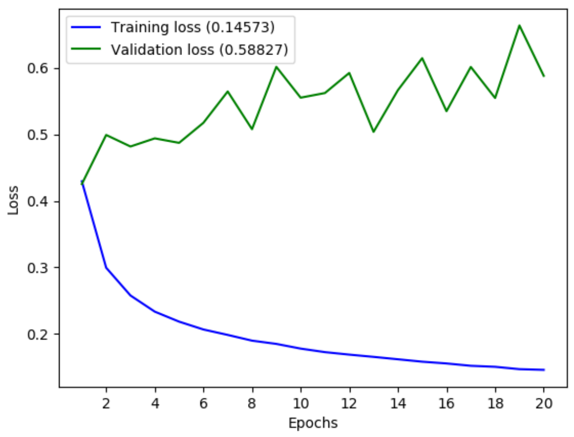
(a) Accuracy – chunk-size/margin: 50/20



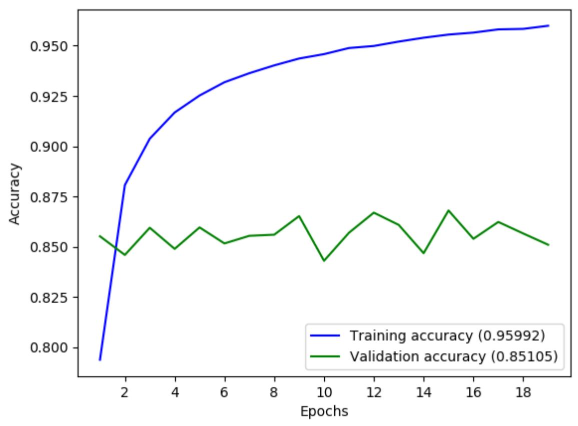
(b) Loss – chunk-size/margin: 50/20



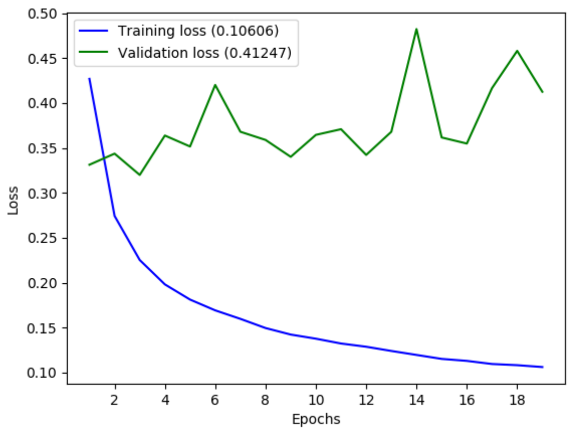
(c) Accuracy – chunk-size/margin: 100/45



(d) Loss – chunk-size/margin: 100/45



(e) Accuracy – chunk-size/margin: 200/95



(f) Loss – chunk-size/margin: 200/95

Fig. 5.4 Training histories over 20 epochs for the three overfitting networks trained and tested with song-based partitions.

also in this experiment, larger chunk sizes seemed to affect training results positively, in spite of their having virtually no impact on the overfitting behavior. One possible explanation for this outcome is that these network architectures made use of the larger chunks to memorize larger sections of spectrogram data during a single training step.

Although the results of the earlier frame-based training process imply that the resulting network models did not develop extensive capabilities to "separate the signal from the noise," they also showed that these models were able to uncover at least some features of importance, however basic. Moreover, it appears that these basic features can be learned fairly quickly. This assumption is supported by the observation that, in most cases, huge learning steps were achieved during the first epoch but during consecutive epochs, the process started to quickly degrade or stop completely.

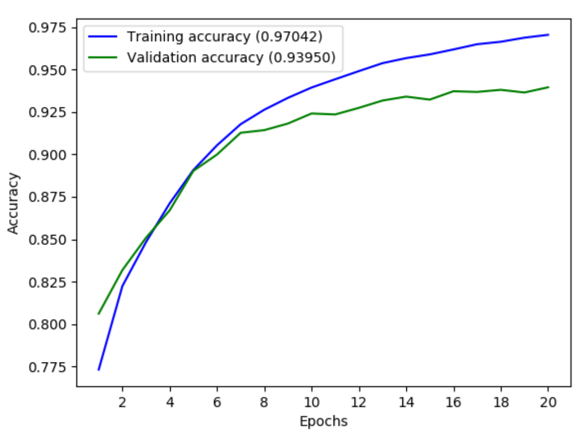
5.4 Preliminary Runs – RNN

In this experiment, two RNNs were trained over 20 epochs. Their settings were identical with the exception of chunk size and margin parameters. Figure 5.5 demonstrates that the network trained with the smaller data samples led to a relatively smooth learning curve, while the larger chunk size completely inhibited the same network from learning. The validation curves started flattening out after approximately ten epochs. It can be seen that at this point, overfitting sets in.

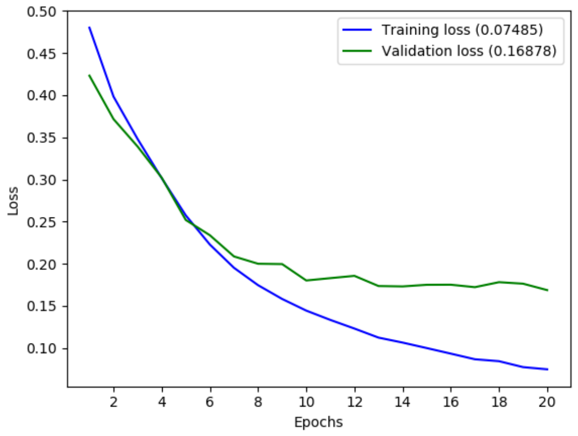
This network architecture did not exhibit a high learning capability (see Table 5.14), yet it serves as a demonstration that, not surprisingly, RNNs do not follow the same internal logic as CNNs, especially with respect to context handling. The results reflect the inner workings of an RNN since it extracts contextual information (which is, in fact, temporal information) by repeatedly inspecting smaller chunks. Conversely, CNN architectures are not designed to infer associations between series of sequential training samples and are, therefore, forced to inspect larger pieces in order to uncover any periodic patterns in a single step.

20 / 6	80 / 35
0.725 / 0 / 0.412	0.801 / 0.598 / 0.729

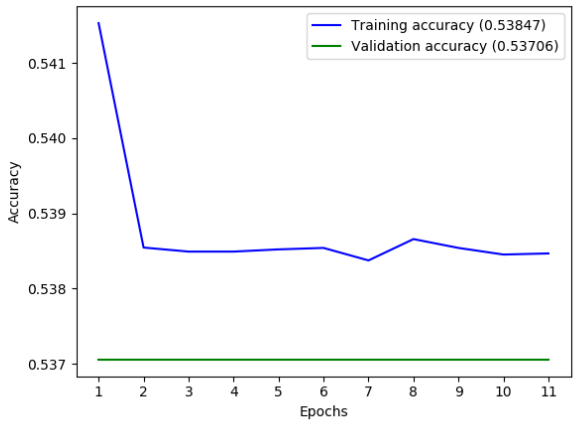
Table 5.14 F_1-score (0/1/avg) for three overfitted networks trained and tested with the song-based partitioning scheme.



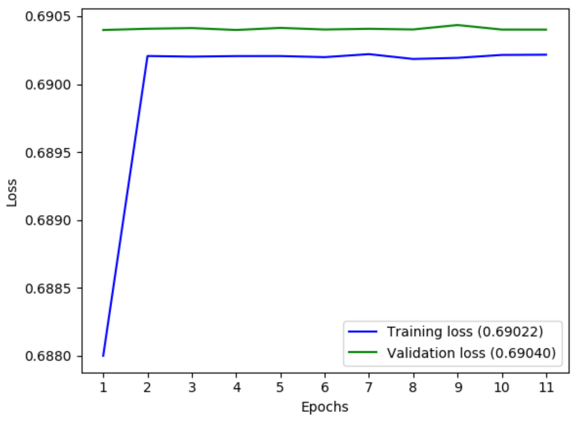
(a) Accuracy – chunk size/margin: 20/6



(b) Loss – chunk size/margin: 20/6



(c) Accuracy – chunk size/margin: 80/35



(d) Loss – chunk size/margin: 80/35

Fig. 5.5 Training histories of two RNNs over 20 epochs. The network with the larger chunk size (in the lower row) fails to initiate the learning process.

5.5 Enhanced Architecture with the Full Dataset

For the second phase of experiments, the objective had initially been to train and evaluate the networks developed in the first phase with the full dataset. Until this point, however, none of these models were able to converge without overfitting. The first experiments gave enough evidence to safely attribute the issues to the architecture itself. Overfitting can often be reduced or even eliminated by measures such as *batch normalization*, *regularization*, selecting a more appropriate *optimizer*, cutting back on *network complexity*, and *k-fold cross-validation*. Network performance for the CNN was eventually improved by applying batch normalization and substituting the optimizer. All of the following experiments were carried out using the GTZAN dataset unless otherwise stated.

5.5.1 Optimizers

An important component of the Keras framework (this is also the case for the Tensorflow framework) is the *optimizer*[36] which runs an algorithm to adaptively set the learning rate in the calculations of the stochastic gradient descent. The preliminary experiments, which were mainly aimed at determining the parameters of the basic functional networks for both architectures, used the *Adadelta* optimizer. Since the first models obtained in these first training runs seemed to perform sufficiently well, Adadelta was kept as part of the configuration for the majority of later experiments. However, after detecting that almost all models were susceptible to overfitting, a small set of networks were trained with an alternative, the *Adam* optimizer.

As described in the Keras documentation, Adadelta (similar to its predecessor *Adamgrad*) enables the network to continue learning even after a large number of updates have been made. Essentially, this is achieved by decreasing the learning rate continuously, depending on the average of past and current gradients. Using the valley analogy from Section 3.2.1 for describing the gradient descent curve, the central idea of applying an adaptive learning rate is to "slow down the ball" when the bottom of this valley is approached and the curve starts flattening out.

The Adam optimizer adds another level of sophistication to this process, namely, the concept of *momentum*. Momentum, as well, is often explained by using the above-mentioned valley analogy: the farther the ball rolls down the valley, the more momentum it gains. Since the

[36]For a short overview of optimizers, see `http://ruder.io/optimizing-gradient-descent` and `https://medium.com/datadriveninvestor/overview-of-different-optimizers-for-neural-networks-e0ed119440c3`

algorithm accelerates the learning rate over long "down-hill" stretches, the networks are often able to converge faster. The Adam optimizer is generally recommended for large datasets.

In the experimental setup of the second phase, Adadelta was replaced by Adam. As a consequence, the training process stabilized notably. However, this change was introduced together with a major redesign of the CNN network architecture. For this reason, no conclusive statement can be made about the impact of this change on the overall performance.

5.5.2 Optimizing the Network Architecture (CNN)

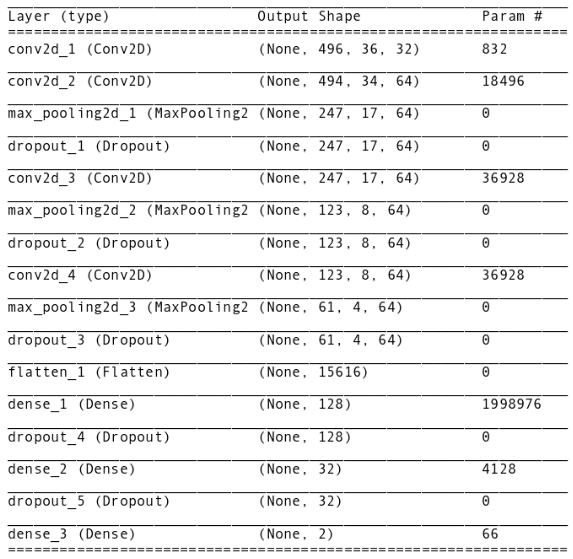
Having identified overfitting as a major issue of the network design, an attempt was made to simplify the architecture of the CNN. Lowering the number of convolutional layers from four to three did not improve training effectiveness; on the contrary, these networks decreased their learning rate significantly. Similar effects were observed when trying to decrease the filter size of one or more convolutional layers. Since no improvements were achieved by reducing complexity, it was concluded that the network capacity of the current design was not likely to be oversized.

Another change which was introduced to tackle the overfitting behavior was the implementation of *batch normalization* in all convolutional and dense layers. Batch normalization is a technique which is often recommended to improve training speed and avoid overfitting. The reason for this is that batch normalization has regularizing effects, e.g., by penalizing high weight values, and improves the generalizability of the model by reducing deterministic behavior. For more on this topic, see Ioffe and Szegedy [32].

In the Keras framework, batch normalization can be achieved by simply splitting the non-linearity, which is usually part of the convolutional or dense layer, into its own component. A new batch normalization layer is then plugged in between these two layers, the convolutional/dense layer and the non-linearity, effectively intercepting the output of the former before it reaches the latter. Figure 5.6 shows the previous architecture (from phase I) alongside the enhanced version.[37]

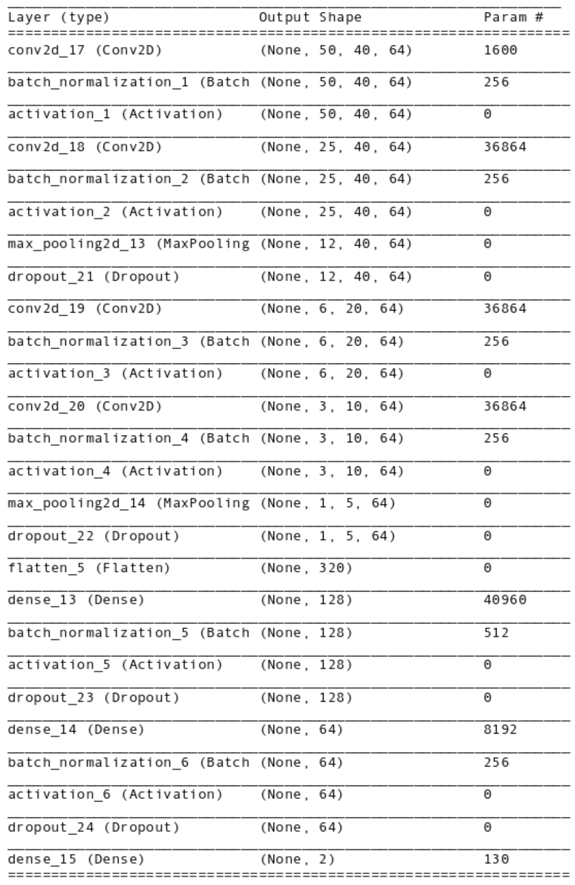
After successfully integrating batch normalization, a boost in network performance was observed. Average test loss declined from around 48% to approximately 28%, average test

[37]In order to avoid confusion between these two architectures, the improved version of the CNN architecture will be referred to as the *enhanced* architecture.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 496, 36, 32)       832
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 494, 34, 64)       18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 247, 17, 64)       0
_________________________________________________________________
dropout_1 (Dropout)          (None, 247, 17, 64)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 247, 17, 64)       36928
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 123, 8, 64)        0
_________________________________________________________________
dropout_2 (Dropout)          (None, 123, 8, 64)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 123, 8, 64)        36928
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 61, 4, 64)         0
_________________________________________________________________
dropout_3 (Dropout)          (None, 61, 4, 64)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 15616)             0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               1998976
_________________________________________________________________
dropout_4 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 32)                4128
_________________________________________________________________
dropout_5 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 66
=================================================================
```

(a) Architecture used for preliminary runs.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_17 (Conv2D)           (None, 50, 40, 64)        1600
_________________________________________________________________
batch_normalization_1 (Batch (None, 50, 40, 64)        256
_________________________________________________________________
activation_1 (Activation)    (None, 50, 40, 64)        0
_________________________________________________________________
conv2d_18 (Conv2D)           (None, 25, 40, 64)        36864
_________________________________________________________________
batch_normalization_2 (Batch (None, 25, 40, 64)        256
_________________________________________________________________
activation_2 (Activation)    (None, 25, 40, 64)        0
_________________________________________________________________
max_pooling2d_13 (MaxPooling (None, 12, 40, 64)        0
_________________________________________________________________
dropout_21 (Dropout)         (None, 12, 40, 64)        0
_________________________________________________________________
conv2d_19 (Conv2D)           (None, 6, 20, 64)         36864
_________________________________________________________________
batch_normalization_3 (Batch (None, 6, 20, 64)         256
_________________________________________________________________
activation_3 (Activation)    (None, 6, 20, 64)         0
_________________________________________________________________
conv2d_20 (Conv2D)           (None, 3, 10, 64)         36864
_________________________________________________________________
batch_normalization_4 (Batch (None, 3, 10, 64)         256
_________________________________________________________________
activation_4 (Activation)    (None, 3, 10, 64)         0
_________________________________________________________________
max_pooling2d_14 (MaxPooling (None, 1, 5, 64)          0
_________________________________________________________________
dropout_22 (Dropout)         (None, 1, 5, 64)          0
_________________________________________________________________
flatten_5 (Flatten)          (None, 320)               0
_________________________________________________________________
dense_13 (Dense)             (None, 128)               40960
_________________________________________________________________
batch_normalization_5 (Batch (None, 128)               512
_________________________________________________________________
activation_5 (Activation)    (None, 128)               0
_________________________________________________________________
dropout_23 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_14 (Dense)             (None, 64)                8192
_________________________________________________________________
batch_normalization_6 (Batch (None, 64)                256
_________________________________________________________________
activation_6 (Activation)    (None, 64)                0
_________________________________________________________________
dropout_24 (Dropout)         (None, 64)                0
_________________________________________________________________
dense_15 (Dense)             (None, 2)                 130
=================================================================
```

(b) Architecture used for full test set.

Fig. 5.6 Main architectures used in the experiments. Output taken from the Keras *print_summary* module.

accuracy increased by 7%, and avg f_1-score increased by 7% as well.[38] Overfitting during or directly after the first epoch generally ceased to occur and was, in most cases, delayed or eliminated altogether.

5.5.3 Enhanced Architecture (CNN) with Excerpt-Based Data Partitions

After implementing the above-mentioned changes, the state of the current network architecture was considered sufficiently mature for the second phase of the experiments to be initiated. From the first phase, it had become evident that the excerpt-based data partitioning was far more applicable to a real-world scenario. For this reason, some of the already analyzed parameters were subjected to new evaluations under the changed conditions. The primary goal was to corroborate (or refute) the earlier findings about the influence of these parameters on network performance. Secondly, it was expected that more insight into the viability (or non-viability) of frame-based data partitioning versus excerpt-based partitioning could be gained by comparing the earlier findings with the newer results.[39]

Initially, network training was resumed by making further use of the limited dataset to accelerate training times. In spite of having notably improved the architecture in the previous phase, overfitting still occurred occasionally in some configurations. As a consequence, subsequent experiments were exclusively carried out using the full dataset. The disposal of the limited dataset for future runs was found to be an effective means to further eliminate overfitting. This can be attributed to the fact that a greater data volume inhibits the networks' ability to memorize patterns as long as the network capacity is kept constant.

5.5.4 Hot-Patch

Table 5.15 compares the evaluation results of three networks distinguished by the size of the hot-patch. As in previous experiments, all other parameters were fixed. Again, these results are particularly interesting in the light of the support numbers since it can be confirmed that f_1-scores are correlated to the number of samples the networks inspect in each category. By implication, this outcome seems to suggest that the setting of the hot-patch width is more

[38]These values should be considered as rough estimates only since the exact numbers depend on whether extreme outliers are included or not. Furthermore, the employment of the enhanced version is not strictly tied to a particular instance of a trained network.

[39]It should be noted, however, that a direct comparison is not possible due the fact that results from the overfitted networks (and therefore partially degraded models) cannot be regarded as completely valid equivalents to the ones obtained in the second phase.

effective in terms of permitting the processing of more domain-specific information, e.g., more beat-positive chunks, than it is by letting the network extend its view of the actual data, particularly the structure and composition of an onset.

	300 / 140	300 / 147	300 / 149
F_1-scores	0.678 / 0.918 / 0.860	0.909 / 0.733 / 0.860	0.958 / 0.641 / 0.918
Support	14390 / 45546 / 59936	43232 / 16704 / 59936	52346 / 7590 / 59936

Table 5.15 F_1-scores (0/1/avg) and support (0/1/total) obtained by running a network under three different hot-patch configurations (enhanced architecture).

In spite of a low detection rate in the category of beat-positives, the rightmost hot-patch (300/149) in Table 5.15, which is also the narrowest one, scored highest when measured by the weighted average. Since the training history showed very stable loss and accuracy curves as well, this network model was selected as a *baseline* for subsequent experiments. Table 5.16 resumes the part of the configuration which is relevant for this discussion.

Chunk size	100
Margin	49
Hot-patch	2
Frame size	1024
Sample rate	22050
Hop size	441
Batch size	32
Spectrogram	logarithmic-filtered
Optimizer	Adam
Training set	Full
Test set	Full

Table 5.16 Configuration of the baseline model.

5.5.5 Chunk Size

The chunk size was tested with three different values, 100, 300 and 500, which, expressed in time units, correspond to 2, 6 and 10 seconds (at a sample rate 22.05 kHz). The earlier finding from the chunk size experiment from phase 1 was confirmed: larger chunks (with a fixed hot-patch) improve results by a small margin. Extending the chunk does not affect support numbers; therefore, the improved results observed when using larger chunks can genuinely be attributed to augmented context.

100 / 49	300 / 149	500 / 249
0.955 / 0.643 / 0.916	0.958 / 0.641 / 0.918	0.958 / 0.659 / 0.920

Table 5.17 F_1-scores (0/1/avg) with three different chunk sizes.

5.5.6 Frame Size

Table 5.18 shows the results for frame sizes of 1024, 2084, and 4096. Incrementing the frame size, as described above, includes more beat-positive samples. The outcome of this experiment is in accordance with previously made observations: the more the frame size is incremented, the more beat-positive chunks are analyzed, and, consequently, detection rates for this category rise. The drawback is, of course, that f_1-scores for beat-negatives fall and, consequently, the weighted average moves toward this more strongly represented category.

	1024	2048	4096
F_1-scores	0.955 / 0.643 / 0.916	0.929 / 0.706 / 0.881	0.863 / 0.782 / 0.831
Support	61085 / 8867 / 69952	54912 / 15040 / 69952	42505 / 27447 / 69952

Table 5.18 F_1-scores (0/1/avg) with two different frame size parameters.

5.5.7 Batch Size

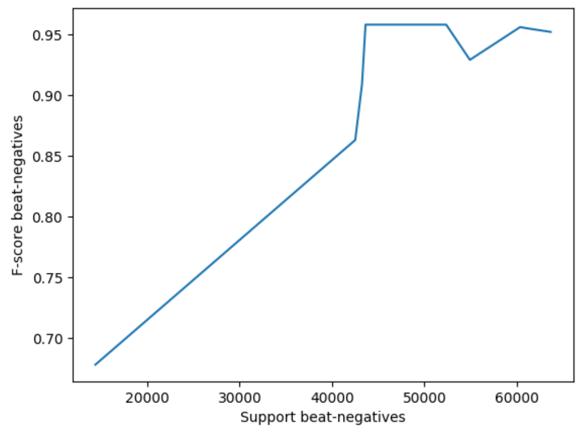
After running a network configuration with mini-batches of 256 against the baseline model, in this case too, the outcome from the corresponding experiment from phase 1 was confirmed: the batch size had almost no impact on the final result.

32	256
0.955 / 0.643 / 0.916	0.956 / 0.623 / 0.914

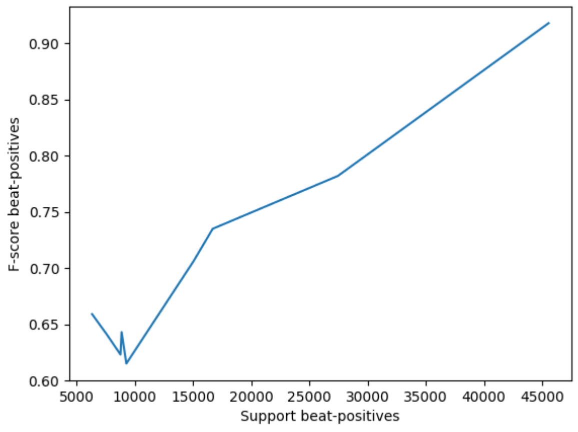
Table 5.19 F_1-scores (0/1/avg) with two different batch size parameters.

5.5.8 Support Numbers

Figure 5.7 shows two graphs which demonstrate the high degree of correlation between support numbers and the corresponding f_1-scores for each category. Although the performance of these



(a) Support/f_1-score beat-negatives



(b) Support/f_1-score beat-positives

Fig. 5.7 F_1-scores are highly correlated to support numbers in each category.

networks depended on other factors as well, it can be said with a high degree of certainty that this relation is a dominant factor.

The nine networks selected for this comparison share identical base settings, e.g., training with the full dataset GTZAN, batch size, number of bins, hop size, type of spectrogram, optimizer, and training epochs. The varying parameters were altered one at a time with respect to the baseline network to make them comparable, as was the case in all experimental setups in phase II.

5.5.9 Cross-Validations of GTZAN and SMC MIREX

In order to evaluate the (up to this point) best-performing network under the conditions of the significantly more challenging SMC MIREX dataset, two new experiments were conducted. First, the network was trained with the GTZAN dataset and then evaluated with the SMC MIREX test set. In the second experiment, the setup was inverted: SMC MIREX was used for training, and the evaluation was done with GTZAN. Table 5.20 and Table 5.21 show the results. The first experiment revealed that the network was able to predict beat-negatives almost perfectly but struggled to recognize beat-positives, especially when dealing with the SMC MIREX data. The beat-negative bias can, as in previous experiments, be explained by the fact that it was trained mainly on samples of this category. The performance drop in the SMC MIREX dataset had its origin in the structure of the data, chiefly the nature of onsets but also other factors, e.g., expressive timing or poor sound quality. This was an expected outcome.

GTZAN	SMC MIREX
0.958 / 0.659 / 0.920	0.950 / 0.331 / 0.889

Table 5.20 F_1-scores (0/1/avg) for best-performing network trained with GTZAN and cross-validated with SMC MIREX.

SMC MIREX	GTZAN
0.948 / 0.087 / 0.863	0.931 / 0.028 / 0.817

Table 5.21 F_1-scores (0/1/avg) for best-performing network trained with SMC MIREX and cross-validated with GTZAN.

More surprising were the results of the second experiment. The network which, under the conditions of GTZAN, trained reasonably well failed completely when trained with SMC MIREX. Accordingly, the cross-validation with GTZAN returned poor results too. This outcome suggests that the network architecture was overly specialized for recognizing music with hard onsets and a relatively steady tempo. When training the network on GTZAN, it was able to find a sufficient number of these easier features on which it performed well while failing to correctly detect most of the more difficult features, soft and missing onsets. The SMC MIREX dataset, it appears, did not provide enough easily detectable onset information, and, as a consequence, the network failed to recognize most these events.

CHAPTER 6

CONCLUSIONS

6.1 General Considerations

This chapter walks briefly through the main findings gathered in the experimental part of this thesis while striving to extract the essential outcomes and putting them into a broader context. Most of the aspects discussed in the following sections were specifically targeted by this thesis; others arose from the constraints and impediments encountered during execution. These conclusions include considerations with respect to dataset limitations and partitioning, network complexity, architectural improvements, measures designed to counter overfitting, observations related to context embedding and training sample size, and the effect of inherently differing datasets on network performance.

6.1.1 Dataset Limitations

Neural networks require big quantities of data, but not only data volume is important; so is quality. Networks need correctly labeled data samples, which are often difficult to obtain. This is especially true for beat tracking since producing beat annotations manually is a time-consuming and difficult task. While annotating beats is already problematic because of the ambiguities that can arise as a consequence of, e.g., different pulse levels, the creation of onset-annotated datasets is even more challenging. In most cases, it might still be feasible to correctly detect and record hard onsets, but it is undoubtedly more difficult to do for soft onsets.

The traditional approach of first detecting onsets and then, in a second step, using these onsets to infer the beat is error-prone due to the fact that neither type of event[40] necessarily resembles the other. Datasets often annotate only beat times (not onsets) at different pulse levels. A learning system, however, needs to distinguish beat events from many other events which are almost identical in appearance but do not share this somehow "mystical connection" in which beats participate. On the other hand, onsets still give the most valuable clues, simply because they tend to be clearly distinguishable from the rest of the signal. Working with onset or beat tracking datasets is difficult for these reasons; even if a system has access to high-quality annotated data, these annotations are inherently limited in providing deterministic labels for the beat events.

6.1.2 Data Partitioning

Training samples were generated from every frame in the framed audio signal; therefore, neighboring samples were highly correlated. At a sample rate of 22.05 kHz and a frame size of between 1024 and 4096, these samples were spaced at distances of between 46 and 185 milliseconds, thus sharing much of the same information. The setting up of a frame-based data partitioning scheme, where frames from all audio excerpts were mixed together before the data was split up into training, validation, and test subsets, was intended to evenly distribute the whole range of features that are present in the data over these different domains. It was assumed that this approach would help to prevent data domains from developing biases towards specific musical styles. Furthermore, it was expected that these data partitions would not be disproportionately dominated by outliers. Generally, when a training data distribution is reflected closely by the test distribution, performance suffers less from a problem known as dataset shift. This latter phenomenon is characterized by a test set that incrementally ceases to represent training data.

During the first phase of experiments, a series of trial networks was trained and evaluated by means of the frame-based dataset. It was demonstrated that all these networks started to overfit. This was due to the fact that all data partitions contained neighboring samples (slightly time-shifted copies) which were similar to such a degree that it permitted the networks to memorize them to a great extent. Although overfitting and the memorization which ensued degraded these models considerably, some learning took place nevertheless: many of the results produced by these overfitted models were corroborated in phase II, where an alternative partitioning scheme

[40]It should be remembered that an onset merely represents a point in time. The physical entity in the audio signal, which can be detected, analyzed, and compared in reality, is what is called the transient.

had been employed. Results obtained in both phases were generally in accordance with each other, e.g., the finding that a larger chunk size was beneficial for the learning process.

In the second phase of experiments, the excerpt-based data partitioning scheme had the effect of correcting the earlier results downwards. Restricting neighboring samples to not being spread across different data domains produced both more consistent and realistic results. Evaluations of these later models showed that test data did not digress much from validation data. In some cases, even these models started to overfit; this, however, was due to suboptimal parameter settings and cannot be attributed entirely to the partitioning scheme. It is beyond doubt that the excerpt-based approach can be considered as more effective and is certainly more applicable when developing real-world applications.

6.1.3 Network Complexity

Leveraging network complexity was one of the major factors in the optimization process carried out during the whole project, heavily influencing performance and training times. It was also found to be a crucial measure in order to eliminate overfitting behavior. Networks designed to be too powerful started to memorize in all analyzed cases; a better approach was,therefore, to build up the architecture from the ground up, starting with a very simple network and gradually adding complexity (in the form of layers or nodes) to it.

Batch normalization was found to be essential, involving all networks explored in the second phase. In most runs, this extra layer was applied consistently after the convolutional and dense layers where it was positioned between the layer output and the activation function. Subsequent layers did seem to benefit from receiving a previously normalized input since they did not have to respond to extremely high[41] weight values and shifts in the data distribution. As a result, the network converged faster and overfitting was notably reduced.

As a side note, overfitting was also further diminished by increasing the data volume. Some network configurations started overfitting after the first epoch when trained with the small dataset but stabilized when being run on the full dataset.

[41]Batch normalization helped in regularizing mainly high, but not low, values due to the use of the ReLU non-linearity which cuts off negative values.

6.1.4 Embedding of Context Information

One special topic of interest that this thesis set out to explore was how context could be encoded in the training data. The smallest unit that is able to detect a beat time is called a *frame*. In order to collect sufficient data to detect a beat event, these frames were bundled into groups. The creation of groups was also influenced by the hop size parameter, a setting that was chosen to ensure that frames partly superimposed each other. As a consequence, beat events were always encompassed by several of these basic units. The training chunk was introduced with the intention to organize frames so that they would act as larger entities. A mask was applied to each training sample so that the adjustable margins could be used to provide context and the center area, the hot-patch, would function as an adjustable detection window (thus allowing for a tolerance). This mechanism was expected to be useful particularly for CNNs, less so for RNNs. The reason for this assumption was that CNNs are not designed to analyze temporal context but rather, the latter has to be encoded in the individual training samples. The RNN-type architecture, in contrast, supports this natively.

In all experiments conducted with CNNs where smaller training chunks were directly compared to larger ones, the results indicated that the performance of these networks benefited from their being provided with larger chunks of data. This is true for both preliminary test models and the enhanced architectures tested in phase II. The downside of training with larger chunk sizes was mainly one of practical concern: training times increased substantially. The *baseline* network which served as a reference configuration to facilitate direct comparisons at parameter level was selected for the reason of practicality: it was not the highest performing model; yet, with its relatively narrow temporal input dimension (100 frames), it was 4.3 times faster than a slightly superior model that was operating on a considerably larger scale (500 frames).

RNNs behaved exactly contrary to CNNs when comparing chunk size parameters: larger chunk sizes decreased performance considerably. This is not surprising since this type of architecture establishes temporal, i.e., context, information through the process of recurrent analysis of the same input data. Hence, it is easy to imagine that a large training sample comprises significant redundancies.

6.1.5 Architectural Considerations

Another practical issue arose from trying to feed differently sized input data arrays into the same network with unchanged layer configurations in a sequence of programmed training runs.

Ideally, the stacked convolutional layers transform the input array step by step into an elongated output volume through the application of filters during convolutions, followed by MaxPooling operations. These filters were required to be dimensioned appropriately in order to downsize the output volume smoothly and symmetrically[42] while traversing these CNN layers. Whenever a network received a set of training samples with greatly differing chunk sizes in consecutive runs, some of these output volumes eventually became too strongly compacted. In some cases, the program even crashed because one or more dimensions were being pushed off the scale into a negative range of numbers. As a consequence, direct performance comparisons of networks trained with hugely divergent chunk sizes were often not possible due to the fact that these networks were adapted to one target size only. The alternative was to adjust the kernel size and MaxPooling parameters specifically to fit every single target size. Modifying the network capacity, however, would not only have resulted in an extensive configuration overhead but, more importantly, would have made it impossible to safely associate performance differences with the parameter of interest, namely, the chunk size parameter.

6.1.6 Support Numbers and Run-Time Dataset Distribution

Closely related to the question of how context can be captured by leveraging training parameters is the size of the hot-patch or, phrased differently, how much margin area will be cut off the detection window. The reason for introducing the hot-patch window was two-fold: firstly, it was expected to function as a tolerance window since beat intervals in dynamic performances almost never adhere to the exact positions computed by an algorithm. Secondly, the hot-patch could potentially be adjusted to capture only relevant (onset) information, i.e., the nature of the peak, e.g., frequency distribution, or other distinctive properties inherent in the event.

To interpret the results obtained in the hot-patch experiments, it is important to understand that the number of frames associated with both categories, which varied under different hot-patch configurations, directly impacted performance. By widening the detection window, the support numbers were shifted towards beat-positives while reducing beat-negatives and vice versa. Consequently, a key observation from these experiments was that support numbers of each category were correlated to their f_1-scores; i.e., the higher the number of beat-positive samples which a network was able to analyze during a run, the more it improved its predictions with respect to this specific category. In most cases, this performance shift between categories did

[42]Smooth transitioning of the output volume is not a strict requirement, but it is certainly a useful design decision in most cases.

not change or improve overall performance measured by the weighted average f_1-score: gains in one category were usually compensated by losses in the other. A corollary of this observation is that any effort to balance the distribution of both categories can potentially impact network performance positively. In one of the initial experiments during phase I, an attempt was made to balance out beat-positive with beat-negative samples. Although the experiment showed greatly improved results, this approach was quickly discarded because it was (perhaps too quickly) assumed that artificially balancing the natural distribution would not reflect reality.

6.1.7 Hot-Patch Size and Detection Groups

Another interesting aspect of the different hot-patch settings was the "blur" effect that occurred with larger hot-patches, especially in combination with higher frame sizes. It was found that frame sizes of 4096 samples at a hop size of 441 and sample rate of 22050 kHz (and similarly large frame sizes) were blurring the detection window too heavily because these settings resulted in detection groups of more than 12 frames. The portions of the signal which were then labeled beat-positive often started to exceed the beat-negatives, a distribution which, with most of the audio material, was no longer realistic. Blurring the detection window, whether accomplished by large frame sizes, large hot-patches or both, generally resulted in adverse effects on the learning process. Consequently, the initial assumption that directing the networks' attention towards the attack phase of transients could be used as a control mechanism was found to be not viable since no data supported this conjecture. The best results were obtained by confining the detection window to the minimum width, suggesting that an explicitly integrated tolerance window does not necessarily promote performance. Rather contrarily, the investigated networks achieved better scores when able to focus only on a minimal set of frames inside the hot-patch window and infer timing variations from the context.

The fact that detection groups do not return single, unified predictions for an event but rather accumulate the individual responses of frames that comprise them has another important implication: during a later peak detection stage, the beat tracker has to include a task that merges these micro-predictions into a single prediction. While peak-picking, which deals with the interpretation and selection of these frame-based predictions, is not the subject of this thesis, it should be noted that the system, developed for the experimental part, could easily be adapted to return frame-based predictions as probabilities instead of categories. This change could potentially facilitate the task of peak-picking.

6.1.8 Cross-Validation with Different Datasets

Network performance was hugely impacted by the use of a special dataset, called SMC MIREX, which was designed with the specific intent to be challenging for automated beat tracking systems. Testing with the SMC MIREX dataset did, indeed, show that the developed architectures were primarily adapted to the GTZAN dataset. One conclusion that can be drawn from the performance drop under the SMC MIREX dataset is that the networks had specialized too much on features which were easier to detect, particularly hard onsets with steady timing, and were still not mature enough to fill in the gaps where such information was scarce.

6.1.9 Limitations and Achievements

Audio preprocessing played a fundamental role in the entire training process. However, in order to stay within a reasonable time frame in the experimental part of this thesis, the set of tunable audio parameters was limited to include only a few, specifically, the frame size, the filterbank setting, and the decision on whether a first-order difference was applied to the spectrogram or not. All other audio settings were fixed. Focusing on tuning the networks mainly during the training sample generation and architecture development stages was considered as the best (if not the necessary) starting point, but there is no doubt that audio settings are just as important. For instance, the hop size, which defines the distance between frames, directly influences the degree of frame overlap and is tightly interconnected with the frame width. Together, these parameters define what can be seen as the spectrogram resolution on the time axis. Leveraging the resolution and other audio preprocessing-related experiments would surely have been insightful. Furthermore, other very promising techniques that have successfully been employed in other research projects, such as the difference spectrogram, were not fully explored.

In total, close to a hundred runs were carried out. During the course of all experiments, test loss fell below 20% only three times. In the majority of runs, most notably in phase II, networks seemed to hit a learning barrier at a weighted average f_1-score of approximately 87, with beat-positives scoring significantly lower (averaging an f_1-score of approximately 69 by taking the most representative runs). This indicates that there is still plenty of room for improvement.

Nevertheless, it can be concluded that in spite of the difficulties encountered related to overfitting and the learning barrier, which prevented networks from discovering some of the less obvious beat tracking indicators, important progress has been made in uncovering some of the fundamental principles and limitations of BT. The best model achieved an average weighted f_1-score of 92% after four epochs on the GTZAN dataset and was still improving. The next steps inspired by this research effort, which could potentially help to overcome the limitations faced by the current architectures, will be explored in Section 6.2.

6.2 Future Work

One of the main obstacles to network performance was overfitting, as described earlier. Overfitting was, in part, strongly related to the data volume but also to how data samples were divided and organized. The step from a frame-based to an excerpt-based partitioning scheme brought important improvements, and comparing them directly helped to explain the nature of the underlying data. However, better results can be expected from employing a more complex data splitting technique called *k-fold cross-validation* (KFCV). KFCV is known[43] to be a very effective overfitting countermeasure. The data is divided into *k* equal groups (folds) where each of these groups is then used to test a model which was previously trained on the remaining groups. One advantage of this method is that, statistically, the two categories are evenly distributed over all data domains (no data shift). One drawback of implementing KFCV in the current system would be that of added complexity due to the handling of multiple folds and models internally.

Data augmentation is another powerful technique potentially useful for boosting network performance and operates purely at the data level. It augments the total number of training samples by creating several slightly modified copies, which are then added to the batches a data generator dispatches. Data augmentation can be used to *balance the set of training samples*, an approach which was tried earlier in this thesis but abandoned prematurely. As a consequence of data augmentation, the network would train on equal numbers of beat-negatives and positives, helping to achieve better scores on the latter.

At the architectural level, *network layer configurations* can be refined substantially. One example is the use of *regularizers*, which penalize extreme values taken on by network layer parameters, such as weights or biases. It should be remembered that batch normalization has also the side effect of regularizing outputs between layers. However, the use of dedicated regularizers, built in at layer level, provides more configuration options, e.g., the type of regularization,

[43]"Known," rather vaguely, refers to the general knowledge held by the ML community, expressed in forums and other web-based sources.

such as L_1 and L_2. Another architectural example is the handling of dropout layers. Batch normalization can, in some cases, have the effect of making dropout layers redundant. The reason for this is that the data samples are treated as batches and their individual outputs are averaged. As a consequence, the network no longer constantly produces the same output for every single sample during a forward pass, changing this behavior to favoring non-deterministic layer outputs. This can help the network to generalize features, thus precisely matching the purpose of dropout layers.

Apart from improving and extending some of the already explored and implemented tools and techniques, entirely new approaches should be explored too. Starting with a more extensive round of experiments centered around the already discussed RNNs, other more sophisticated architectures could be constructed, for instance by combining architectures. The idea of coupling a CNN with an RNN, where the CNN functions as a frontend for the RNN, is particularly interesting for the beat tracking problem since it involves image recognition as well as the handling of time dependencies. Another promising approach is the employment of a separate neural network specifically designed for the task of peak-picking, essentially postprocessing the frame-wise predictions in order to select the single most likely instant of a beat time among all candidates in a group of beat-positives.

6.3 Contributions

For this thesis, a modular, flexible, and extensible framework was developed that facilitates the automated building and running of different network architectures, provides unsupervised evaluation with preconfigured datasets, compiles the results, and organizes them in an automatically maintained folder structure. The framework includes audio preprocessing capabilities, handles efficient storage of spectrogram slices, and bootstraps itself completely from JSON configuration files. The program comes with a set of useful helper functions. Some of them are directly used by the main module; others are code snippets that can conveniently be imported from iPython/Jupyter in order to get immediate access to several more complex objects for testing purposes without the need to build them from scratch in a multi-step procedure. Examples of these objects are file objects and data generators. Another feature is the flexible handling of configuration options in the form of a modifiable, splittable configuration object which is automatically parsed from disk at startup and provides dot-separated access to all option branches and properties. The framework is programmed in Python and makes use of the Keras framework for neural networks. The code is still under active development and can be expected to be released as fully tested and documented software under the GPLv3[44] in the course of 2019. The configuration object, which is now named *dot-configs* and is installable via pip, was released earlier in 2019 as a separate software under the GPLv3.

[44]GPLv3 refers to the third version of the widely-used GNU General Public License, a Free Software license that guarantees end-users the right to use, share, and modify the source code under the restriction that derivative work is published under the same terms. More information about the GPLv3 can be found on the license website of the Free Software Foundation `https://www.gnu.org/licenses/gpl-3.0.en.html`.

REFERENCES

[1] Abdallah, S. and Plumbley, M. (2003). Unsupervised onset detection: a probabilistic approach using ica and a hidden markov classifier. In *Cambridge Music Processing Colloquium*.

[2] Aggarwal, C. C. (2018). *Neural Networks and Deep Learning - A Textbook*. Springer.

[3] Bello, J. P., Daudet, L., Abdallah, S., Duxbury, C., Davies, M., and Sandler, M. B. (2005). A tutorial on onset detection in music signals. *IEEE Transactions on speech and audio processing*, 13(5):1035–1047.

[4] Bello, J. P., Duxbury, C., Davies, M., and Sandler, M. (2004). On the use of phase and energy for musical onset detection in the complex domain. *IEEE Signal Processing Letters*, 11(6):553–556.

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

[6] Böck, S., Krebs, F., and Widmer, G. (2015). Accurate tempo estimation based on recurrent neural networks and resonating comb filters. In *ISMIR*, pages 625–631.

[7] Böck, S. and Schedl, M. (2011). Enhanced beat tracking with context-aware neural networks. In *Proc. Int. Conf. Digital Audio Effects*, pages 135–139.

[8] Cemgil, A. T. and Kappen, B. (2003). Monte carlo methods for tempo tracking and rhythm quantization. *Journal of Artificial Intelligence Research*, 18:45–81.

[9] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

[10] Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2015). Gated feedback recurrent neural networks. In *International Conference on Machine Learning*, pages 2067–2075.

[11] da Silva, I. N., Spatti, D. H., Flauzino, R. A., Liboni, L. H. B., and dos Reis Alves, S. F. (2017). *Artificial Neural Networks - A Practical Course*. Springer.

[12] Davies, M. E., Degara, N., and Plumbley, M. D. (2009). Evaluation methods for musical audio beat tracking algorithms. *Queen Mary University of London, Centre for Digital Music, Tech. Rep. C4DM-TR-09-06*.

[13] Degara, N., Rúa, E. A., Pena, A., Torres-Guijarro, S., Davies, M. E., and Plumbley, M. D. (2012). Reliability-informed beat tracking of musical signals. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(1):290–301.

[14] Dixon, S. (2001). Automatic extraction of tempo and beat from expressive performances. *Journal of New Music Research*, 30(1):39–58.

[15] Dixon, S. (2006). Mirex 2006 audio beat tracking evaluation: Beatroot. *MIREX 2006*, page 27.

[16] Du, K.-L. and Swamy, M. (2014). *The differences between Artificial and Biological Neural Networks*. Springer.

[17] Duxbury, C., Bello, J. P., Davies, M., Sandler, M., et al. (2003). Complex domain onset detection for musical signals. In *Proc. Digital Audio Effects Workshop (DAFx)*, volume 1, pages 6–9. Queen Mary University London.

[18] Duxbury, C., Sandler, M., and Davies, M. (2002). A hybrid approach to musical note onset detection. In *Proc. Digital Audio Effects Conf.(DAFX,'02)*, pages 33–38.

[19] Elowsson, A. (2016). Beat tracking with a cepstroid invariant neural network. In *17th International Society for Music Information Retrieval Conference (ISMIR 2016); New York City, USA, 7-11 August, 2016.*, pages 351–357. International Society for Music Information Retrieval.

[20] Gers, F. A. and Schmidhuber, J. (2000). Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 3, pages 189–194. IEEE.

[21] Gers, F. A., Schmidhuber, J., and Cummins, F. (1999). Learning to forget: Continual prediction with lstm. *Neural Computation*.

[22] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

[23] Goyal, P., Pandey, S., and Jain, K. (2018). *Deep Learning for Natural Language Processing*. Springer.

[24] Graves, A. and Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5-6):602–610.

[25] Grosche, P. and Müller, M. (2009). Computing predominant local periodicity information in music recordings. In *2009 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, pages 33–36. IEEE.

[26] Hainsworth, S. W. and Macleod, M. D. (2004). Particle filtering applied to musical tempo tracking. *EURASIP Journal on Advances in Signal Processing*, 2004(15):927847.

[27] Haykin, S. O. (2009). *Neural Networks and Learning Machines*. Pearson.

[28] Heaton, J. (2015). *Artificial Intelligence for Humans, Volume 3: Deep Learning and Neural Networks*. Heaton Research, INC.

[29] Hochreiter, S. (1998). The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116.

[30] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[31] Holzapfel, A., Davies, M. E., Zapata, J. R., Oliveira, J. L., and Gouyon, F. (2012). Selective sampling for beat tracking evaluation. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(9):2539–2548.

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

[33] Karpathy, A. (2018). Cs231n: Convolutional neural networks for visual recognition. `http://cs231n.github.io/`. [Online; accessed 10-Jan-2019].

[34] Keller, J. M., Liu, D., and Fogel, D. B. (2016). *Fundamentals of Computational Intelligence - Neural Networks, Fuzzy Systems, and Evolutionary Computation*. IEEE Press, Wiley.

[35] Khan, S., Rahmani, H., Shah, S. A. A., and Bennamoun, M. (2018). *A Guide to Convolutional Neural Networks for Computer Vision*. Morgan & Claypool Publishers.

[36] Klapuri, A. (1999). Sound onset detection by applying psychoacoustic knowledge. In *1999 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings. ICASSP99 (Cat. No. 99CH36258)*, volume 6, pages 3089–3092. IEEE.

[37] Klapuri, A. P., Eronen, A. J., and Astola, J. T. (2006). Analysis of the meter of acoustic musical signals. *IEEE Transactions on Audio, Speech, and Language Processing*, 14(1):342–355.

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

[39] Lanham, M. (2018). *Learn ARCore - Fundamentals of Google ARCore*. Packt Publishing.

[40] Lipton, Z. C., Berkowitz, J., and Elkan, C. (2015). A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*.

[41] Lyon, D. A. (2009). The discrete fourier transform, part 4: spectral leakage. *Journal of object technology*, 8(7).

[42] Michelucci, U. (2018). *Applied Deep Learning: A Case-Based Approach to Understanding Deep Neural Networks*. Apress.

[43] Montavon, G., Orr, G., and Müller, K.-R. (2012). *Neural Networks: Tricks of the Trade (Lecture Notes in Computer Science)*. Springer.

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

[45] Nielsen, M. A. (2015). Neural networks and deep learning. `http://neuralnetworksanddeeplearning.com/index.html`. [Online; accessed 10-Jan-2019].

[46] Peeters, G. and Papadopoulos, H. (2011). Simultaneous beat and downbeat-tracking using a probabilistic framework: Theory and large-scale evaluation. *IEEE Transactions on Audio, Speech, and Language Processing*, 19(6):1754–1769.

[47] Rojas, R. and Feldman, J. (1996). *Neural Networks: A Systematic Introduction*. Springer.

[48] Rose, D. (2018). *Artificial Intelligence for Business: What You Need to Know about Machine Learning and Neural Networks*. Chicago Lakeshore Press.

[49] Sasaki, Y. et al. (2007). The truth of the f-measure. *Teach Tutor mater*, 1(5):1–5.

[50] Schreiber, H. and Müller, M. (2018). A single-step approach to musical tempo estimation using a convolutional neural network. In *Proceedings of the 19th International Conference on Music Information Retrieval (ISMIR), Paris, France*.

[51] Schuster, M. and Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681.

[52] Stansbury, D. E. (2018). The clever machine, topics in computational neuroscience and machine lerning. `https://theclevermachine.wordpress.com/`. [Online; accessed 10-Jan-2019].

[53] Tzanetakis, G. and Cook, P. (2002). Musical genre classification of audio signals. *IEEE Transactions on speech and audio processing*, 10(5):293–302.

[54] Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer.

[55] Zhou, R., Mattavelli, M., and Zoia, G. (2008). Music onset detection based on resonator time frequency image. *IEEE Transactions on Audio, Speech, and Language Processing*, 16(8):1685–1695.

APPENDIX A

DERIVATIONS

A.1 Derivation of the Sigmoid Function

First the reciprocal rule is applied:[45]

$$s'(x) = \frac{d}{dx}\left(\frac{1}{e^{-x}+1}\right) = \frac{-\frac{d}{dx}(e^{-x}+1)}{(e^{-x}+1)^2} \tag{A.1}$$

Second, the derivative is taken of each part in the numerator. The derivative of constant 1 is 0.

$$= \frac{-\frac{d}{dx}(e^{-x})}{(e^{-x}+1)^2} \tag{A.2}$$

Next, the exponential function rule and the chain rule are applied:

$$= \frac{-(e^{-x}\cdot\frac{d}{dx}(-x))}{(e^{-x}+1)^2} \tag{A.3}$$

Again, the derivative of each part is taken:

$$= \frac{-(-1)\cdot e^{-x}}{(e^{-x}+1)^2} \tag{A.4}$$

[45]The derivation steps were retrieved from `https://www.heatonresearch.com/aifh/vol3/deriv_sigmoid.html` [accessed 15 May, 2019]

The double negative cancels out:

$$= \frac{e^{-x}}{(e^{-x}+1)^2} \tag{A.5}$$

As Heaton [28] describes, it is convenient to further transform this equation for "computational efficiency:"

$$= \left(\frac{1}{e^{-x}+1}\right)\left(\frac{-e^{-x}}{e^{-x}+1}\right) \tag{A.6}$$

The commonly used form of the sigmoid's derivative can now be denoted as:

$$s'(x) = s(x)(1-s(x)) \tag{A.7}$$

APPENDIX B

DETAILED CONFIGURATIONS OF EXPERIMENTS

Exp.	Chunk	Margin	Hot-patch	Mel bins	Frame	Sample rate	Hop Size	Batch Size	Spectrogram	Optimizer	Dataset	Training set	Test set	Epochs
Preliminary Experiments CNN – Phase I														
1	20/40/80	6/15/35	8/10/10	0/40	1024	22050	441	128	Log-Filt	Adadelta	GTZAN	Limited	Limited	20
2	500	245	10	40	1024	22050	441	32/128	Log-Filt	Adadelta	GTZAN	Limited	Limited	20
3	100/200/300/400	47/97/147/197	6	40	1024	22050	441	256	Log-Filt	Adadelta	GTZAN	Limited	Limited	5
4	500	247	6	20	4096	22050	441	128	Log-Filt/Diff	Adadelta	GTZAN	Limited	Limited	15
Phase I – Frame-Based versus Excerpt-Based Datasets														
5	50/100/200	20/45/95	10	40	1024	22050	441	128	Log-Filt	Adadelta	GTZAN	Limited	Limited	20
Phase I – Preliminary Experiments RNN														
6	20/80	6/35	8/10	40	1024	22050	441	128	Log-Filt	Adam	GTZAN	Limited	Limited	20
Phase II – Enhenced Architecture Experiments														
7	300	140/147/149	20/6/2	40	1024	22050	441	32	Log-Filt	Adam	GTZAN	Full	Full	4
8	40/100/300/500	19/49/149/249	2	40	1024	22050	441	32	Log-Filt	Adam	GTZAN	Full	Full	4
9	100	49	2	40	1024/2048/4096	22050	441	32	Log-Filt	Adam	GTZAN	Full	Full	4
10	100	49	2	40	1024	22050	441	32/256	Log-Filt	Adam	GTZAN	Full	Full	4
Phase II – Cross-Validation of Datasets														
11	500	249	2	40	1024	22050	441	32	Log-Filt	Adam	GTZAN/SMC	Full	Full	4

Table B.1 Configuration Overview. The table shows the full configurations of all final experiments. The highlighted fields mark the varied parameters of an experiment.