Technical University Berlin
Faculty I - Humanities
Summer Term 2020

Audio Technology Group Native Instruments GmbH

in collaboration with

Beat Detection on Popular Music with Varying Tempo using Neural Networks

Master Thesis

Anyere Bendrien

✉ anyere.bendrien@campus.tu-berlin.de

Matriculation Number: 382004

supervised by

Prof. Dr. Stefan Weinzierl

D.Sc. Julian Parker

October 20, 2020

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt gegenüber der Fakultät I der Technischen Universität Berlin, dass die vorliegende, dieser Erklärung angefügte Arbeit selbstständig und nur unter Zuhilfenahme der im Literaturverzeichnis genannten Quellen und Hilfsmittel angefertigt wurde. Alle Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, sind kenntlich gemacht. Ich reiche die Arbeit erstmals als Prüfungsleistung ein. Ich versichere, dass diese Arbeit oder wesentliche Teile dieser Arbeit nicht bereits dem Leistungserwerb in einer anderen Lehrveranstaltung zugrunde lagen.

Titel der schriftlichen Arbeit

Beat Detection on Popular Music with Varying Tempo using Neural Networks

Verfasser

Anyere Bendrien
Matrikelnummer: 382004

Betreuende Dozenten

Prof. Dr. Stefan Weinzierl
D.Sc. Julian Parker

Mit meiner Unterschrift bestätige ich, dass ich über fachübliche Zitierregeln unterrichtet worden bin und verstanden habe. Die im betroffenen Fachgebiet üblichen Zitiervorschriften sind eingehalten worden. Eine Überprüfung der Arbeit auf Plagiate mithilfe elektronischer Hilfsmittel darf vorgenommen werden.

Berlin, October 20, 2020

Zusammenfassung

Aus Beat-Informationen lässt sich eine Vielzahl von zeitlichen Musik-Eigenschaften herleiten. Eine Audioanwendung zum Mischen von Musik könnte diese Informationen nutzen, um die zeitliche Grundstruktur von Musikstücken automatisch anzugleichen. Insbesondere für Musik, die von Menschen gespielt wird, ist dies eine herausfordernde Aufgabe, da die Musik subtilen zeitlichen Variationen unterliegen kann. Mit einem Computer hingegen lässt sich Musik von einer festen Timeline exportieren und folgt daher einer präzisen zeitlichen Struktur. Mit dieser Arbeit wird ein neuer Datensatz erstellt, der für das Trainieren neuronaler Netze zur Beat-Erkennung optimiert ist. Zusätzlich wird ein Konzept zur automatischen Vorhersage von Beat-Informationen aus Musik mit Hilfe neuronaler Netze erforscht. Besonderes Augenmerk wird auf Musik mit variierendem Tempo gelegt. Ziel ist es, diese Technologie für den Einsatz in einem Produkt robuster zu machen mit einem Fokus auf reduzierte Rechenkosten.

Abstract

A multiplicity of temporal music properties can be deduced from beat information. An audio application like a music mixing software could exploit this information to automatically align the temporal basis structure of music pieces. This is a challenging task especially for music played by humans who are not as precise as a machine exporting audio from a fixed timeline. With this thesis, a new dataset suited for training neural networks on beat detection will be created. In addition, a concept for automatically predicting beat information from the music itself, using a neural network, is being researched. Special attention is given to music with varying tempo. The aim is to make this technology more robust for production use cases with a focus on decreased computational cost.

Contents

Acknowledgements

I would like to thank…
Julian Parker for his comprehensive support in the process of writing this thesis, being a great help for making the right decisions and maintaining focus on the relevant topics.
André Bergner for building my fundamental knowledge of machine learning.
Mickael Le Goff for introducing me to the topic of beat detection.
Moritz Heppner for giving valuable insights into the Maschine codebase.
The **Traktor Pro Team** for the appreciation of my research.
Native Instruments for the opportunity of collaboration and providing infrastructure.
Jonas Köhler, **Maximilian Wagenbach**, **Moritz Lehr**, **Sebastian Wolf** and **Nils Bultja** for proofreading.
My family for being supportive my whole life.

Thank you !

List of Figures

List of Tables

Notation

$\boldsymbol{x}$	A vector
$\mathbf{0}$	Vector filled with zeros
$\odot$	Operator for element wise multiplication also known as the Hadamard product
$*$	Convolution operator
$\oplus$	Concatenation operator
$\sqrt[\odot]{}$	Element wise applied square root
$_t$	Iteration step of an algorithm
$\boldsymbol{X}$	A data set
$^{(i)}$	Layer within a NN

1 Introduction

When asked about their preferred musical style, their favorite recording, or performance, many music listeners will most likely answer with a rather nebulous description containing ambiguous terms like 'feeling' or 'groove' or, and this is often a more solid way to describe music: rhythm and melody. The two terms seem ubiquitous but are seldom clearly defined by the person saying them. Both terms can loosely be described by the same definition: A series of events structured temporally, which in their context become musical. It becomes apparent here that the beat is an integral part of the music listening experience for popular music.

Compared to other kinds of music, popular music has developed a large number of permutations, styles, genres, sub-genres, movements and counter-movements but the set of rhythmical choices has been rather limited. Nevertheless, music listeners prefer certain recordings over others. One reason for this might be found in what is often called the 'performance' of the players. A term that might be defined as tiny shifts in the series of (musical) events, small imperfections, deliberate accentuations that, compared to a static grid of notes would be interpreted as mistakes, but ultimately makes a performance feel human. Depending on what type of music a particular listener favors, this might be seen as the 'normal' or even 'real' way of making music, but for many people, contemporary popular music is exactly the opposite: made with machines and with computers in particular. Here a programmed sequence is always played back exactly the way it was programmed.

Arguably the task at hand should be to help these imperfections, the 'feeling', the 'groove', the partial departure from the fixed grid of the beat to find their way into the context of modern popular music. And since dj-culture has been on the rise for years and is just on the verge of taking over the 'live'-music industry, it is imperative to emphasize the way djs are working which is heavily relying on 'the grid' as the basis of mixing.

What this thesis proposes is to expand on the already existing method of beat tracking in a way that will enable digital software to recognize and incorporate said imperfections into the practice of modern music production and digital djing.

Here the focus will be put on music with varying tempo. The aim is to make the technology more robust for production use cases with a focus on decreased computational cost. The task of giving modern technology the tools to interpret and work with the human element will play an integral part in the ongoing process of modern music development. The potential of digital djing software is already huge, but often at its limits when it comes to leaving 'the grid' or recognizing and dealing with the lack of it. Working on this frontier will not only enable new music to be produced but also traditionally made music to be incorporated into contemporary dj culture.

1.1 Motivation

Rhythm is a fundamental attribute of music with distinguishable elements such as onset, beat, tempo and meter defining its temporal structure. Automatically extracting this rhythm related information plays a decisive role in machine-human interaction within a musical context. One of the most basic rhythmic elements building the foundation for more advanced temporal metrics is the onset. It refers to the beginning of a musical event like a playing note or sound. A beat is a regular repeating event denoting the basic unit of time. An intuitive analogy of a beat is how a listener of music would tap or clap to it. The beat is a similar attribute to the onset regarding the likeliness of onsets being aligned with beat events.

Detecting beats in music is part of the music information retrieval (MIR) field of research. Since the success story of machine learning (ML) for image and language processing, ML is getting more attention in the audio domain. ML gives new approaches to solve all kinds of MIR related tasks. In general ML tries to deduce specialized information from data. Therefore a NN has to be trained with aggregated training data, the more the better. This usually includes data related annotations giving the answer to an intended problem to solve. Creating the annotations for the data in the first place is mostly not automatable and therefore a time-consuming task. This holds especially true if a high quality of the annotations is desirable.

In the domain of digital djing, there is a huge potential for supporting a live performing dj with algorithms. *Traktor Pro* from Native Instruments (NI) is a digital dj application that implements a multitude of features in this regard allowing the dj to focus on more high-level features. One of the most important functionality is the automatic synchronization of two simultaneously playing tracks. Listening to both at the same time is more pleasing if their rhythmic structure is aligned. This works very well on music with a steady tempo but those who have unsteady tempo can not be aligned that easily. In such a situation a dj usually has to compensate for the repeatedly appearing drift of both music pieces by adjusting the tempo himself.

Varying tempo is likely in music recorded from a band without the use of a metronome. This is common for music like funk which was recorded before the personal computer got mainstream in the recording studios. The process of digitalizing music from analog media like vinyl or tape could also introduce unsteady tempi especially if the used hardware is not working precisely. Acquiring the beat structure on music with varying tempo could be useful to support a dj in the process of mixing such music within a live situation. The beat information could be used to automatically compensate for the drift between the to be mixed music pieces.

1.2 Related Work

The research field of beat detection has a long history reaching back to 1985 with the work of Schloss [25] and is traditionally based on processing a list of onsets. Starting in 2011, the first beat tracking system based on an artificial NN has been introduced with the work of Böck et al. [2]. Because this thesis tries to solve the problem of tracking beats in music with a NN as well, only related work also utilizing NNs is presented here.

The work of Böck et al. [2] utilizes a recurrent bidirectional long short-term memory (BLSTM) network to perform a frame by frame beat classification of the input signal. The signal is pre-processed into spectral features and then directly transformed into a beat activation function via the network. In a post-processing stage, the predominant tempo is extracted with an autocorrelation for further refinement of the beat activation function. Therefore they present two algorithms, one for steady tempo and one adapting to tempo changes.

Following up work of Böck et al. [4] extend their first approach [2] with a concept of modeling different music styles independently and thus more specialized. The most appropriate beat activation function for the input signal is then chosen from all models. In a post-processing stage, a dynamic Bayesian network (DBN) extracts the final beat positions from the selected beat activation function. Their work reaches state-of-the-art performance.

The idea of modeling tempo and meter in music with a DBN originates from the work of Whiteley et al. [28]. They introduced the idea of a probabilistic bar pointer model which maps an input signal to one cycle of a latent, periodic-rhythmical pattern. It is capable of detecting changes in rhythmic pattern and meter which makes it an important ingredient for the process of tracking beats in music. The work of Krebs et al. [20] improves on the original bar pointer model by drastically reducing time and memory complexity. Furthermore, their approach improved the beat and downbeat tracking performance as well.

Dannenberg et al. [7] have characterized tempo changes in musical performances.

They developed mathematical models to describe tempo variations which could be exploited for synthesized performances.

One of the most successful methods for beat detection is a joint beat and downbeat tracking system proposed by Böck et al. [5]. Their model is also based on a recurrent BLSTM network in addition to a DBN in a post-processing stage to acquire the global most suitable sequence of beat timestamps. The output of the system clearly distinguishes between beats and downbeats. Since 2016 their approach is in the top list of the *Music Information Retrieval Evaluation eXchange* (MIREX) beat tracking task on all datasets and can be considered the current state of the art[1]. The work of Davies et al. [9] has made a big step towards forming a foundation of beat tracking evaluation.

With *madmom* Böck et al. introduced an open-source audio processing and MIR library[2] written in Python [6]. This library provides implementations of beat tracking processors [4, 5] and evaluations which this thesis is making use of.

Vogl et. al. [27] are extending the ideas of [5] by introducing additional covolutional neural network (CNN) layers in their NN architecture. Since their main focus was drum transcription they concluded that their method could also be applied for beat detection. With *WaveNet* [22] a generative model for raw audio was presented which is based on CNNs. They introduced the concept of dilated convolution to exponentially increase the receptive field of their CNN architecture. Recent studies show that CNN architectures are also capable of modeling long term dependencies similar to recurrent long short-term memory (LSTM) networks. "An Empirical Evaluation of Generic Convolutional and Recurrent Networks" [1] shows that dilated CNNs combined with residual connections, called temporal convolutional networks (TCNs), outperform generic recurrent architectures such as LSTMs on a majority of sequence modelling tasks. They also show that TCNs exhibit longer memory than recurrent architectures with the same capacity. Adapting the concept of dilated CNNs for beat detection will be researched further in this thesis.

The work of [8] present a beat-tracking system implemented with a TCN. They demonstrate three promising attributes of TCNs.

1. They achieve state-of-the-art-performance.
2. They are suited for parallel processing and highly efficient.
3. They require a small number of weights.

Further, they suggest exploring the potential of operating on raw audio itself and simultaneously modeling beat and downbeats.

[1] MIREX results from 2016, 2017 and 2018
`https://www.music-ir.org/mirex/wiki/2018:MIREX2018_Results`

[2] `https://github.com/CPJKU/madmom`

Böck et al. [3] are using a CNN highly inspired by the WaveNet architecture to extract beats and tempo in a multi-task approach. Further, they show that the accuracy of one task can be improved by training the other.

Purwins et al. [23] are giving a review of state-of-the-art deep learning techniques in the context of audio signal processing. Among other things they comment on the approach of working directly with raw audio features on page 10:

> "Raw waveforms avoid hand-designed features, which should allow to better exploit the improved modeling capability of deep learning models, learning representations optimized for a task. However, this incurs higher computational costs and data requirements, and benefits may be hard to realize in practice."

1.3 Contribution of this Thesis

With this thesis, a new training dataset for beat detection will be created focusing on popular music with varying tempo. This dataset will be generated from the *Maschine Expansion*s (MEs) content of NI, which spreads over a big variety of genres in popular music. The quality of the dataset annotations will have a high priority. Additionally, a neural network architecture suitable for inferring beat positions from music will be presented and implemented.

As a novel approach a model of the proposed network architecture will be trained directly on raw audio data within the time domain instead of pre-processed fast fourier transoform (FFT).

The beat predictions from the NN will be refined using a DBN. The accuracy of the trained model will be evaluated against the beat detection implementation of [4] and [5] as a baseline. Furthermore, the performance of both baseline models and the proposed model on the new dataset will be presented.

1.4 Outline

The following chapter gives an introduction to machine learning, covering the for this thesis relevant topics 2.1. Further, the approach of implementing a beat tracking system is described in detail 2.2. This is followed by an evaluation of the implementation 3 and presentation of the datasets 3.1. A discussion 4 summarizes the findings of this thesis.

2 Method

2.1 Machine Learning

Machine learning is the science of solving statistical problems, like pattern recognition or regression, using efficient algorithms. Typically this involves a process called *training*, which iteratively improves the performance of a model on a problem with some data given. This chapter gives an introduction to machine learning in particular neural networks and how to design and train them for beat recognition.

There are two classes of learning problems in ML, *supervised* and *unsupervised* learning. Unsupervised learning algorithms search for an alternative representation of some data without supervision during this process. E.g. this means reducing the complexity of the data according to some constraint while preserving as much information as possible [14, ch. 5.8]. This thesis focuses on the supervised learning approach, thus the topic of unsupervised learning will not receive further attention.

2.1.1 Supervised Learning

In supervised learning a dataset is supplemented with annotations representing a specialized observation about the data. These annotations are also known as *ground truth*, *target* or *labels* conditioning the data. Datasets are organized in pairs of input data $\boldsymbol{x}_i$ and corresponding labels $\boldsymbol{y}_i$ with m being the overall number of training samples $\boldsymbol{X} = \{(\boldsymbol{x}_1, \boldsymbol{y}_1), ..., (\boldsymbol{x}_m, \boldsymbol{y}_m)\}$. In practice, such a dataset is just a finite observation of a real problem and therefore a noisy approximation of it. Supervised learning tries to predict the labels $\boldsymbol{y}$ given some features $\boldsymbol{x}$ in a dataset.

Assuming an unknown function f^* exists, modeling the relationship between a ground truth distribution $\boldsymbol{y}$ and some data $\boldsymbol{x}$ with some small valued noise ϵ

$$\boldsymbol{y} = f^*(\boldsymbol{x}) + \epsilon, \tag{2.1}$$

supervised learning tries to approximate f^* using a parameterized family of curves

$$g(\boldsymbol{x}; \boldsymbol{\theta}) \approx f^*(\boldsymbol{x}). \tag{2.2}$$

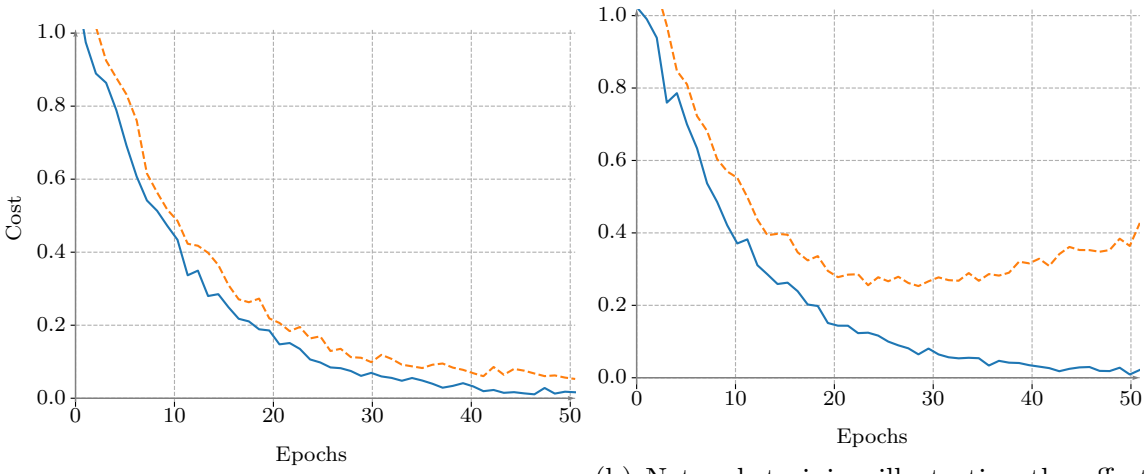
The right parameterization $\boldsymbol{\theta}$ of g has to be found to model the relationship between $\boldsymbol{x}$ and $\boldsymbol{y}$ without getting too specialized on the finite observation itself. In this process, a tradeoff between variance and bias has to be made. A high error in variance indicates that g is modeling the noise of the finite observation which leads to a bad generalization of the problem. If the bias error is too high g misses the relationship between the features and their target. Diverging from the right balance of variance and bias is known as overfitting and underfitting of which is detailed in the following section [14, ch. 5.7].

Overfitting

Fitting a very powerful model with a high variance to a small training data set could lead to perfect memorization and in that case perfect performance on this training set. If the training data set does not represent the data source completely, the model may perform a lot worse for new data. This phenomenon is called *overfitting*. A crucial step for good performance outside the training data distribution is thus carefully tuning the model's flexibility while measuring the generalization performance. The general approach here is to achieve that is to maintain at least two datasets, one for training the model parameters and the other just for performance evaluation. The important distinction between the training and evaluation stage is that the evaluation dataset is not used for the optimization of the model. The terminology for a model to pass through both stages is an epoch.

When training a model going from epoch to epoch a descending training error is desirable to become as small as possible. Overfitting describes the situation where the error of the evaluation starts to diverge from the curve progression of the training error. Figure 2.1b visualizes this scenario of the evaluation error rising again while the training error is still descending. This phenomenon is a high indicator for the model learning too specialized information of the training data set which is not applicable in the evaluation data set anymore. Usually, the evaluation error is a small margin above the training error since it is harder for the network to truthfully predict data it has never seen before. In a good performing training situation, both error values share the same curve progression just having a small gap as seen in figure 2.1a.

The opposite of overfitting is called underfitting and describing the situation in which the model is not able to converge to a satisfactory result at all. Both situations are usually in direct connection with the capacity or parameter count of



(a) A good performing network training

(b) Network training illustrating the effect of overfitting

Figure 2.1: Two fictitious training scenarios over several epochs showing the effect of overfitting. The training error is the solid line and the validation error is the dashed line.

a model. The capacity in turn depends on the depth and width of the network. Having a low capacity may result in underfitting and having a high capacity could lead to overfitting by just memorizing properties of the training data. Finding the right balance between both scenarios is important to the model's performance [14, ch. 5.2]. Section 2.1.4 is presenting several approaches to counteract overfitting.

2.1.2 Neural Networks

Recent work [5, 8] has shown that NNs provide useful function classes for solving the beat recognition problem in a supervised learning fashion. An artificial neural network is a collection of connected computation nodes whose mode of operation is inspired by neurons in the human brain. A common structure for it is following a feed-forward principle consisting of serial chained *layers* which are clusters of parallel *neurons*. The number of layers of a NN is called its *depth*. The number of neurons in each layer is its width. The connections between layers and their neurons are called *weights*.

Probably the most straightforward layer configuration is a dense layer also known as a fully connected layer. Each node in a dense layer holds a connection from all neurons of its preceding layer as shown in figure 2.2. For m neurons in the dense layer and n neurons in the preceding layer, this results in a $m \times n$ matrix of weight connections. Applying this weight matrix to a vector has a runtime cost of $O(m \times n)$.

Data is fed into the input layer and the result is passed on to the following layers (also called *hidden layers*) until reaching the final output layer (see figure 2.3). The values of the output layer neurons are the predictions of the NN. This process is

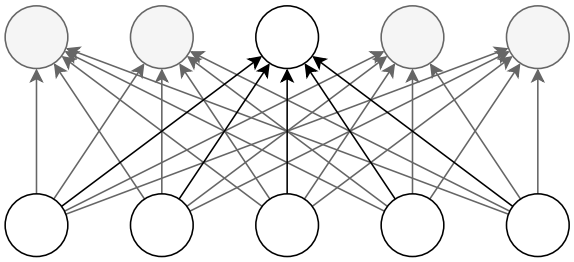


Figure 2.2: A dense layer configuration. Each neuron of the upper layer holds a connection from all neurons of its preceding layer.

called *feed-forward pass*.

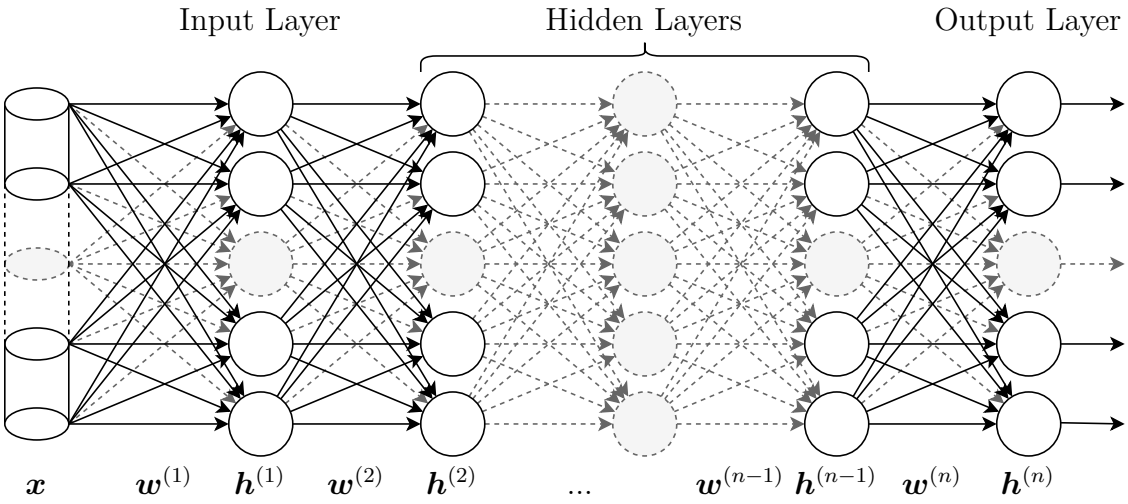


Figure 2.3: A common fully connected feed forward NN architecture.

Feed-Forward Pass

During a forward pass, intermediate results are held by the neurons and reflect their *activation* with respect to the input data. With a fully connected feed-forward NN architecture as seen in figure 2.3 each neuron j in layer k has individual weights $w_{ij}^{(k)}$ for all neurons i from the preceding layer $k-1$ or in case of the input layer for the input data $\boldsymbol{x}$.

The activation of a neuron is the weighted sum of input activations from all neurons in the preceding layer. The weights control relevancy of preceding data for the current neuron's activation. This sum could take arbitrary runaway values. Therefore a non-linear activation function g squeezes it into a bounded range. Non-linearities are important for modeling complex relationships and to overcome the limitations of linear models. With a linear model, it does not matter how many layers a NN has since there could always be a synonymous configuration with just one layer. That is because adding together linear functions still results in a linear

function. Purely linear models are not capable of solving nonlinear problems like modeling the xor function as Marvin Minsky already pointed out in the 1940s [14, ch. 6.6].

Often a *bias* b is added right before applying the activation function. It sets a threshold that the neuron has to exceed to become active. All weights and biases combined are called the parameters of a NN. For the first neuron in the first layer $h_1^{(1)}$ this leads the the equation

$$a_1^{(1)} = g^{(1)} \left(\sum_i w_{i,1}^{(1)} x_i + b_1^{(1)} \right) \tag{2.3}$$

with $\boldsymbol{x}$ as input data. In vector notation with

$$\boldsymbol{h}^{(1)} = g^{(1)} \left(\boldsymbol{w}^{(1)\top} \boldsymbol{x} + \boldsymbol{b}^{(1)} \right) \tag{2.4}$$

being the input layer and

$$\boldsymbol{h}^{(k)} = g^{(k)} \left(\boldsymbol{w}^{(k)\top} \boldsymbol{h}^{(k-1)} + \boldsymbol{b}^{(k)} \right) \tag{2.5}$$

describing all successive layers with $k < K$ [14, ch. 6.4].

Equations 2.4 and 2.5 are defining a whole feed-forward network architecture with a depth of K. Having the basic definition of a NN in place the following section describes how to optimize the parameters to approximate the unknown target function f^* of equation 2.2.

2.1.3 Optimization of Neural Networks

The critical step for training a NN is finding weights and biases, such that it can perform predictions on previously unseen data. The first step is choosing a loss function which serves as a proxy for the model performance. In a second step, we try to find parameters that minimize this loss both for training and unseen data. There are many choices for designing loss functions. For regression problems, it is common to use the *mean-squared error*. For binary classification problems, a common loss function is the *binary cross-entropy*.

Mean-Squared Error

Starting in the 1980s and 1990s the mean squared error (MSE) was popularized to compute losses. Given a training sample $(\boldsymbol{x}, \boldsymbol{y})$ and the final output prediction $\hat{\boldsymbol{y}}$ of the NN, the difference $\boldsymbol{z} = \hat{\boldsymbol{y}} - \boldsymbol{y}$ describes how far the prediction is deviating from

the target. It is the average value of the squared elements of a vector $\boldsymbol{z}$

$$\text{MSE}(\boldsymbol{z}) = \frac{1}{m}\sum_{i}^{m} z_i^2. \tag{2.6}$$

It is important to note that negative values can not cancel each other out since their squared value will always be positive. Otherwise, this could lead to unpredictable behavior and would make training less reliable [14, ch. 5.2.4].

Binary Cross-Entropy

The cross-entropy measures the difference between two probability distributions. In the domain of ML these are the probability distribution of the data set labels $p(\boldsymbol{y})$ and the probability distribution of the model predictions $p(\hat{\boldsymbol{y}}|\boldsymbol{x})$ for the corresponding input $\boldsymbol{x}$. It is computed via the negative sum of the logarithm of the model prediction times the label

$$H(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{i} y_i \; log_2(\hat{y}_i). \tag{2.7}$$

Minimizing the cross entropy results in a $\hat{\boldsymbol{y}}$ that requires as few extra bits as possible to encode classes from $\boldsymbol{y}$ [14, ch. 5.5]. Using sigmoid (see section Sigmoid) as the output unit of a NN assures that all predictions are probabilities. Seeing the sigmoid function in conjunction with cross-entropy loss is a common setup for a binary optimization problem.

Minimization of average loss

Given a loss function L, we can now analyze a model's performance by its average performance on the data set for a fixed parameter $\boldsymbol{\theta}$:

$$J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}), \boldsymbol{y}_i). \tag{2.8}$$

Under the assumption that the training set is representative for the whole data distribution, by minimizing $J(\boldsymbol{\theta})$ we will thus achieve a good model performance for unseen data.

Minimization via Gradient Descent

One way to minimize $J(\theta)$ is using *gradient descent*, an algorithm to numerically find local minima of differentiable scalar functions. This algorithm works via the following three steps.

1. All parameters of a NN model are initialized with an initial value which is explained in more detail in section Parameter Initialization.

2. The gradient of the cost function with respect to all parameters $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is computed. This can be achieved in an efficient way using the so called back-propagation algorithm (see section Efficient gradient computation using back-propagation).

3. The computed negative gradient pointing towards a (local) minimum of the cost function is used to iteratively adjust the model parameters $\boldsymbol{\theta}$ in small increments.

Each update thus has the form

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \epsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t), \tag{2.9}$$

where the scalar value ϵ is called learning rate. This procedure will converge on a local minimum $\boldsymbol{\theta}^*$ of J with $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^*) = \mathbf{0}$ once $t \to \infty$ [14, ch. 4.3].

Difficulties using Gradient Descent for NN optimization in practice

Minimizing the average loss of a NN in practice is a challenging task due to the following problems.

1. Optimization with gradient descent struggles to find a global minimum on a non-convex function due to many possibly bad performing local minima. Furthermore, the whole dataset could be large which makes updates slow. A practical solution is to apply gradient descent on *mini-batches* doing many fast but noisy updates rather than few slow and exact ones. This is known as the *stochastic gradient descent (SGD)* algorithm as described in section Non-convex optimization with Stochastic Gradient Descent.

2. The convergence could be slow depending on the curvature of the average loss function. It could take a long time to escape a bad performing local minima or saddle points. A solution to this is to dynamically adapt the learning rate (see section Importance of controlling learning rate) and introduce a momentum to the update process (see section Faster convergence with Momentum).

3. The computation of the gradients is an extensive and complex task. An efficient gradient computation is possible with the *back propagation* algorithm as described in section Efficient gradient computation using backpropagation.

4. A poor initialization of model parameters as well as the problem of vanishing gradients (see section Vanishing/Exploding Gradients in deep architectures) could lead to tiny update steps or long convergence time. These problems can be tackled by applying improved initialization schemes (see section Parameter Initialization), self-normalizing layers (see section Weight Normalization) and utilizing skip-connections (section Skip-Connections and Residual Networks).

Non-convex optimization with Stochastic Gradient Descent

During SGD we evaluate J and ∇J only for a small random subset of the whole data (also called a *mini-batch*) before performing a gradient descent update step. The reduced size of a mini-batch speeds up a single iteration, especially for large data sets. Furthermore, the stochasticity introduced from random subsamples helps to escape local minima and saddle points of J due to the increased variety of approximated gradients.

On each iteration the SGD algorithm uniformly takes m' training samples from a data set $\boldsymbol{X}$ building a minibatch $\boldsymbol{B} = \{(\boldsymbol{x}_1, \boldsymbol{y}_1), ..., (\boldsymbol{x}_{m'}, \boldsymbol{y}_{m'})\} \subseteq \boldsymbol{X}$ with size m'. The gradient equation becomes

$$\boldsymbol{g}_t = \frac{1}{m'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m'} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}_t), \boldsymbol{y}_i) \tag{2.10}$$

for each time step t. The size m' of a minibatch usually stays fixed during training and is decoupled from the data set size m which may vary. The most basic version of SGD with $m' = 1$ comes with the highest variance in approximating the desired gradient. Thus increasing m' will decrease the variance leading to more stable convergence on a minimum of the cost function. In a final training typical sizes for m' are in a range of 32 to 256 samples. With m training samples in a data set the computational cost of the basic gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is $O(m)$. Using SGD with $m' << m$ training samples the computational cost will be much lower at $O(m')$ [14, ch. 5.9].

Importance of controlling learning rate

One crucial part using optimizing with gradient descent is the learning rate ϵ. If ϵ is set too high, J might be oscillating around a minimum without ever reaching it or even steering away from it. If ϵ is set too low, the convergence of J would be slow and the training might get stuck.

It is common practice to expand the SGD algorithm with a linear learning rate

decay $\tau \in \mathbb{N}$ transitioning the learning rate from ϵ_0 to ϵ_τ on each iteration t

$$\epsilon_t = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau \tag{2.11}$$

with $\alpha = \frac{t}{\tau}$. Usually once $t \geq \tau$ the learning rate wont be adjusted anymore and stays constant at ϵ_τ. With $\epsilon_0 > \epsilon_\tau$ this technique counteracts overshooting a minimum of the cost function going back and forth over it while reaching a gradient of 0 [14, ch. 8.3.1]. For the sake of simplicity, the learning rate decay will not be labeled explicitly in this thesis.

AdaGrad and RMSProp The AdaGrad [11] and RMSProp algorithms extend stochastic gradient descent with the concept of per parameter adapting learning rates. Instead of using one learning rate each parameter gets an individually tailored learning rate. In AdaGrad the adaptive learning rates $\boldsymbol{r}$ are scaled inversely proportional to the square root of the sum of all the historical squared values of the gradient

$$\boldsymbol{r}_t = \boldsymbol{r}_{t-1} + \boldsymbol{g}_t \odot \boldsymbol{g}_t, \tag{2.12}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\epsilon}{\delta + \sqrt[\odot]{\boldsymbol{r}_t}} \odot \boldsymbol{g}_t \tag{2.13}$$

where δ is a constant for numerical stability around 10^{-7} and avoids division by zero [11, 14, ch. 8.5.1].

The RMSProp algorithm modifies AdaGrad by changing the gradient accumulation into an exponentially weighted moving average. This turns equation 2.12 into

$$\boldsymbol{r}_t = \rho\boldsymbol{r}_{t-1} + (1 - \rho)\boldsymbol{g}_t \odot \boldsymbol{g}_t \tag{2.14}$$

with $\rho \in [0, 1)$ being the decay rate and leaving equation 2.13 as is. Due to the exponential decay the history from the distant past does not have as much influence as it does have in the basic AdaGrad algorithm. This prevents a shortcoming of AdaGrad of potentially reaching an infinitely small learning rate due to the steady accumulation of always positive squared gradients in the nominator. In such a situation the algorithm would not acquire any more knowledge [14, ch. 8.5.2].

Faster convergence with Momentum

Inspired by physical motion stochastic gradient descent with momentum introduces a new variable v modeling a velocity-like behavior while moving through the parameter space of the cost function. This has several benefits like passing through flat

regions of the solution space with more pace and overcoming local minima due to the accumulated momentum.

SGD with simple momentum The velocity variable accumulates the exponential decaying moving average of the past negative gradients. A hyperparameter $\alpha \in [0, 1)$ controls the weight of the previously contributed gradients or in other words the exponential decay of them. Starting with an initial value of $\boldsymbol{v} = \mathbf{0}$ this consolidates into the following update rules

$$\boldsymbol{v}_t = \alpha \boldsymbol{v}_{t-1} - \epsilon \boldsymbol{g}_t, \tag{2.15}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{v}_t. \tag{2.16}$$

The velocity becomes greater when successive gradients are pointing in the same direction, which accelerates the learning process. If the successive gradients are alternating the velocity becomes smaller and averages into a mutual direction of the successive gradients [14, ch. 8.3.2].

Adam The adaptive moment estimation (Adam) algorithm combines RMSProp with momentum. It computes an exponentially decaying average of past gradients similar to SGD with momentum

$$\boldsymbol{s}_t = \rho_1 \boldsymbol{s}_{t-1} + (1 - \rho_1)\boldsymbol{g}_t \tag{2.17}$$

as well as an exponentially decaying average of past squared gradients like RMSProp

$$\boldsymbol{r}_t = \rho_2 \boldsymbol{r}_{t-1} + (1 - \rho_2)\boldsymbol{g}_t \odot \boldsymbol{g}_t \tag{2.18}$$

with ρ_1 and $\rho_2 \in [0, 1)$ being the decay rates. The estimates of the first moment $\boldsymbol{s}_t$ being the mean and the second moment $\boldsymbol{r}_t$ being the off-centered variance tend to have a bias towards zero especially in the beginning of the training when initialized with $\mathbf{0}$. Therefore two bias correcting terms

$$\hat{\boldsymbol{s}} = \frac{\boldsymbol{s}_t}{1 - \rho_1^t}, \tag{2.19}$$

$$\hat{\boldsymbol{r}} = \frac{\boldsymbol{r}_t}{1 - \rho_2^t} \tag{2.20}$$

are computed to counteract this tendency. Common values for ρ_1 and ρ_2 proposed by the authors are 0.9 and 0.999 respectively [19]. Using the bias corrected terms

the final update rule of Adam becomes

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\epsilon}{\delta + \sqrt[\odot]{\hat{\boldsymbol{r}}}} \odot \hat{\boldsymbol{s}} \tag{2.21}$$

being very similar to RMSProp. Adam is one of the most popular optimization algorithms for ML due its generally good performance [14, ch. 8.5.3].

Efficient gradient computation using backpropagation

The gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ can be computed efficiently using the backpropagation algorithm. It consists of two steps: a forward-pass and a backward-pass.

During the forward-pass we propagate an input $\boldsymbol{x}$ through a NN and compute the scalar cost $J(\boldsymbol{\theta})$. In this step, it is crucial to store the activations of the intermediate hidden layers. It is best practice to build some kind of computational graph keeping track of this.

In the backward pass we compute the gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ using these stored values. Going backward through the network starting at the output layer we compute local gradients for each layer via partial differentiation and recursively apply the chain rule

$$\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = \sum_i \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{h}^{(i)}} \frac{\partial \boldsymbol{h}^{(i)}}{\partial \theta_j} \tag{2.22}$$

$$= \sum_i \frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{h}^{(K)}} \frac{\partial \boldsymbol{h}^{(K)}}{\partial \boldsymbol{h}^{(K-1)}} \cdots \frac{\partial \boldsymbol{h}^{(i)}}{\partial \boldsymbol{\theta}}. \tag{2.23}$$

Using the already saved intermediate activations $\boldsymbol{h}^{(i)}$ we can now use dynamic programming to recursively compute this gradient from left to right. This procedure is efficient as we only need to store K activations to compute the gradient in $O(m \times K \times D)$ where D denotes the *width* of the largest hidden layer. This algorithm was a critical optimization making NN usable in real world applications [14, ch. 6.5].

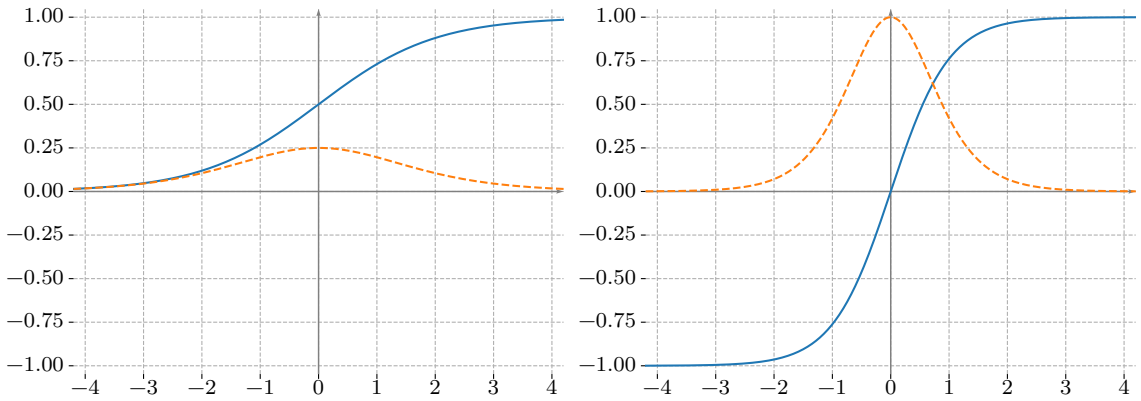
Vanishing/Exploding Gradients in deep architectures

For very deep NNs the gradients of the activation functions tend to get infinitesimally small during the backward pass which implies undiscernible updates to its affecting parameters. This holds especially true if the gradients of an activation function are always below one. By multiplying several of them together they are becoming successively through the chain rule. This results in the weights on the first layers training the slowest. The problem is known as the vanishing gradient as described by Hochreiter et al. [17] and occurs especially in very deep NN architectures. The

opposite can happen if the majority of gradients are greater than 1 which is known as the *exploding gradient* problem.

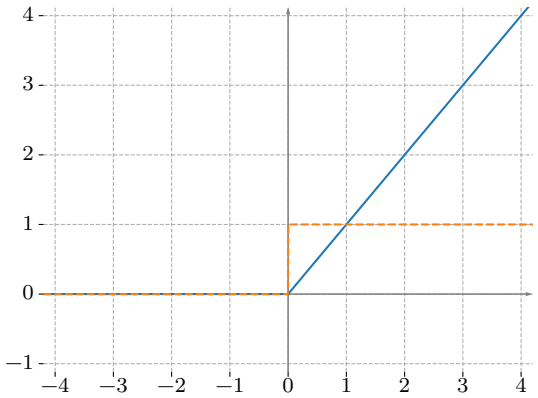
Activation Functions

Activation functions are usually the last instance of a layer in a NN and get applied element-wise on their input vector introducing non-linearities within a specific range. Figure 2.4 shows a selection of common activation functions used in ML.

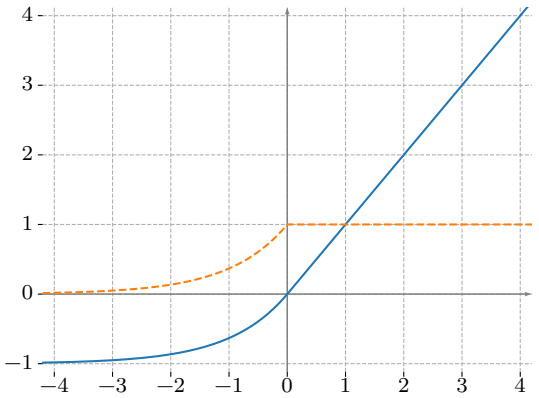


(a) Sigmoid

(b) Hyperbolic Tangent



(c) Rectified Linear Unit



(d) Exponential Linear Unit

Figure 2.4: A selection of common activation functions (solid blue line) including their first order derivatives (dashed orange line).

Sigmoid The sigmoid function also known as the logistic function has a S-shaped curve with one lower and one upper bound. It maps any real numbered value to the interval $(0, 1) \in \mathbb{R}$:

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{2.24}$$

The sigmoid is bound to the limits 0 and 1 which are unreachable at $\pm\infty$. Its derivative also known as the logistic distribution is computed by

$$\frac{d}{dx}sigmoid(x) = sigmoid(x)(1 - sigmoid(x)) \tag{2.25}$$

which can be seen in figure 2.4a. Due to its low valued gradient, it tends to suffer from the vanishing gradient problem especially for input values far away from the origin.

Hyperbolic Tangent The *tanh* function is very similar to the sigmoid function as it also follows a S-shaped curve but expands the mapping from an unipolar to a bipolar interval of $(-1, 1) \in \mathbb{R}$

$$tanh(x) = 1 - \frac{2}{1 + e^{2x}}. \tag{2.26}$$

Utilizing a bipolar interval makes *tanh* suitable for signals like raw audio. Its derivative is defined by

$$\frac{d}{dx}tanh(x) = 1 - tanh^2(x). \tag{2.27}$$

The *tanh* activation function including its derivative are visualized in figure 2.4b. Due to the similarity to the logistic function *tanh* also suffers from the vanishing gradient problem.

Rectified Linear Unit The rectified linear unit is a piecewise linear function defined by an identity function with its negative part clipped at 0

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} 0 & \text{if } x < 0, \\ x & \text{otherwise.} \end{cases} \tag{2.28}$$

Due to its piecewise linearity its computational cost is far below of the sigmoid function. This greatly improved performance of neural networks [14, ch. 6.6]. The derivative of the rectified linear unit (ReLU) is the unit step function

$$\frac{d}{dx}\text{ReLU}(x) = \frac{\text{sign}(x) + 1}{2} = \begin{cases} 0 & \text{if } x < 0, \\ 1 & \text{otherwise.} \end{cases} \tag{2.29}$$

Figure 2.4c shows the ReLU activation function and its derivative. In contrast to the sigmoid function, the ReLU does not have an upper bound and therefore does not struggle with saturated gradients on positive inputs. But for example, if a bias

gets too big and therefore all incoming values are negative it could still go inactive due to zero clipping all negative values [13].

Exponential Linear Unit The exponential linear unit (ELU) is a piecewise combination of the identity function and a down shifted exponential curve for the negative values

$$\mathrm{ELU}(x) = \min(0, e^x - 1) + \max(0, x) = \begin{cases} e^x - 1 & \text{if } x < 0, \\ x & \text{otherwise.} \end{cases} \tag{2.30}$$

Its derivative is defined by

$$\frac{d}{dx}\mathrm{ELU}(x) = \min(1, e^x) = \begin{cases} e^x & \text{if } x < 0, \\ 1 & \text{otherwise.} \end{cases} \tag{2.31}$$

Both functions can be seen in figure 2.4d.

Skip-Connections and Residual Networks

In residual networks, the problem of vanishing gradients gets counteracted using recurring skip connections between hidden layers which maintain a steady gradient flow. Here each subsequent hidden layer results in a residual of the prior layer: $\boldsymbol{h}^{(i+1)} = f(\boldsymbol{h}^{(i)}) + \boldsymbol{h}^{(i)}$ (see figure 2.5). This idea has been proven to be very efficient in suppressing vanishing gradients and therefore allow arbitrary deep NN architectures [16].

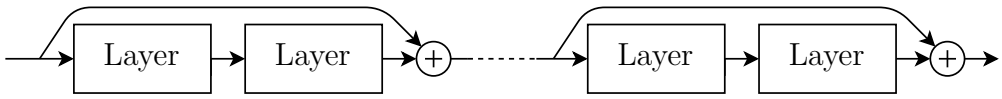


Figure 2.5: A residual block.

Parameter Initialization

The choice of initial parameters $\boldsymbol{\theta}$ before starting the training can also be critical for the optimization. Common random parameter initialization can be interpreted as a randomly chosen starting point within the solution space of the model's cost function. This random initialization breaks the symmetry and enforces different parameter updates for each unit within a network. Commonly, a scaled Gaussian or a uniform distribution is used for the initialization [14, ch. 8.4]. For different activation functions and numbers of hidden units in a layer, optimal scaling parameters can

be computed [12]. The use of such proper initialization was an important step in making deep NN trainable.

Weight Normalization

To compensate bad initialization of weights multiple self-normalizing layers have been proposed, e.g., [24, 18]. Weight normalization [24] decouples the length of the weight vectors from their direction. The weight vector $\boldsymbol{w}$ gets reparameterized with $\boldsymbol{v}$ and a scalar g

$$\boldsymbol{w} = \frac{g}{||\boldsymbol{v}||_2}\boldsymbol{v} \tag{2.32}$$

with $||\boldsymbol{w}||_2 = g$. SGD is then applied with respect to those parameters instead. Doing so improves the conditioning of the gradients and leads to improved convergence of the optimization procedure. Furthermore, NN with weight normalization work well with a much wider range of learning rates given an intial choice of parameters.

2.1.4 Regularization

While NN are powerful approximators, they can overfit to small data sets easily. So even if we can optimize them on our training set, careful regularization techniques are required to make them generalize to unseen data [14, ch. 7]. Common regularization tricks involve *early-stopping*, *choice of inductive bias*, *dropout layers* 2.10, and *data augmentation* 2.1.5.

Early-Stopping

The idea of early stopping is to recognize if a model stagnates or starts overfitting during training and to consider the training to be done as a consequence. Along the way, the best performing model state will always be saved according to a metric coming from the evaluation. This metric is generally the cost of the validation dataset. A common criterion to end the training is when the model does not improve upon the metric for some predefined number of epochs in a row. Once the criterion was met, to end the training the saved model state is taken as the outcome of the training procedure, rather than the most recent model parameters.

Early stopping is a popular but simple regularization technique with great effect. It does not modify the model parameters directly but decides after how many training iterations to settle on a sweet spot. Due to how early stopping is driven by the evaluation dataset it has an indirect effect on the parameterization of the final selected model. For this reason, a third independent test dataset has to be utilized to receive a truly unopinionated performance measure of the model.

To improve performance further the iteration count determined via early stopping could be exploited to recommence a new model training using the iteration count as a hard stop but this time incorporating all available datasets for parameter updates. This can be reasonable if the amount of available training data is quite limited as long as the first experiment confirmed the model to be functional [14, ch. 7.8].

Choice of inductive bias

If we have an understanding of how our data is structured we can restrict our space of all neural networks to those which represent this structure. This restriction concerning a domain problem is called an *inductive bias*. For audio signals as analyzed in beat-recognition tasks, a sensible inductive bias is given by CNNs. They are built from convolution layers and pooling layers.

Convolution Layer The common convolution used in ML [14, ch. 9] is very similar to the convolution in digital signal processing (DSP) also applying a filter kernel on sequential data like audio. In contrast to the common convolution known in DSP the convolution in ML has a flipped kernel which makes the operation a cross-correlation. Although talking about convolution in ML refers to applying the cross-correlation operation on the whole input in parallel by sliding over it. This allows different sized input data making the convolution very flexible on what data it works with compared to a fully connected layer configuration.

The connections to a neuron within a convolutional layer are locked by the size of the filter kernel

$$(\boldsymbol{x} * \boldsymbol{k})_m = \sum_i x_{m+i} k_i. \tag{2.33}$$

In contrast to a fully connected layer configuration a great performance improvement is achieved by omitting most operations. An exemplary convolution is visualized in figure 2.6. With m output neurons and n input neurons, the sparse connectivity of having a kernel size smaller than the input $k \ll n$ results in a runtime of $O(k \times m)$. Sliding over the input with the same kernel allows for parameter sharing over the whole input which enforces generalizing behavior and reduces the memory footprint.

The properties of a convolution have a regularizing impact on a model. Especially the reduction of the parameter count via parameter sharing emphasizes exploiting exploiting the structure of information inherent to the data. Overall this reduces the search space drastically compared to a fully connected version [14, ch. 9.2].

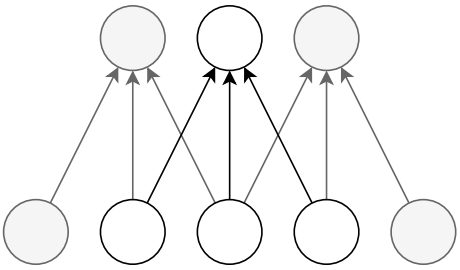


Figure 2.6: Convolutional layer with a kernel size of 3. With 5 neurons in the input layer the resulting output has 3 neurons.

Pooling A pooling function summarizes nearby features usually with the purpose to reduce the size of its input. This exploits the assumption that the local information of its input is likely to be redundant, thus compressing the information is comprehensible for achieving a reduced search space. There are different summarizing operations to choose from of which the most common are the maximum and the average function. Within a rectangular neighborhood, the maximum function selects the maximum while the average function computes its L^2 norm. Likewise to the convolution operation they use a kernel to compute their statistical summarization. Thus, the convolution can be viewed as a pooling operation with trainable parameters of its filter kernel. The operations of the maximum and average pooling functions are predefined and have no trainable parameters, thus also no impact on the computational cost of the backward pass [14, ch. 9.3]. Both convolution layers and pooling, are determined by padding, stride and dilation.

Padding For a convolutional layer, the size of its output is depending on the size of the filter kernel and the input layer. With an input size of n and a kernel size of k the output m will be shrunk by $m = n - k + 1$ as seen in fig 2.6. Stacking many convolutional layers like this on top of each other could shrink an input down to 1 output in which case the last layer would be fully connected again. To prevent behavior like this padding the input decouples the output size from the kernel size. Padding in general refers almost always to pad a sequence with zeros. Besides that, alternative padding patterns can be used like mirroring at the edges or repeating the outmost values of a sequence (see table 2.1). Additionally, the value count to pad with can be different on each side of the sequence.

A typical use case for padding is to maintain the same size on the output as on the input by padding at least $k - 1$ values. That makes building arbitrarily deep CNNs possible [14, ch. 9.5]. See figure 2.7 for an illustration supplementing the layer configuration of figure 2.6 with padding.

Table 2.1: Different padding patterns for a given sequence $\boldsymbol{S}$ with 4 elements and a padding of 2.

Sequence	Zero Padding	Repeat Padding	Mirror Padding
$\boldsymbol{S} = [a, b, c, d, e, f]$	$[0, 0] \oplus \boldsymbol{S} \oplus [0, 0]$	$[a, a] \oplus \boldsymbol{S} \oplus [f, f]$	$[c, b] \oplus \boldsymbol{S} \oplus [e, d]$

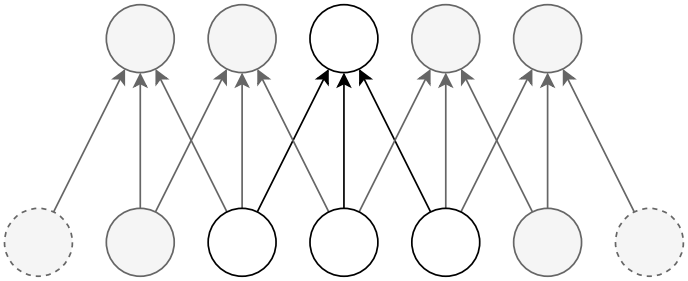


Figure 2.7: Convolutional layer with a kernel size of 3 and a padding of 2. The padded values (dotted circles) on both borders of the bottom input layer allow the convolution to maintain the same output size as the input.

Stride The convolution or pooling operation allows specifying a distance between the beginnings of all kernels covering the input sequence. This distance is formally known as stride and is measured in input data points. With a stride value of 1, the beginning of all kernels is placed on each data point of the input sequence as long as enough data points are available to be fully covered by the kernel. This is the default configuration. The same goes for a stride of 2, 3, etc. but skipping every second, third, etc. input data point before applying the filter. Having a stride smaller or equal to the kernel size guarantees that no input data point is skipped entirely. Stacking multiple strided layers without padding reduces an input sequence by a factor of the product of all stride values. Each layer emphasizes a specific time resolution of the original input sequence. A layer configuration utilizing stride is visualized in figure 2.8. Since stride simply omits data its biggest advantage is the reduced computational cost and memory requirements for all subsequent layers due to the reduction of data points [14, ch. 9.3].

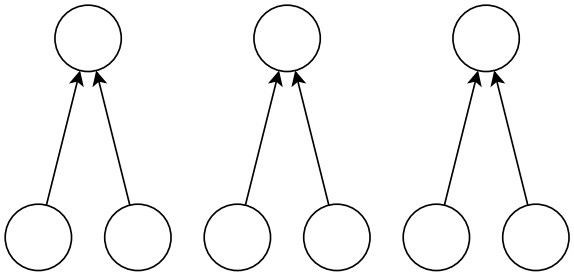


Figure 2.8: A pooling layer with equally sized kernel and stride of 2 is down sampling the input by a factor of 2.

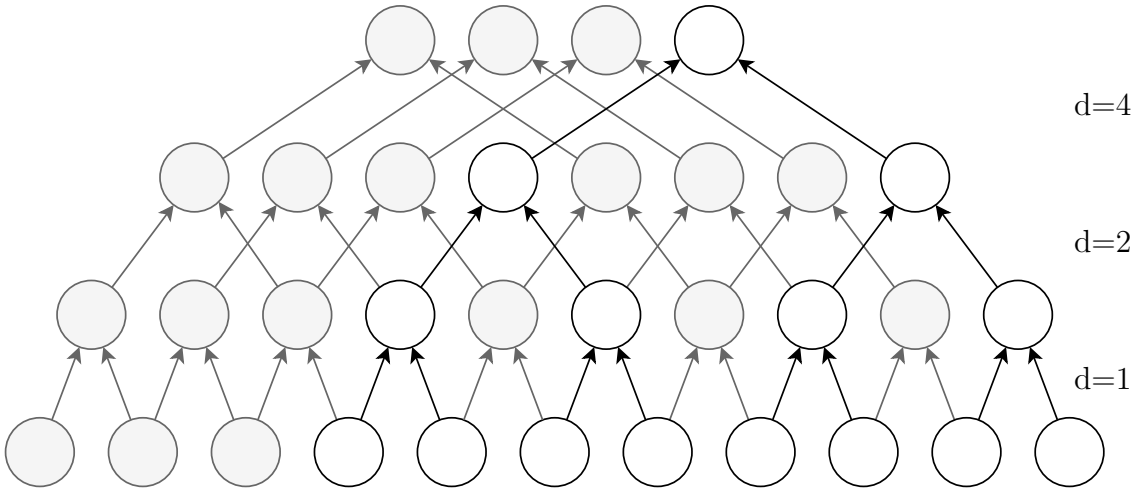


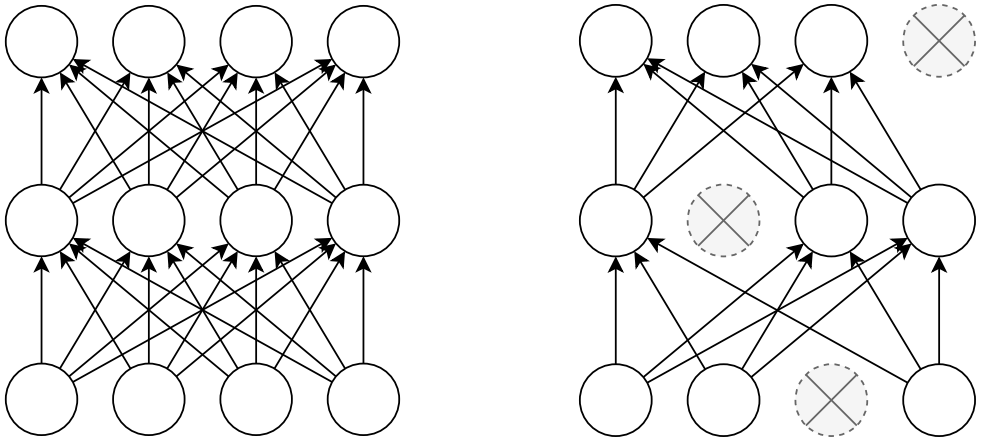
Figure 2.9: Dilations of 1, 2 and 4 with a fixed kernel size of 2. One output represents a receptive field of 8 inputs.

Dilation Dilation defines an equally spaced spread of the individual kernel points over its input. More specifically the uniform dilation d of the kernel points is omitting $d-1$ input data points in between. The vanilla convolution utilizes a dilation of 1 resulting in a filter kernel without a gap. A benefit of dilation is to change the receptive field without increasing the parameter count of the filter kernel itself. But this happens at the cost of regularly skipping in-between values. Nevertheless, applying an increasing dilation on multiple stacked convolutional layers allows the output to still be built upon the input sequence without gaps. With i being the ith layer and a dilation value $d_i \leq k^i$ it is guaranteed that the overall receptive field has no gaps. Figure 2.9 shows this using an example of 3 stacked convolutional layers with a kernel size of 2. Similar to the stride parameter each layer emphasizes a specific time resolution depending on the corresponding dilation value. An exponential growth of the receptive field is achieved with a linear growth of parameters which makes dilated convolution very efficient for long sequence modeling [1].

Dropout

Dropout is another very popular regularization technique. In each training iteration, individual neurons of a network are ignored with a probability p. Thus, only a randomly chosen subset of the original network is updated in this step (see figure 2.10). This encourages the network to become less sensitive to specific weights and thus leads to more robust feature learning. During test time, the whole network is used again, but now all weights are down-scaled by $1-p$. This can be interpreted as averaging over a random ensemble of many models at once. While dropout will slow down convergence due to the additional noise, it is much faster compared to

training multiple independent random models in parallel [26].



(a) A normal neural network (b) After applying dropout

Figure 2.10: After applying dropout on a normal network (left) a randomly chosen subset of that network (right) remains which still should be able to adequately handle its input.

2.1.5 Data Augmentation

Finally, more training data from the same source will help generalization. Unfortunately, in practice the amount of training data is limited. One way to circumvent this limitation is to extend an existing data set with variations of its data samples. There are numerous ways to create variations like translation, rotation, scaling or noise injection. It is important to make sure that those adjustments are not changing the meaning defined by the label and that they are common variations of the data [14, ch. 7.4].

2.2 Implementation

The problem of recognizing beats in audio is modeled as a binary classification problem utilizing a NN. Therefore the training data is divided into two classes; no-beats and beats. Section 2.2.2 describes the preparation of the data in more detail.

The overall network architecture uses stacked CNN-blocks capable of capturing a suitable temporal context similar to the work of Oord et al. [22]. The temporal context increases gradually with each layer utilizing the stride and dilation parameters of the convolutions. This temporal context is further referred to as the receptive field of the NN. A binary cross-entropy loss in conjunction with a sigmoid output layer determines the performance of the model. Section 2.2.3 describes the overall network architecture in more detail.

Due to the strided convolutions of the network, it will have a reduced time resolution or frame rate on the output compared to the input sequence. As a consequence, each output feature of the NN is corresponding to a block or frame of several input samples. This downsampling process prepares the data for a final refinement by a DBN as described in section 2.2.4.

An overview of the whole network architecture and how all parts are interconnected can be seen in figure 2.14. The signal conditioning and various block configurations are summarized in table 2.2.

2.2.1 Pre-Processing

In the research field of beat tracking, it is common practice to convert the audio from the time domain into the frequency domain as a first step before engaging in further signal processing. This thesis skips this pre-processing stage entirely and follows an approach to solve the problem working directly on the raw audio itself. For this reason, the data used for training the NN is a raw pulse code modulated (PCM) audio signal. The DJ software *Traktor Pro* from NI is using a sample rate of 22050 Hz for its multitude of MIR analyses. Following this guidance and to create a common ground as well as reducing the impact of high-resolution audio on the computational complexity upfront, the audio going into the NN is restricted to have 22050 Hz as well. In addition to that, each audio file will be mixed down into a single mono channel. The audio material across all datasets is pre-processed as described in this section.

2.2.2 Data Preparation

A dataset used for beat detection consists of a collection of audio files with an accompanying text file holding the beat annotations. Each beat annotation consists of a timestamp in seconds and a type of beat. For this thesis all beat kinds are equally relevant, thus their differentiation can be ignored. An exemplary excerpt of a dataset is visualized in figure 2.11

Labels

The beat annotations are holding the information for conditioning the network. They are divided into 2 classes; no-beats and beats which represent the target the model is trained to predict. All classes are considered to be exclusive as preconditioned by the binary cross-entropy. Since the annotations are only holding information about beats, the no-beat information has to be deduced indirectly. The timestamps of the classes are put into the representation of the audio material, e.g.

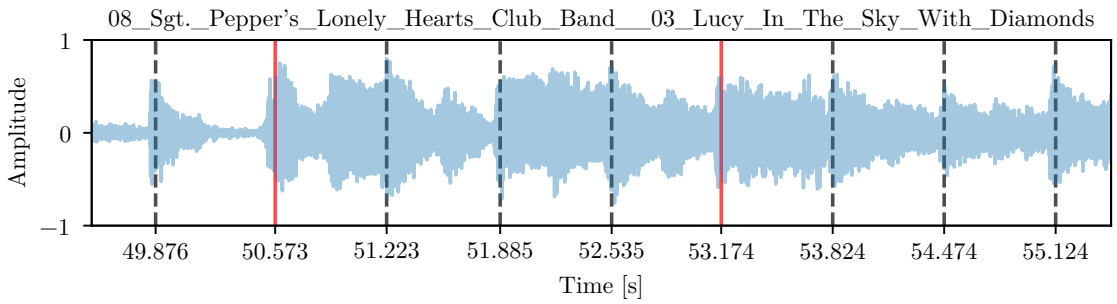


Figure 2.11: One exemplary item taken from the Beatles dataset including vertical beat annotations with timestamps. Solid lines are downbeats and dashed lines are normal beats representing the different beat kinds.

samples, to have a common ground. It is useful to interpret the input sequence as being grouped into equally sized blocks or frames due to the downsampling property of the NN. All audio surrounding the annotation timestamps within such a frame is classified as a beat. All remaining audio is classified as no-beat from which a no-beat timestamp can be randomly chosen.

Audio Chunks

During training, the proposed model will be fed with equally sized chunks of audio capturing at least the period defined by the receptive field of the NN. These audio chunks are directly taken from the audio files and are centered around the beat or no-beat timestamps. It is important to note that the receptive field associated with a no-beat class could indeed overlap the receptive field associated with a beat class. But this does not apply to the much smaller frame size due to the reduction of the time resolution of the network. The center of the extracted audio chunks and their vicinity are not allowed to overlap within this frame size. This is necessary to make a clear distinction between all beat classes during training. While respecting this restriction a no-beat chunk can just be randomly sampled from the whole track, which already applies data augmentation for the no-beat events.

If a beat event is at the beginning or end of the audio material there is a chance that not enough audio samples are available to fill up a whole chunk since the termini of the corresponding audio file would be surpassed. In such situations, the missing content is compensated by zero-padding the audio material accordingly.

Dataset Balance

At a sample rate of 22050 Hz, the amount of as no-beat classified audio is much higher compared to the audio classified as a beat. Training a network with an unequal distribution of annotations could encourage a bias towards the more frequent

ones. The overall number of beat events is fixed according to the available annotation data within a dataset. Since all no-beat events can be randomly sampled from the audio material their share in the dataset can be tailored to complement the beat events by matching the sum of all beat annotations. This leads to a well-defined distribution of no-beat events across all audio files.

To expand the temporal region of beats *label smoothing* is applied similar to the work of Davies et al. [8]. Both neighboring frames of a beat class will be considered a beat as well but weighted half as much. This doubles the impact of beat labels, thus allowing the no-beat class to be sampled twice as much.

Augmentation

To improve the ability to generalize in the context of a limited amount of training data it is useful to expand the search space of the NN model by augmenting the available data. While doing so it is important to maintain the underlying meaning encoded by the labels. During training of the proposed model, 3 kinds of data augmentation techniques are applied to the audio material of each loaded chunk.

Translation Since the model downsamples its input an extended frame in which the beat annotations uphold their meaning exists. For this reason, the timestamp of each beat can randomly be translated back and forth by half the frame size. The translated timestamp will then be the final center for extracting an audio chunk. In case of no-beat labels this kind of data augmentation is already applied at by randomly sampling the whole audio material as described in section Labels.

Attenuation Each audio chunk gets attenuated to increase the variety of dynamics. Therefore a randomly chosen attenuation factor within a range of 6db is applied when loading the data. It is important to note that this attenuation is affecting an audio chunk as a whole so its dynamic range is left untouched.

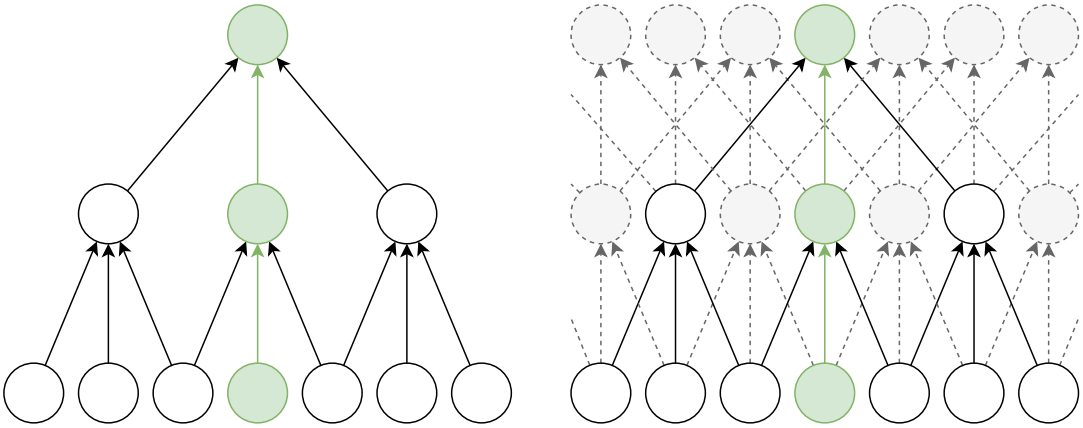
Inversion With a probability of 50%, a whole audio chunk gets multiplied by -1 to invert the signal. Given that the network is acting on raw audio material this helps to improve generalization over phase information.

2.2.3 Network Architecture

This section presents the architectural decisions and components on which the proposed network model is based.

Non Causality

The problem of beat detection is tackled in an offline fashion. This allows the network to take future information into account which seems reasonable in the context of the sequential nature of audio and its accompanying beat events. To maintain a balance between the future, present and past the model is trained on uniform chunks of audio having their corresponding beat or no-beat event in the center (see section 2.2.2). Moreover, the kernel size of all layers is set to an odd number allowing one straight connection through the center of all kernels maintaining the spatial relationship to the input. Such a configuration is visualized in figure 2.12. This builds a well-defined focus on the central features of each chunk as well as their surroundings.



(a) Strided configuration (b) Dilated configuration

Figure 2.12: A stack of two convolutional layers with an odd kernels size of 3 in a strided and dilated configuration. The straight spatial relationship of the centered features is highlighted in green.

Padding

Information coming from the audio material is more valuable for decision making than taking arbitrary padded values into account. For this reason, all layers within the proposed network abstain from using padding. As a consequence, the incoming sequence shrinks while passing through the network. With this behavior in mind, enough input material has to be provided upfront to achieve the desired output size.

As an example, the input sequence has to be as long as the receptive field to get one output feature. Based on that, adding samples as many as the downsampling frame size to the sequence gives an additional output. To get a total of 3 output features one has to provide an input sequence with the length of the receptive field plus two times the frame size.

Network Width

The very first block of the network will expand from a one-dimensional raw mono audio input to a hidden width of 16 channels. The width of the skip-connections is configured the same way. Once expanded, the network width will remain constant until reaching the output layer.

Building Blocks

The architecture of the proposed model consists of three principal constituents which are described in more detail within this section. Its core design is highly inspired by the work of Oord et al. [22] with their concept of dilated convolutions as well as Böck et al. [3] and their adaption to use the *ELU* activation function instead of a gated *tanh*. All trainable parameters within the network are initialized using Gaussian distribution and are configured to use weight normalization as proposed by [1].

Striding Front This is the entry point of the network. It will be fed with raw audio at a sample rate of 22050 Hz to be transformed into a beat activation with a framerate of $\approx$ 115 frames per second (FPS). Therefore the input needs to be downsampled within the network, giving it the possibility to optimize upon this process. The striding front of the network architecture is taking care of this using the stride parameter on convolutional and average pooling layers.

A striding block is following the principle of a residual block. It doubles its input into two parallel branches, a trainable convolutional layer and a non-trainable average pooling layer passing the input minimally invasive. The stride of both layers defines their downsampling factor and is set equally. An *ELU* activation function right after the convolutional layer introduces non-linearities. It will be followed by a dropout layer to regularize the network. In the end, both parallel branches are added together to have one unified output again.

In total 3 striding blocks are stacked on each other until the desired framerate is reached. Since all striding blocks are placed in the front of the network the whole architecture benefits from an early reduction of parameters. This is a crucial part of the network to lower its computational cost facing raw audio signals on a reasonably high sampling rate. Besides downsampling, this stage of the network transforms the input into a unified intermediate feature representation tailored for the following parts of the network.

Dilation Core After the striding front, the desired frame rate of the postprocessing DBN is reached. This has increased the intermediate receptive field of which

each feature accounts information for 192 input features or $r \approx 8.7$ ms. A timeframe of this size seems too small to understand the temporal nuances of a beat within the audio material. Hence the receptive field has to increase further but this time utilizing dilated convolutional layers.

Similar to the striding block a dilation block also follows the principle of a residual block utilizing two parallel branches. One simply passes the input unaltered and the other applying a dilated convolution. The dilated convolution is followed by an *ELU* activation function as well as a dropout layer to regularize the network. Subsequently, a 1x1 convolution layer provides skip connections to the output layer. They accumulate and build the foundation of the network's final output. A portion of them is also added back to the unaltered input, carrying over relevant information to the upcoming blocks in a common information stream.

Altogether 8 dilation blocks with an exponential growing dilation parameter with a base of 2 are stacked on top of each other. Along the way, all skip connections are accumulated for further processing in the upcoming part of the network. The common information stream gets terminated after the very last dilation block.

Consolidating Back The back of the network transforms all accumulated skip connections into the desired output shape of the network. All skip connections are concatenated before going into two sequences of an *ELU* activation function, a dropout and a 1x1 convolutional layer. The first convolution mixes the concatenated channels down to the hidden width of the network and the second convolution transforms from the hidden width to the desired output channel count.

A 1x1 convolutional layer can be interpreted as a dense layer over channels capable of scaling with the length of the input sequence fed into the network. This gives the model the ability to handle audio of arbitrary length if it is at least as long as the overall receptive field demands. Since the network is designed for solving a classification problem a *sigmoid* activation function finalizes the network's output.

2.2.4 Post-Processing

Music and its temporal structure generally follow a more or less regular pattern based on tempo and meter. Taking this property for granted, it can be exploited to infer the most likely sequence of repetitive beat events given the rhythm activity of a whole musical piece. Therefore a DBN will be used as a post-processing stage as seen in [5] taking the results of the NN as an observation of beat activation to extract the most likely sequence of beat timestamps. For convenience the DBN implementation within the *madmom* library [6] will be used.

A DBN is an advanced hidden markov model capable of incorporating the tem-

poral structure within music by modeling probabilities over a variety of tempo and meter configurations. Therefore, an initial state distribution and a transition model define a probabilistic model of within-bar position and tempo. It is based on a bar pointer model associating a velocity to a bar pointer simulating the steady continuity of music following a tempo. More detail on hidden markov models and the bar pointer approach can be taken from the original work of Whiteley et al. [28]. Given an external probability distribution as observation of beat activity a directed acyclic graph is constructed using the probabilistic model. The graph is the hidden state space of a DBN discretized into as many fixed steps as needed to cover the given observation. The underlying bar pointer model allows to continuously advance even when facing a low activity period within the given observation. The Viterbi algorithm is efficient in finding the most likely path within the state-space due to its strategy of dynamically rating paths and excluding unlikely ones as early as possible. Using the most likely path of the state space of the DBN, the observation's most likely sequence of beat events can be predicted.

The work of Krebs, Bock, and Widmer [20] improves upon the original bar pointer model by introducing a more efficient probabilistic model for their DBN. They make the number of discrete bar positions dependent on the tempo to distribute them evenly over the whole tempo range and increase the time resolution especially for music with a higher tempo. Additionally, tempo transitions are only allowed on beat positions but therefore to all available tempo states instead of the nearest three as in the original model. All these changes reduce the computational complexity and memory footprint significantly. For the initial state, no prior knowledge of the data is incorporated and thus a uniform distribution used. To impede excessive tempo state changes the exponential transition function of two hidden tempo variables $\dot{\Phi}_k$ and $\dot{\Phi}_{k-1}$

$$f(\dot{\Phi}_k, \dot{\Phi}_{k-1}) = exp\left(-\lambda \times \left|\frac{\dot{\Phi}_k}{\dot{\Phi}_{k-1}} - 1\right|\right) \tag{2.34}$$

models the likeliness of a tempo change. Its λ parameter scales the tempo transition probability with a value of $\lambda \in [1, 300]$ referring roughly to constant tempo and a value of zero makes transitions to all tempi equally probable.

2.3 Roundup

This section aggregates all relevant information from the previous sections of this chapter to give a centralized overview of the proposed model architecture and how it is trained. Figure 2.14 shows the individual parts of the model and how they are

connected while table 2.2 provides detailed information on the model inside. Figure 2.13 shows the whole signal chain from an audio file to the final beat timestamps.

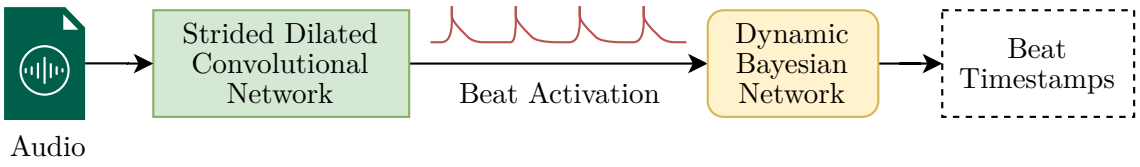


Figure 2.13: The signal chain of the whole beat-tracking system.

[1]This FPS correspond to a sample rate of 22050 Hz for the input sequence.

Table 2.2: Overview of the configuration of the various network components

Downsampling Front	
Channels	1, 16, 16
Kernel Size	33, 17, 9
Stride	8, 6, 4
Activation Function	*ELU*
Pooling Function	*Average Pooling*
Dropout	0.2
Dilation Core	
Channels	16
Skip Channels	16
Kernel Size	5
Dilation	2^0, ..., 2^7
Activation Function	*ELU*
Dropout	0.2
Consolidating Back	
Channels	$16 + 16$, 16
Output Channels	1
Output Activation Function	*Sigmoid*
Activation Function	*ELU*
Dropout	0.2
Model Summary	
Model Parameters	22978
Frame Size	192 samples ($\approx$ 8.7 ms)
Receptive Field	196385 samples ($\approx$ 8.9 s)
Output Frame Rate	$\approx$ 115 FPS[1]

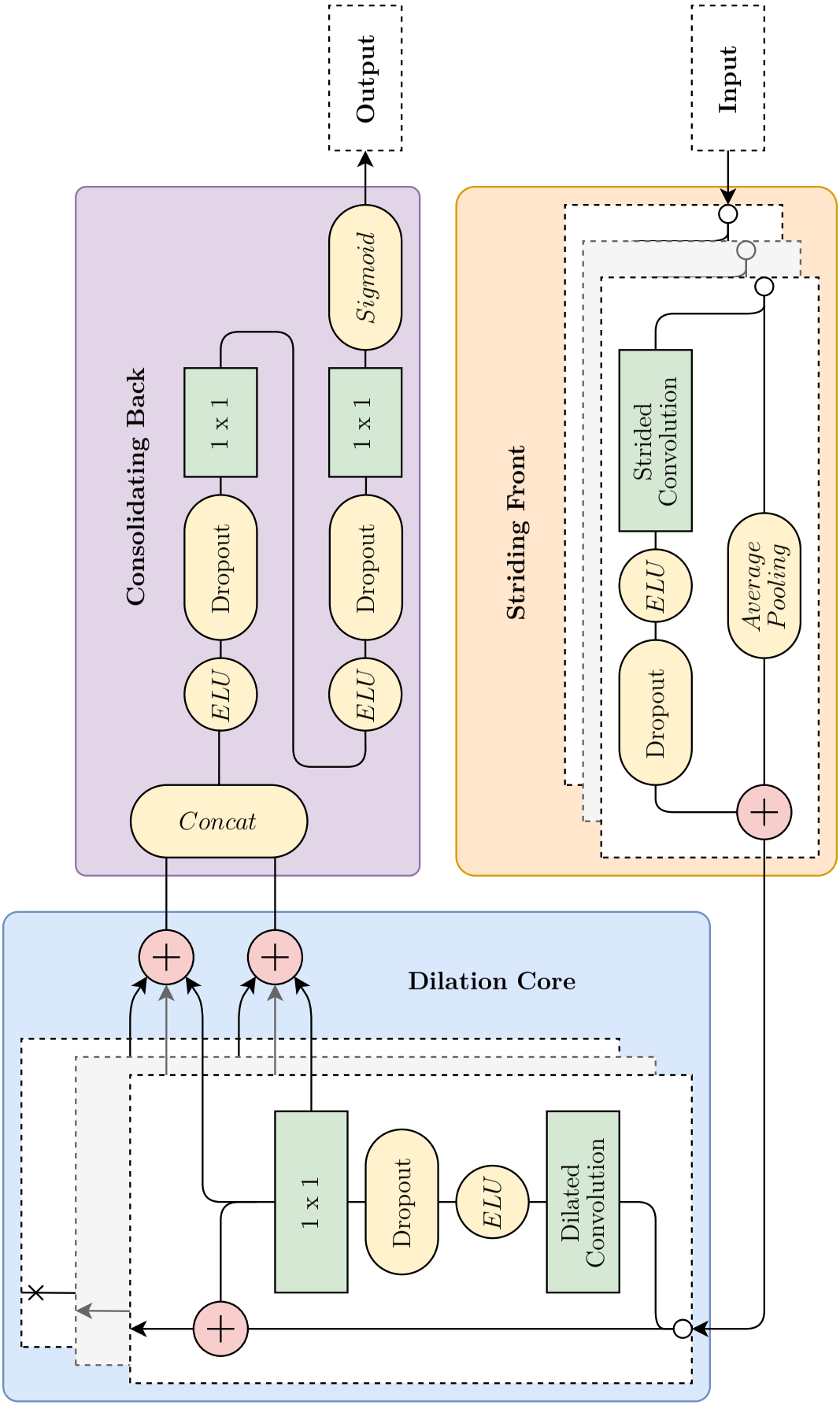


Figure 2.14: The various building blocks forming the network architecture.

3 Evaluation

The evaluation of a beat detection system is a crucial part of measuring its performance in context of related research. This chapter introduces the datasets used for training, evaluation and testing, the metrics common for beat detection and the baseline to compare against as well as the results thereof.

3.1 Datasets

A beat-tracking dataset consists of audio files accompanied by beat annotations. This section presents several datasets and why they are used in this thesis. An overview of all datasets and their content is shown in table 3.1. An excerpt from the Beatles dataset can be seen in figure 2.11.

Table 3.1: Overview of the datasets.

Dataset	Files #	Beats #	Length
TapCorrect[10]	89	36078	6h 09m
Ballroom[15, 21]	698	44604	6h 04m
Beatles[9]	179	52729	8h 00m
Maschine_{train}	224	40053	6h 17m
Maschine_{test}	223	40902	6h 18m
Maschine $\sum$	447	80955	12h 36m

3.1.1 TapCorrect

The TapCorrect dataset [10] consists of beat annotations for a variety of established popular music. It is comprised of a variety of genres from various recording conditions and instrumentations. The whole audio corpus can be found on the *YouTube* video platform. The researchers put much effort into detecting annotation errors as well as refining their annotations which results in a high-quality dataset. It includes

music with varying tempo (see figure 3.1c). Due to its quality it is suited to be used as a training dataset for the proposed NN architecture. Nevertheless, during the inspection of the dataset, some further adjustments had to be made. One file has been split in two and some audio files were unfortunately not available. Thus a subset of the original TapCorrect dataset is used in this thesis consisting of 89 audio files with a total playback length of 6 hours and 9 minutes and has 40902 beat annotations.

3.1.2 Beatles

The Beatles dataset [9] contains 12 studio albums of the band *The Beatles*. It has 179 audio files with a playback length of 8 hours and overall 52729 beat annotations. It is a great example of music with varying tempo as seen in 3.1b. This dataset is used as a test dataset to neutrally measure the actual performance of the proposed network architecture.

3.1.3 Ballroom

The ballroom dataset [15, 21] is an established dataset for beat tracking. It consists of ballroom dancing music across various genres. The genres comprise Cha Cha, Jive, Quickstep, Rumba, Samba, Tango, Viennese Waltz and Slow Waltz. Each file is an excerpt of a song and has an average length of 30 seconds sampled at 11.025 kHz. It consists of 698 files with an overall playback time of 6 hours and 4 minutes and a total of 44604 beat annotations. This dataset is used as an evaluation dataset in the training of the proposed network.

3.1.4 Maschine

This dataset was created as a research goal of this thesis and to be used for training and testing of the proposed NN architecture having varying tempo in mind. It is based on the content of ME[1] coming from the *Maschine* ecosystem of NI. A ME is a collection of synthesizer presets, audio samples and loops tailored to the needs of producing music for a variety of popular music genres. For example EDM, Electronica, Experimental, Funk, Future Bass, Global, Hip Hop, House, Indie, Pop, R&B, Soul and Techno. Each ME includes several projects of demo songs showcasing the bought product. These demo songs are the foundation of the Maschine dataset.

All demo songs are reexported from the *Maschine* software with an automation modifying the playback speed. This automation is applied during the export of the audio via a *tempo factor* multiplied by the base beats per minute (BPM) of the song.

[1] `https://www.native-instruments.com/de/catalog/maschine/expansions/`

This makes the exported songs vary in tempo as an alternative approach to post-processing technics like resampling or time-stretching. While exporting the audio, all occurring beat events including their timestamps are tracked to be written to an annotation file once the export is finished (see algorithm 1).

Unfortunately, all Maschine projects are set up with a fixed BPM value over the whole song and automating the tempo is currently not an available feature. This feature had to be implemented as well as the mechanism of extracting the beat information of the song during the exporting process. Additionally, the process of exporting all projects was automated by implementing a robotic process automation (RPA) script. It is based on the *robot framework*[2] communicating with the testing interface of the Maschine software. Due to the implementation efforts, the dataset creation was mostly automated and therefore less time-consuming compared to a manual annotation approach. The final dataset consists of 447 audio files with a total playback length of 12 hours and 36 minutes and has a total of 80955 beat annotations. Since the annotations are generated in the export process itself, they are very consistent and accurate compared to handmade ones. Having high quality annotations could benefit the overall performance of a beat detection system.

A simplified version of the whole process of exporting the audio files with an altered tempo sequence and the accompanying beat annotations is shown in algorithm 1 in pseudo-code. For each beat within a song, a random tempo factor is set which manipulates the playback speed during the export. The beats within a song are iterated in ascending order from the first to the last. The *random()* expression draws a random sample from a Gaussian noise distribution within the interval $(-1, 1)$ which results in a random tempo increase or decrease of a maximum of 2% going from beat to beat.

Algorithm 1: Routine exporting the Maschine projects

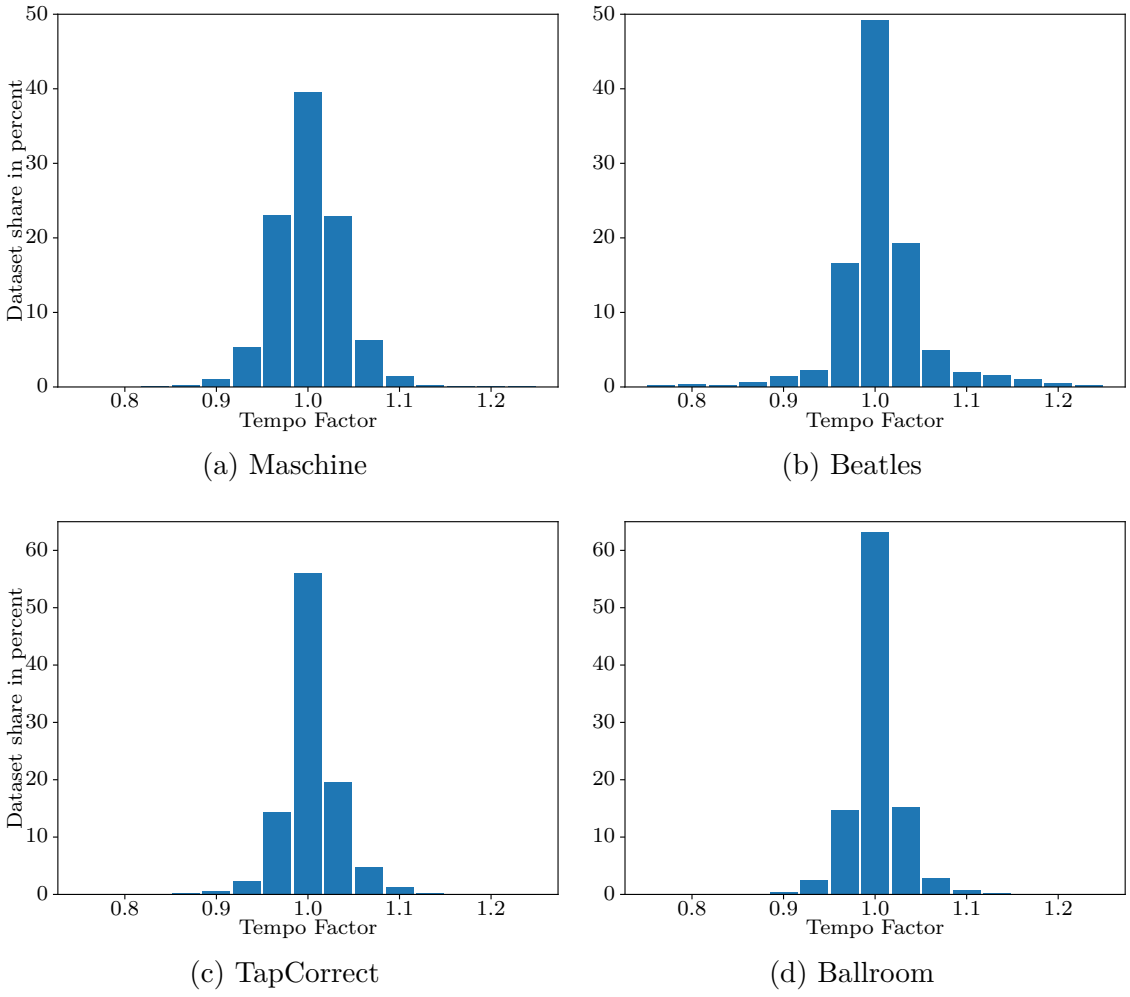
```
Data: projects
foreach p in projects do
    song = p.loadSong();
    exp = 0.0;
    foreach beat in song do
        exp += random() * 0.02;
        song.tempoFactorSequence[beat] = 2^exp;
    end
    p.exportSongAsWav();
    p.exportAnnotations();
end
```

The chosen rate of tempo change is inspired by the findings of Dannenberg et al.

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

[7] who analyzed the tempo changes in live performances of amateurs and professionals. The Maschine dataset takes the tempo variations of amateur performances as a starting point but goes further allowing even higher tempo changes. Since the dataset is used for training, this emphasizes the aspect of varying tempo in music as a research topic of this thesis. The available tempo variations within the dataset can be measured by looking at the *beat-interval*, the period of two ascending beats. For each song, a *tempo factor* sequence $\boldsymbol{s_{tf}}$ is computed based on the songs beat-interval sequence $\boldsymbol{s_{bi}}$ and its mean as a reference

$$\boldsymbol{s_{tf}} = \frac{mean(\boldsymbol{s_{bi}})}{\boldsymbol{s_{bi}}}. \tag{3.1}$$



(a) Maschine (b) Beatles

(c) TapCorrect (d) Ballroom

Figure 3.1: The distribution of tempo factors across all datasets.

Figure 3.1a shows the distribution of the concatenation of all tempo factor sequences within the Maschine dataset represented by a histogram across 15 bins within the range $[0.75, 1.25]$. The tempo factor distribution of all other datasets is shown in figure 3.1 as a reference. Comparing the distributions, the focus on varying tempo of the Maschine dataset is visible by the broader share of the side bins within

the dataset while staying in similar bounds.

3.2 Beat Evaluation Metrics

For the evaluation of a classification problem, it is useful to categorize all predictions into four groups concerning their associated target as seen in table 3.2.

Table 3.2: Confusion matrix for classification problems.

	Positive Target	Negative Target
Positive Prediction	True Positive	False Negative
Negative Prediction	False Positive	True Negative

True positives are all predictions that are correctly classified as positive. *True negatives* are all predictions that are correctly classified as negative. Together they are representing all correctly classified predictions. *False positives* are all predictions that are falsely classified as positive and *false negatives* are all predictions that are falsely classified as negative. Together they are representing all falsely classified predictions.

3.2.1 F-Measure

This metric summarizes the four classification groups seen in table 3.2 and provides a more general score of a classifier's performance. It is based on two intermediate metrics known as *precision* p and *recall* r. They describe the fraction of all correctly positive classified predictions among all positive targets and among all positive predictions respectively

$$p = \frac{tp}{tp + fp}, \qquad r = \frac{tp}{tp + fn}. \tag{3.2}$$

The f-measure F_1 is computed from their balance as follows

$$F_1 = \frac{2\ p\ r}{p + r} = \frac{2tp}{2tp + fp + fn} \tag{3.3}$$

and is known to be a reliable metric even in the context of an imbalanced dataset. In contrast, the very popular *accuracy* metric is a more straightforward measure of a classifier's performance. It is built upon the fraction between all correct predictions among the total number of examined cases. The problem with accuracy is that it does not respect the balance of all classes. On a highly imbalanced dataset, good accuracy could be achieved by always predicting the most frequent class which would not be a useful classifier.

In the context of beat tracking evaluation, a tolerance window around the target timestamps allows predicted beat timestamps to be considered as a true positive as long as they are within this window. For comparison to related research, the default tolerance window of ±70 ms around a target timestamp is used for the final evaluation.

3.2.2 Continuity-Based Scores

The correct metrical level (CML) scores are measuring how many beats have been predicted correctly in a row according to the beat-intervals of the annotations. Therefore two consecutive predicted beats have to match the corresponding annotation beat-interval within a tolerance window of ±17.5%. The CML_c is only accounting for the longest consecutive period while CML_t takes all consecutive periods into account. Both allowed metrical level (AML) scores, AML_c and AML_t are working the same respectively but consider the predicted beats also to be correct if they are offbeat or at half, double or triple beat-interval tempi. More information about the continuity-based scores is available in [9].

3.2.3 Information Gain

This metric measures how much information about the annotations is describable with the predictions. The higher the information gain the better the prediction. It is built upon a probability distribution of *beat-errors*. A beat-error is the distance between the annotation and the prediction within a one-beat window around the annotation. The information gain is the Kullback–Leibler divergence (KLD) between such a probability distribution of a beat-error sequence and a uniform probability distribution as a worst-case reference. In the context of beat tracking it is common to estimate the beat-error probability distribution with a histogram composed of 40 bins allowing a maximal information gain of $D_{max} = log_2(40) \approx 5.3$ bits. More details about this metric can be found in [9].

3.3 Baseline

For the evaluation of the proposed model, it is being compared to two baseline models [4, 5] with state of the art performance. Both provide an implementation within the madmom library [6]. They utilize a DBN as a post-processing stage for extracting the final beat timestamps. In this thesis the efficient implementation [20] is used for all models as a post-processing stage. Both baseline beat-processors have to be fed audio at a sample rate of 44.1 kHz. To achieve this requirement the audio

material from the datasets has to be sampled up for the evaluation, doubling the sample rate.

The first baseline model [4] uses a multi-model approach to detect beats which is in turn based on the approach of [2]. From a collection of specialized models for certain musical styles, it chooses the model with the most appropriate beat activation function.

The second baseline model [5] is trained to jointly detect beats and downbeats. To make it comparable to the model proposed by this thesis, both outputs (beats and downbeats) are added forming a unified beat prediction.

Both baseline models are built upon an advanced recurrent neural network (RNN) architecture; a BLSTM network. A RNN allows feedback connections that extend the underlying computational graph to have cycles. RNNs are very popular NN configurations besides CNNs and are suited for sequence modeling. The long short-term memory extension allows the network to discern which information to keep or to forget while processing the input sequence. Due to the bidirectional implementation, future and past information of the input sequence is simultaneously available for decision making in an offline fashion. Unfortunately, the recurrent structure of RNNs hinders them from maxing out modern hardware resources which are highly parallelized like graphic cards. This makes their training slow. More information about RNNs is available in [14, ch. 10] and about the model implementations in the respective papers [2, 4, 5].

3.4 Experiments

The final model is configured with a hidden width of 16 channels and a depth of 11 residual blocks followed by two 1x1 convolutions as described in section 2.2.3. This adds up to a total of 22978 trainable parameters. During training, a dropout of 0.2 has proven to reliably counteract overfitting. An overview of the whole network configuration is listed in table 2.2.

3.4.1 Training

The TapCorrect dataset is used for training in conjunction with a portion of the Maschine dataset which approximately matches TapCorrect's annotation count. Both datasets sum up to a total of 76131 beat annotations. The ballroom dataset is used for measuring the evaluation loss due to its diversity of genres. The remaining portion of the Maschine dataset and the Beatles dataset are reserved for testing the actual performance of the model.

The model was trained with batches of 256 training samples. A binary cross-

entropy loss measured its performance during the whole training process. An Adam optimizer was chosen and initialized with default parameters as described in paragraph 2.1.3, zero weight decay and a learning rate of 0.002. The training ended whenever the validation loss did not increase over the 50 most recent epochs as an early stop criteria similar to the work of Davies et al. [8]. The best performing model state is saved to be used for the final model comparison. An overview of the training setup is shown in table 3.3.

3.4.2 Hardware and Training-Time

The experiments were run on a desktop PC with a 4 core Intel Core i7 CPU 870 @2.93gHz, 8 GB of RAM and a single NVIDIA GeForce 970 graphics card with 4 GB of VRAM. Due to the hardware limitations of the PC the datasets could not be loaded fully into RAM and thus had to be loaded from a Samsung SSD 850 EVO with 500 GB. Nevertheless, the performance of the graphics card could on average be utilized more then 80% after some optimization efforts on the software implementation.

The training of the final model took 21 hours and 55 minutes for 145 epochs. Epoch #95 had performed best on the evaluation loss after a training time of 14 hours and 45 minutes. A second run was restored from the best performing epoch #95 to refine the model. This time the learning rate has been reduced by a factor of 5 similar to the work of Davies et al. [8]. This second training took additional 12 hours and 16 minutes ending with 182 epochs of which epoch #131 has performed best after a training time of 5 hours and 9 minutes. This adds up to a total training time of 34 hours and 1 minute or an effective training time of 19 hours and 54 minutes for the final model from epoch #131.

3.4.3 Evaluation Details

All models are compared to each other on all datasets looking at the complete music pieces instead of audio chunks as it is done during the training process. The output sequence of the models are describing the likelihood of a beat being present. This is also also known as the beat activation curve or function. Instead of using a threshold on the activation curve to extract final beat positions, a DBN [20] is used and initialized with the default parameters[3]. The FPS are set to 100 for the baseline models and approximately 115 for the proposed model.

With r being the receptive field of the proposed model, the input audio sequence

[3] BPM_{min}=55, BPM_{min}=215, transition-lambda=100, observation-lambda=16 (`https://madmom.readthedocs.io/en/latest/modules/features/beats.html#madmom.features.beats.DBNBeatTrackingProcessor`)

has to be padded with $\frac{r}{2}$ zeros on both sides to align the original input sequence to the downsampled output of the model. This is necessary because the model does not utilize padding (see section 2.2.3).

Table 3.3: Overview of the training and conditioning details of the network

Data Preparation	
Channels	Mono
Audio Sample Rate	22050 Hz
Zero Padding	at borders of a track
Attenuation	6 dB range
Inversion	0.5 probability
Translation (Beat Event)	±96 audio samples
Position (No-Beat Event)	random over whole track

Training	
Optimizer	*Adam*
Learning Rate	0.002 (0.0004)[4]
Loss Function	*Binary Cross-Entropy*
Test Datasets	Maschine_{test}, Beatles
Evaluation Dataset	Ballroom
Training Datasets	Maschine_{train}, TapCorrect
Training Samples	76131 beats + 152262 no-beats[5]
Beat Label Smoothing	$+2 frames$ (0.5 *weighted*)
Batch Size	256
Epochs	181

3.5 Results

The predicted beat timestamps are compared to the annotations using the **BeatEvaluation** class[6] from the *madmom* library initialized with its default parameters as described in section 3.2. The whole beat-tracking system based on the final model of the proposed network architecture and the DBN is further referred to as strided dilated convolution (SDC), named after its main mechanics.

[4]Once the stop criterion was reached the first time a second training run with the learning rate reduced by a factor of 5.

[5]The no-beat annotation count is two times the beat annotation count to maintain dataset balance in context of beat label smoothing.

[6]`https://madmom.readthedocs.io/en/latest/modules/evaluation/beats.html#madmom.evaluation.beats.BeatEvaluation`

3.5.1 Beat-Tracking Performance

Table 3.4 shows the performance of the SDC model in comparison to the baseline models over all datasets. The best performing model on the beat evaluation metrics is highlighted in bold text.

Table 3.4: Performance of the proposed network architecture in comparison with the beat detection implementations of Madmom.

	F_1	CML_c	CML_t	AML_c	AML_t	D
Beatles						
SDC	0.853	0.621	0.693	0.769	0.870	2.620
BLSTM[4]	0.930	0.793	0.859	0.842	0.914	**3.047**
BLSTM[5]	**0.936**	**0.809**	**0.870**	**0.855**	**0.930**	3.046
Maschine$_{test}$						
SDC	0.791	0.587	0.638	0.701	0.762	2.863
BLSTM[4]	0.846	**0.738**	**0.758**	0.803	0.825	3.118
BLSTM[5]	**0.850**	0.726	0.748	**0.808**	**0.833**	**3.259**
Maschine$_{train}$						
SDC	0.841	0.676	0.707	0.769	0.805	3.016
BLSTM[4]	0.847	**0.744**	**0.761**	**0.824**	**0.843**	3.066
BLSTM[5]	**0.849**	0.731	0.756	0.816	0.842	**3.172**
TapCorrect						
SDC	0.808	0.505	0.553	0.723	0.852	2.826
BLSTM[4]	**0.885**	**0.689**	**0.729**	**0.816**	**0.894**	**3.521**
BLSTM[5]	0.853	0.610	0.653	**0.816**	0.893	3.388
Ballroom						
SDC	0.805	0.599	0.616	0.800	0.841	2.849
BLSTM[4]	**0.933**	**0.856**	**0.876**	0.908	**0.932**	**3.475**
BLSTM[5]	0.915	0.816	0.829	**0.914**	**0.932**	3.368

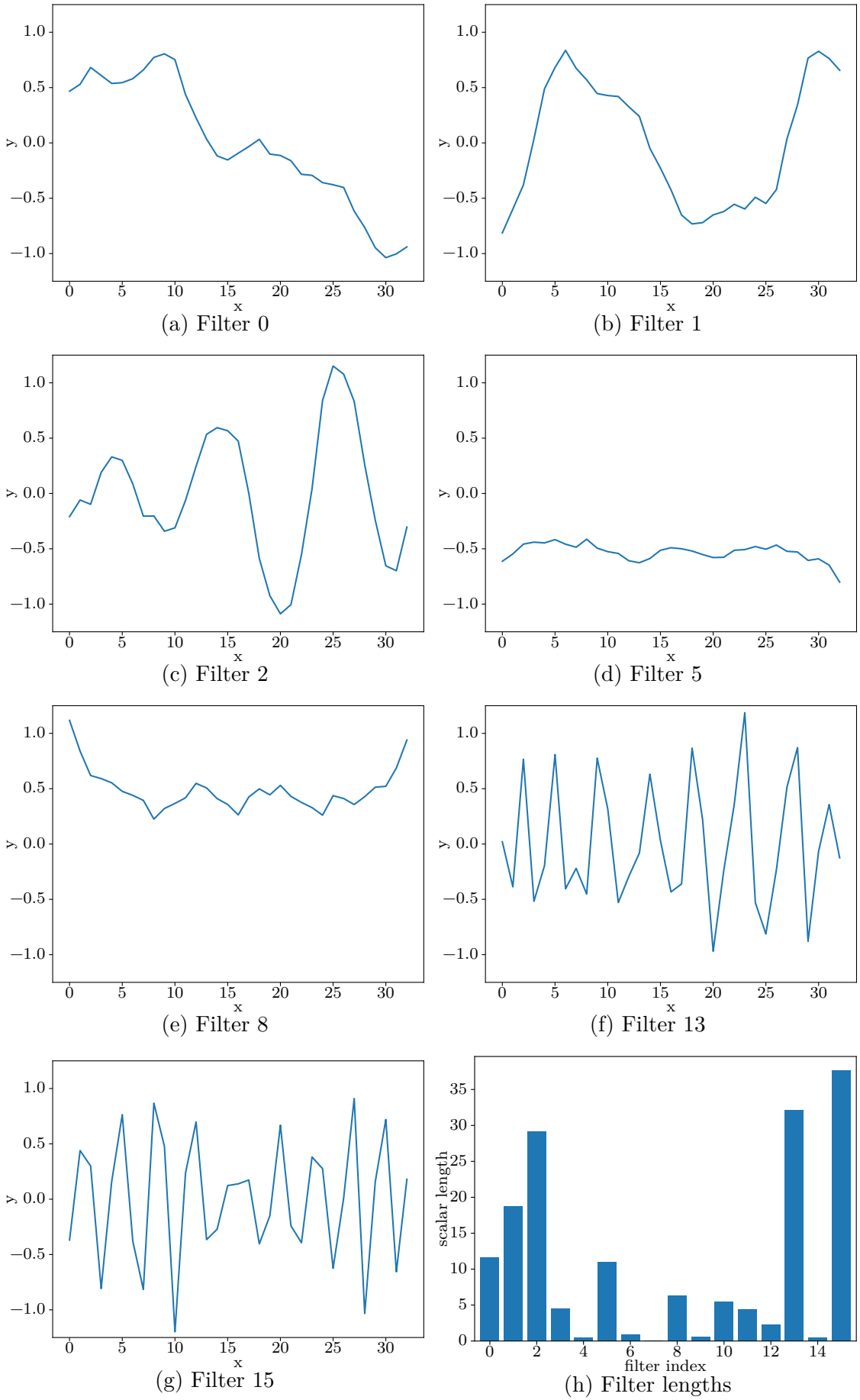
Investigating table 3.4 in more detail, it is noticeable that both baseline models are leading the scores across all metrics and datasets. Regarding the f-measure, SDC is on average about 7% below the better baseline performance with the smallest gaps on the Maschine datasets. The competitive performance of SDC on Maschine$_{train}$ seems reasonable since the model was trained on it, but this argument does not hold

up for the TapCorrect dataset. Looking at the continuity-based metrics, the discrepancy between CML and AML on the TapCorrect dataset indicates that SDC tends to misjudge beat occurrences while still figuring out the temporal structure. Even though the SDC performance is worst on the Maschine$_{test}$ dataset, having a small gap to the baseline indicates that SDC emphasizes knowledge on music with varying tempo due the bias towards tempo changes in the dataset itself. Furthermore, seeing the baseline models performing worst on the Maschine datasets indicates that it is more challenging compared to the others. The biggest difference between the baseline and SDC is on the ballroom dataset. Surprisingly SDC is performing best on the Beatles test dataset even though it has been used neither for training nor evaluation. This proves that the SDC model is capable to solve the task of detecting beats but with room for optimization.

3.5.2 Filter Visualization

Understanding how a NN is making its decisions is in general not comprehensible. Nevertheless, one way to get an intuition of its internal mechanics is to analyze the filter kernels of the layers. Therefore the 7 most active filters of the first convolutional layer of the trained model are shown in figure 3.2. They are displayed in a normalized representation due to the weight normalization applied to the model as described in section 2.1.3. Figure 3.2h shows the scalar length g associated to each filter modeling their impact. These filters are operating directly on the audio input sequence.

An interesting insight is the curvature of the 4 most active filters showing oscillations of more or less steady frequencies, eg. filter 1 (3.2b), 2 (3.2c), 13 (3.2f) and 15 (3.2g). This seems reasonable facing audio material. Looking closely at filter 15 (3.2g) and 13 (3.2f) having the highest associated scalar lengths, leads to the conclusion that higher frequencies are providing useful knowledge at this time resolution. The discrepancy of length scalars $\boldsymbol{g}$ is relative high in this layer, with the majority of filters being below its mean ($mean(\boldsymbol{g}) = 10.37$). Moreover, the scalar length of the filters 4, 6, 7, 9 and 14 is almost zero. This indicates that the model had problems recovering from not very useful filter kernels which seems like a waste of resources. One possible solution might be a more gradual channel expansion of the first layers to allow more efficient utilization of model parameters.



(a) Filter 0 (b) Filter 1

(c) Filter 2 (d) Filter 5

(e) Filter 8 (f) Filter 13

(g) Filter 15 (h) Filter lengths

Figure 3.2: The 7 most active filter kernels of the first convolutional layer and the lengths of all filter kernels.

4 Discussion

This thesis is targeting the topic of detecting beats in music with varying tempo by modeling it as a classification problem based on covolutional neural networks. Therefore a dataset has been created using the *Maschine Expansion* demo content of Native Instruments. The audio was reexported with the Maschine software by utilizing the beat information available in the project files. In this process, random tempo variations have been introduced for each song, artificially changing the base BPM of the originally fixed-grid computer-based music. Additionally, the songs beat information has been exported resulting in highly accurate and consistent annotations compared to hand-made ones. A robotic process automation was implemented to automate the process of exporting the data from the Maschine software. This newly created dataset consists of a total of 447 files spanning a total playtime of 12 hours and 36 minutes with 80955 beat annotations. This makes it the largest dataset among the others used in this thesis. The results of the evaluation have shown that the f-measure is the lowest on the Maschine dataset of all models. This seems to correlate with its broader range of tempo variations compared to the other datasets. Since an automation of the dataset creation process was implemented as part of this thesis, additional versions of the ME corpus could be created for future work and also made available to the research community. The tempo variations of these additional datasets could be between the one created in this thesis and no tempo variations at all.

Further, a CNN architecture has been proposed for predicting beat sequences from raw audio without relying on a pre-processing stage converting the audio into the frequency domain. The architecture utilizes the stride and dilation parameter of its convolutional layers to successively increase the temporal context of the input sequence. Residual connections are used to maintain a reliable information flow through all layers. The beat activation curve produced by the network is further processed with a DBN to extract the final sequence of beat timestamps. A model of the proposed network architecture has been trained, evaluated and tested on several datasets including the newly created one described above. The whole beat-tracking

system is composed of the trained model and the DBN, also called SDC.

The SDC achieves an f-measure of 85.3% on the Beatles test dataset which has been used neither for optimizing the model parameters nor for evaluating the model's performance during training. This result proves that the SDC is capable of solving the problem of detecting beats in music. The SDC has been compared to two state-of-the-art performing beat-tracking systems as a baseline, showing an average f-measure difference of 7% to the better performing baseline model. Since both baseline models are utilizing information from the frequency spectrum, this leads to believe that the information coming from the frequency domain gives valuable insights for detecting beats in music. Hints towards that can also be seen in the curvature of the most prominent filter kernels of the first layer of the SDC. Their oscillating structure is modeling frequency content as well. Future work could investigate an approach combining both domains (time and frequency) to train a NN. Furthermore, a multi-task approach including more temporal features like downbeats, tempo and meter could be researched as well as seen in [3]. To improve the performance of the proposed network architecture, it could be trained on a bigger collection of data utilizing cross-validation in the training procedure. Results could be improved further by specializing in specific music styles following the approach of Böck et al. [4].

The proposed network architecture has been designed in a highly compact fashion compared to the baseline models. It consists of a total of 22978 trainable parameters playing in a similar ballpark as the work of Davies et al. [8]. Similar to the findings of [8] the proposed network architecture is capable of encoding temporal information more efficiently compared to the RNN based baseline models. Computing performance improvements of a CNN based architecture as seen in this thesis in comparison to the RNN based baseline models have already been pointed out in the work of Davies et al. [8] with a similar network architecture, parameter count and baseline. Being able to train the proposed network architecture on moderate consumer hardware is a nice side effect of its performance-oriented design. In practice the proposed SDC should be even faster than the model of Davies et al. since it skips the computation of the FFTs completely.

Future work could adapt the presented approach to beat-tracking to act in realtime by turning every non-causal decision into a causal one. For this data loading, padding and timestamp positioning have to be reconfigured as well as adapting the internal truncation mechanics of the network. As long as the aforementioned adjustments are adopted the model could be trained more or less the same way as its offline version. Having a well-performing realtime beat-tracking system adapting on varying tempo has a huge potential for improved immediate communication of rhythm within a computer-based live setup in conjunction with classical instruments

played by humans. The task of giving modern technology the tools to interpret and work with the human element is a complex one but will play an integral part in the ongoing process of modern music development.

Bibliography

[1] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. "An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling." In: *arXiv:1803.01271 [cs]*. Mar. 2018. URL: http://arxiv.org/abs/1803.01271 (visited on 08/13/2019).

[2] Sebastian Böck. "Enhanced Beat Tracking With Context-Aware Neural Networks." en. In: (2011), p. 5.

[3] Sebastian Böck, Matthew E P Davies, and Peter Knees. "Multi-Task Learning of Tempo and Beat: Learning One to Improve the Other." en. In: (2019), p. 8.

[4] Sebastian Böck, Florian Krebs, and Gerhard Widmer. "A Multi-Model Approach to Beat Tracking Considering Heterogeneous Music Styles." en. In: (2014), p. 6.

[5] Sebastian Böck, Florian Krebs, and Gerhard Widmer. "Joint Beat and Downbeat Tracking with Recurrent Neural Networks." In: *Proceedings of the 17th International Society for Music Information Retrieval Conference*. 2016.

[6] Sebastian Böck et al. "madmom: a new Python Audio and Music Signal Processing Library." In: *Proceedings of the 24th ACM international conference*. May 2016. URL: http://arxiv.org/abs/1605.07008 (visited on 11/27/2018).

[7] Roger Dannenberg and Sukrit Mohan. "Characterizing Tempo Change In Musical Performances." In: *Proceedings of the International Computer Music Conference 2011*. Jan. 2011. URL: https://www.cs.cmu.edu/rbd/papers/Tempo-Change-ICMC-2011.pdf.

[8] Matthew E P Davies and Sebastian Bock. "Temporal Convolutional Networks for Musical Audio Beat Tracking." en. In: (2019), p. 5.

[9] Matthew E P Davies, Norberto Degara, and Mark D Plumbley. "Evaluation Methods for Musical Audio Beat Tracking Algorithms." en. In: (2009), p. 17.

[10] Jonathan Driedger et al. “Towards Automatically Correcting Tapped Beat Annotations for Music Recordings.” en. In: (2019), p. 8.

[11] John Duchi, Elad Hazan, and Yoram Singer. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.” en. In: (2011), p. 39.

[12] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks.” en. In: (2010), p. 8.

[13] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. “Deep Sparse Rectifier Neural Networks.” en. In: (Apr. 2011), p. 9.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[15] F. Gouyon et al. “An experimental comparison of audio tempo induction algorithms.” en. In: *IEEE Transactions on Audio, Speech and Language Processing* 14.5 (Sept. 2006), pp. 1832–1844. ISSN: 1558-7916. DOI: 10.1109/TSA.2005.858509. URL: http://ieeexplore.ieee.org/document/1678001/ (visited on 10/14/2020).

[16] Kaiming He et al. “Deep Residual Learning for Image Recognition.” en. In: *arXiv:1512.03385 [cs]* (Dec. 2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385 (visited on 05/08/2020).

[17] Sepp Hochreiter et al. *Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies*. 2001.

[18] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.” en. In: *arXiv:1502.03167 [cs]* (Mar. 2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167 (visited on 10/15/2020).

[19] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization.” In: *arXiv:1412.6980 [cs]* (Jan. 2017). URL: http://arxiv.org/abs/1412.6980 (visited on 11/25/2019).

[20] Florian Krebs, Sebastian Bock, and Gerhard Widmer. “An Efficient State-Space Model for Joint Tempo and Meter Tracking.” en. In: (2015), p. 7.

[21] Florian Krebs, Sebastian Boeck, and Gerhard Widmer. “Rhythmic pattern modeling for Beat and Downbeat Tracking in musical audio.” en. In: (2013), p. 6.

[22] Aaron van den Oord et al. “WaveNet: A Generative Model for Raw Audio.” In: *arXiv:1609.03499 [cs]* (Sept. 2016). URL: http://arxiv.org/abs/1609.03499 (visited on 08/14/2019).

[23] Hendrik Purwins et al. “Deep Learning for Audio Signal Processing.” In: *IEEE Journal of Selected Topics in Signal Processing* 13.2 (May 2019), pp. 206–219. ISSN: 1932-4553, 1941-0484. DOI: 10.1109/JSTSP.2019.2908700. URL: http://arxiv.org/abs/1905.00078 (visited on 02/17/2020).

[24] Tim Salimans and Diederik P. Kingma. “Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks.” In: *arXiv:1602.07868 [cs]* (June 2016). arXiv: 1602.07868. URL: http://arxiv.org/abs/1602.07868 (visited on 05/12/2020).

[25] W. A. Schloss. “On the Automatic Transcription of Percussive Music - From Acoustic Signal to High-Level Analysis.” MA thesis. Stanford, CA: Stanford University, 1985. URL: https://ccrma.stanford.edu/files/papers/stanm27.pdf.

[26] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting.” en. In: (2014), p. 30.

[27] Richard Vogl et al. “Drum Transcription via Joint Beat and Drum Modeling using Convolutional Recurrent Neural Networks.” In: *Proceedings of the 18th Inter- national Society for Music Information Retrieval Conference*. 2017.

[28] Nick Whiteley, A Taylan Cemgil, and Simon Godsill. “Bayesian Modelling of Temporal Structure in Musical Audio.” en. In: (2006), p. 6.

Acronyms

Adam adaptive moment estimation. 16, 17

AML allowed metrical level. 42, 46

BLSTM bidirectional long short-term memory. 3, 4, 43

BPM beats per minute. 38, 39, 44, 49

CML correct metrical level. 42, 46

CNN covolutional neural network. 4, 22, 23, 26, 43, 49, 50

DBN dynamic Bayesian network. 3, 4, 5, 26, 31, 32, 33, 42, 44, 45, 49

DSP digital signal processing. 22

ELU exponential linear unit. 20

FFT fast fourier transoform. 5, 50

FPS frames per second. 31, 34, 44

KLD Kullback–Leibler divergence. 42

LSTM long short-term memory. 4

ME *Maschine Expansion.* 5, 38, 49

MIR music information retrieval. 2, 4, 27

MIREX *Music Information Retrieval Evaluation eXchange.* 4

ML machine learning. 2, 5, 7, 12, 17, 18, 22

MSE mean squared error. 11

NI Native Instruments. 2, 5, 27, 38, 49

NN neural network. vii, xi, 2, 3, 4, 5, 7, 9, 10, 11, 12, 13, 17, 18, 20, 21, 25, 26, 27, 28, 29, 32, 37, 38, 43, 47, 50

ReLU rectified linear unit. 19

RNN recurrent neural network. 43, 50

RPA robotic process automation. 39, 49

SDC strided dilated convolution. 45, 46, 49, 50

SGD stochastic gradient descent. 13, 14, 15, 16, 21

TCN temporal convolutional network. 4

Index