Manuel Planton, BSc

Master's Thesis

Instrument-Specific Music Source Separation via Interpretable and Physics-Inspired Artificial Intelligence

to achieve the university degree of
Diplom-Ingenieur
Inter-universitary Master's degree programme:
Electrical Engineering and Audio Engineering
(UF 066 413)

at the

University of Music and Performing Arts, Graz
University of Technology, Graz

Supervisor: O.Univ.Prof. Mag.art. DI Dr.techn. Robert Höldrich

Graz, February 27, 2023

institut für elektronische musik und akustik

Abstract

Music source separation is the task of extracting individual instrument tracks from a joint music mixture. State-of-the-art music source separation methods employ neural networks exclusively. Most current neural network architectures for music source separation do not contain structural prior knowledge about the sources whose signals are to be extracted from the musical mix signal. This work examines the topic if it is possible to incorporate knowledge about the guitar into a neural network for music source separation using a physical string model. Furthermore it deals with the question if and how an improvement of guitar string signal separation quality is possible by refining the physical string model incorporated in the neural network. The proposed method serves as a proof of concept for introducing differentiable physical modeling synthesis into neural music source separation, leading to a basis for potential high quality guitar string separation. Even better source separation methods of musical instruments yield applications in professional audio production such as remixing, upmixing, extracting stems from single microphone recordings, reduction of microphone crosstalk (mic bleed) and finer control of sound objects in audio tracks. This method is not limited to audio applications and may be extended to other fields in the future.

Kurzfassung

Quellentrennung von Musik ist die Separierung einzelner Instrumentenspuren aus einer gemeinsamen Musikaufnahme. Die aktuell besten Methoden zur Musik-Quellentrennung verwenden ausnahmslos Neuronale Netze. Aktuell verwendete Netzwerkarchitekturen enthalten selten strukturelles Vorwissen über die Signalquellen, deren Signale aus dem musikalischen Mischsignal extrahiert werden sollen. Diese Arbeit behandelt das Thema, ob es möglich ist, Wissen über die Gitarre in ein neuronales Netzwerk für Musikquellentrennung mittels eines physikalischen Saitenmodells einzubringen. Weiters beschäftigt sie sich mit der Frage, ob und wie eine Verbesserung der Trennung von Gitarrensaitensignalen möglich ist, indem das physikalische Saitenmodell verfeinert wird, welches im neuronalen Netzwerk verwendet wird. Die vorgeschlagene Methode dient als proof of concept für die Einbringung von differenzierbarer Physical Modeling Synthese in die neuronale Musikquellentrennung. Dies führte zu einer Basis für potentiell qualitativ hochwertiger Gitarrensaiten-Signaltrennung. Mit noch besseren Methoden zur Quellentrennung von Musikinstrumenten ergeben sich unter anderem Anwendungen im Bereich der professionellen Audioproduktion wie Remixing, Upmixing, Einzelspuren aus Aufnahmen mit einem Hauptmikrofon erhalten, Verminderung von Mikrofon-Übersprechen (mic bleed) und feinere Kontrolle von Klangobjekten in einer Audiospur. Diese Methode ist nicht auf Anwendungen im Audiobereich beschränkt und könnte in Zukunft auf andere Felder erweitert werden.

EIDESSTATTLICHE ERKLÄRUNG

Hiermit bestätige ich, dass mir der Leitfaden für schriftliche Arbeiten an der KUG bekannt ist und ich die darin enthaltenen Bestimmungen eingehalten habe. Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst habe, andere als die angegebenen Quellen nicht verwendet habe und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Ort, Datum	Unterschrift

Contents

Chapter 1

Introduction

This work deals with instrument-specific music source separation using interpretable and physics-inspired artificial intelligence. It is achieved by encoding prior knowledge about the instrument into a neural network architecture for music source separation. The knowledge is encoded as a physical instrument model which receives neurally predicted control signals. These control signals are interpretable in the sense that humans understand the meaning behind them. From these control signals the source signals from a mixture signal are synthesized with the physical model given the predicted control signals.

In 2020 [EHGR20] introduced the concept of differentiable digital signal processing. This lead to a multitude of publications and proved to be beneficial in a number of fields. The concept is to include traditional digital signal processing elements into neural networks, leading to the application in [EHGR20] where it was used for singing voice separation. In this work the method for singing voice separation is adopted for guitar string signal separation. That is to say, a mixture signal such as a monophonic guitar recording is split up into individual guitar string signals.

Methods which employ differentiable digital signal processing proved to be data-efficient and are even able to train unsupervised. Therefor the above mentioned method for music source separation only needs mixture signals for training. The proposed models in the experiments are successors of this method and are trained with a preliminary stage of unsupervised learning. This makes the proposed method's training process effectively supervised, but may be improved in the future by building on the results achieved here.

The thesis is structured as follows. Chapter 2 gives an introduction to deep learning and chapter 3 covers the fundamentals of music source separation, building up to a summary of state-of-the-art methods in this field. Chapter 4 contains an explanation and a literature review of differentiable digital signal processing. The conducted experiments are introduced in chapter 5 followed by the results and discussion of these experiments in chapter 6. Finally, the work is concluded in chapter 7 which also gives an outlook for future research.

Chapter 2

Deep Learning

Neural networks containing multiple layers are called deep neural networks and belong to the group of machine learning systems. The process of training such deep neural networks is referred to as deep learning. Deep learning is currently used to solve new problems and to improve existing systems in an extremely wide range of fields.

An overview of machine learning can be found in [Bur19] and [PK20]. An informal introduction is found in [Gé19] and a more formal definition is given by [Bis06]. Literature about deep learning specifically is found in [Nie15]. Overviews on deep learning for audio signal processing, audio generation and music information retrieval can be found in [PLV+19], [ZXT19] and [CFCS17].

In this chapter first the training process of neural networks is described, treating neural networks mostly as black-boxes. Subsequently, the neural network layers are defined which are the building blocks of neural networks. Then, a selection of neural network architectures are introduced and the chapter concludes with important properties of neural networks regarding this work.

2.1 The Neural Network as a Black Box

Considering a *neural network* (NN) as a black box, it is commonly used to map its input to a certain output with respect to a set of training data. NNs can be viewed as multiple nested functions with numerous free parameters. These parameters are called *weights* and are initialized in a certain way (mostly drawn from a certain random distribution) and then optimized to map the input data to the desired output data. This iterative optimization process is called *training* the NN in which it *learns* the desired mapping. The optimization goal is formalized with the *loss function* or *cost function* and the weights are optimized by minimizing this loss.

The usual goal in training a NN is to have it learn *patterns* from the training data to produce a desired output for unseen new input data. Hence, a NN is also referred to as *model*

because it is a mathematical model *fitted* to the training data.[1] These two terms, neural network and model, are used interchangeably in the literature. A model *generalizes* well if its performance on new unseen data is comparable to its performance on the training data.

To assess the models performance commonly a *test dataset* or abbreviated *test set* is reserved from the overall available data. The data used for model training is called the *training set*. Often also a *validation set* is used between training steps for tuning *hyperparameters* which are parameters of the overall system employed for training. The training set normally consists of roughly 70% to 95% of the overall available data [Bur19]. The remaining 30% to 5% are either utilized as test set or split between a test set and a validation set. Model performance is measured either directly via the loss function used for training or a certain *metric* is employed, which can measure the quality of the models output for the given task.

Depending on the models complexity, type, structure and training progression and also on the loss function, *overfitting* or *underfitting* can occur. The model shows underfitting when it performs bad at the test set. This is indicated by a high training loss produced by poor predictions. On the other hand the model shows overfitting when it performs well on the training set but performs poorly on the test set. This is indicated by a low training loss and a high test loss.

Following [Bis06] and [PK20], data $\mathcal{X}$ for machine learning can be either *unlabeled data* as defined in eq. (2.1) or *labeled data* as defined in eq. (2.2) with the number of data samples N.

$$\mathcal{X} = \{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N\} \tag{2.1}$$

$$\mathcal{X} = \{\langle \boldsymbol{x}_1, \boldsymbol{t}_1 \rangle, \cdots, \langle \boldsymbol{x}_N, \boldsymbol{t}_N \rangle\} \tag{2.2}$$

The set of unlabeled data consists only of the data samples $\boldsymbol{x}_n = [x_n^{(1)}, \cdots, x_n^{(D_{in})}]^T$ of dimension D_{in} with the transpose $(\cdot)^T$. If unlabeled data is used for training exclusively, the training process is called *unsupervised learning*. The inputs $\boldsymbol{x}_n$ are also referred to as *features* in machine learning.

Labeled data consists of tuples with the according *targets* $\boldsymbol{t}_n = [t_n^{(1)}, \cdots, t_n^{(D_{out})}]^T$ of dimension D_{out}. If labeled data is used for training, the training process is referred to as *supervised learning*.

2.1.1 High Level Overview

Figure 2.1 depicts a usual training process for NNs. The NN takes the input data sample $\boldsymbol{x}_n$ and produces from it the neural network output or *prediction* $\boldsymbol{y}_n = [y_n^{(1)}, \cdots, y_n^{(D_{out})}]^T$.

1. This term is also used in statistics.

When $y_n = c \in \mathbb{Z}$ consists of classes c, the model performs a *classification* task. Thereby it outputs a class number c according to its input $\boldsymbol{x}_n$. A *regression* task is performed when $\boldsymbol{y}_n \in \mathbb{R}^{D_{out}}$.

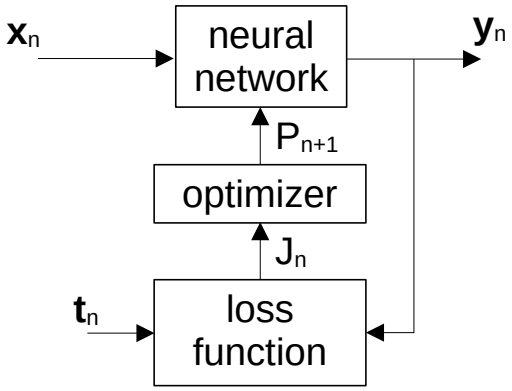


Figure 2.1 – Block diagram of the neural network training process with input data sample $\boldsymbol{x}_n$, prediction $\boldsymbol{y}_n$, target data sample $\boldsymbol{t}_n$, loss function $J(\boldsymbol{y}_n, \boldsymbol{t}_n)$ and the set of learnable parameters P of the neural network.

The loss function $J(\boldsymbol{y}_n, \boldsymbol{t}_n) \in \mathbb{R}$ is calculated after the NN predicted its output according to its input. This loss is a scalar value J_n.

In certain systems it is desired that the NN is optimized to reproduce its input on the output in an unsupervised fashion. Hence the target becomes the input $\boldsymbol{t}_n = \boldsymbol{x}_n$ and the NN output becomes the input estimation $\boldsymbol{y}_n = \hat{\boldsymbol{x}}_n$. In this case the loss function is $J(\boldsymbol{x}_n, \hat{\boldsymbol{x}}_n)$.

J_n is then used from the *optimizer* to update the current NN parameters P_n to the parameters P_{n+1} in such a way that the loss is reduced. By yielding P_{n+1} the learning step n is finished. The updated network parameters P_{n+1} are then used for the next prediction. In this way the NN is trained *multiple times* on the whole training set. One such training pass of the whole training set is called an *epoch*. After training and testing the model, it is employed to predict outputs from new unseen data, which is referred to as *inference*.

The description of the NN training process above serves as an introduction and as an overview. In this overview the important parts

— neural network,
— optimizer, and
— loss function

were intentionally treated as black boxes to gain a high level understanding. In the following sections these elements are clarified in more detail.

2.1.2 The Neural Network

NNs are typically composed of multiple layers. For example a *dense layer* in eq. (2.3) performs a linear transformation on the input and applies a *vectorized* non-linear *activation*

function f to the result.

$$\boldsymbol{y}_n = f(W^l \boldsymbol{x}_n + \boldsymbol{b}^l) \tag{2.3}$$

A scalar function $f : \mathbb{R} \to \mathbb{R}$ is vectorized by applying it separately on all vector elements $f(\boldsymbol{x}_n) = [f(x_n^{(1)}), \cdots, f(x_n^{(D)})]^T \in \mathbb{R}^D$ [Nie15]. The *weights* matrix $W^l \in \mathbb{R}^{D_{out} \times D_{in}}$ and the *bias* vector $\boldsymbol{b}^l \in \mathbb{R}^{D_{out}}$ hold the trainable parameters of the layer l. The superscript l may not be confused with an exponent as it is the layer index of a NN composed of multiple neural layers.

This dense layer originates historically from the *artificial neuron* which itself is inspired by the biological neuron. There are other layer types too which are described below. Mathematically the artificial neuron is related to linear and logistic regression. In linear regression as described in [PK20] a linear model $y_n = \boldsymbol{w}^T \boldsymbol{x}_n \approx t_n$ is fitted to the training set by finding the optimum solution of the weights vector $\boldsymbol{w} = [w_1, \cdots, w_D]^T$ and the bias b_0. The least squares optimum weights vector $\boldsymbol{w}^*$ and bias b_0^* are obtained via an analytical solution.

Logistic regression

$$y_n = f(\boldsymbol{w}^T \boldsymbol{x}_n + b_0) \approx t_n \tag{2.4}$$

is equivalent to the artificial neuron model. Originally the differentiable non-linear activation function $f(x)$ is given by the sigmoid function $\sigma(x)$.[2]

$$\sigma(x) = \frac{1}{1 - e^{-x}} \in (0, 1) \tag{2.5}$$

For logistic regression and the artificial neuron, respectively, no analytical solution for the optimum weights and bias can be found. Hence the optimization is done via an iterative gradient descent method. Artificial neurons from eq. (2.4) are stacked to form a dense layer described in eq. (2.3). For notational convenience all trainable parameters of the NN with a number of L layers are summarized into the trainable parameters set $P = \{W^1, \boldsymbol{b}^1, \cdots, W^L, \boldsymbol{b}^L\}$.

2.1.3 The Optimizer

Given a NN with multiple dense layers and the training set $\mathcal{X}$ an optimum solution is approached by iteratively updating P according to minimize the loss function $J(P) = J(\boldsymbol{y}(\boldsymbol{x}_n, P), \boldsymbol{t}_n)$ via the common iterative *gradient descent* optimization method

$$P_{n+1} = P_n - \eta \nabla_P J(P) \tag{2.6}$$

2. The historical precursor of the artificial neuron is the *perceptron* which used a step function as an activation function f.

where η denotes the *learning rate* or *step size* of the algorithm and $\nabla_P J(P)$ is the gradient of the loss function with respect to all trainable parameters P of the NN. Picturing the loss function as error surface above the trainable parameters, gradient descent takes steps of length η to a minimum of the error surface because the negative gradient points to the steepest descent at the given point on the surface. If the global minimum would be found, the algorithm would have successfully optimized the weights of the NN according to the training set. Although in general the error surface of a NN is not convex. Hence gradient descent most likely finds a local minimum but not the global minimum. However finding a local minimum of the error surface is sufficient in many cases. This training technique requires all parts (layers) of the NN and the loss function to be *differentiable*.

Plotting the loss over the training steps is referred to as the *learning curve* of a model. Usually a learning curve shows the arithmetic mean loss per epoch.

The Backpropagation Algorithm

The *error backpropagation algorithm* or short *backpropagation algorithm* is an efficient method to calculate the gradient $\nabla_P J(P)$ by exploiting the chain rule [Bis06]. As the name implies the backpropagation algorithm works by taking the error of the last layer which is the same as the result of the loss function J and successively calculating the gradient for the output up to the input layer. In this way the loss gradient is *propagated* from the last layer back to the first layer of the NN.

Gradient Descent Variants

Gradient descent computes the gradient of the cost function for every single training example using the backpropagation algorithm to get the gradient of the cost function. The NN training process above was outlined in this fashion that every data sample $\langle \boldsymbol{x}_n, \boldsymbol{t}_n \rangle$ is used to update the parameters P of the NN. In practice it is common to alter the optimization algorithm to perform

— *batch gradient descent*,
— *stochastic gradient descent* or
— *mini-batch gradient descent*

or others. The gradient descent variants differ in computational efficiency and NN learning behavior.

Batch gradient descent performs one training step in the direction of the average of the error gradients according to the full training set [Gé19]. This means, that every epoch the NN parameters P are updated only once. Hence this method is computationally inefficient. Learning curves are smooth but tend to get stuck in local minima.

Stochastic gradient descent picks random data samples from the training set. The gradient is calculated for every single randomly chosen data sample and then a learning step is taken. Learning curves are noisy but may find a way out of local minima.

For mini-batch gradient descent, the training set is randomly divided into small chunks

called *mini-batches*.[3] For every mini-batch of randomly chosen data samples one gradient descent learning step is performed. This learning step is taken into the direction of the average gradient of the samples in the mini-batch.

Mini-batch gradient descent is the main method employed for training NNs since calculation of matrix operations are optimized especially when using GPUs [Gé19] and fit in memory for mini-batches. Usually training is done in several epochs where the mini-batches contain different randomly selected samples from the training set. The mini-batch gradient descent method can be viewed as trade-off between batch gradient descent and stochastic gradient descent in terms of learning behavior and computational efficiency.

In automatic differentiation (auto-diff) libraries such as TensorFlow[4] or PyTorch[5], these learning algorithms are called *optimizers* since they iteratively solve the optimization problem for the weights of neural networks. One of the most used optimizers in the literature is called *Adam* [KB14] but there are the classic methods described above and others as well.

2.1.4 The Loss Function

As described above the loss function $J(\boldsymbol{y}(\boldsymbol{x}_n, P), \boldsymbol{t}_n)$ evaluates to a scalar value for every training step. The loss function is also called error function or cost function. The learning process is highly dependent on the choice of the loss function since it defines the optimization goal for the whole system.

The loss or error can be pictured as a distance from the predicted sample $\boldsymbol{y}_n$ to the target sample $\boldsymbol{t}_n$. Optimizing with batch gradient descent, or as usual with mini-batch gradient descent, an accumulated error measure has to be calculated to yield a scalar. Hence often the mean or the sum of the individual distances between predictions and targets in a batch/mini-batch are calculated. Common generic loss functions are

— the *mean absolute error* (MAE, L_1-distance): $\frac{1}{N}\sum_{n=1}^{N}|\boldsymbol{y}_n - \boldsymbol{t}_n|$
— the *mean square error* (MSE, L_2-distance): $\frac{1}{N}\sum_{n=1}^{N}(\boldsymbol{y}_n - \boldsymbol{t}_n)^2$ and
— the *cross-entropy loss*.

Considering audio applications [PLV+19] says:

> A crucial and creative part of the design of a deep learning system is the choice of the loss function. The loss function needs to be differentiable with respect to trainable parameters of the system when gradient descent is used for training. The mean squared error (MSE) between log-mel spectra can be used to quantify the difference between two frames of audio in terms of their spectral envelopes. To account for the temporal structure, log-mel spectrograms can be compared. However, comparing two audio signals by taking

3. In the literature the word batch is used for the whole training set and mini-batches are small subsets of the training set. However mini-batches are in practice mostly referred to in the short form *batches* especially in code. Hence the nomenclature is ambiguous and should be interpreted from the context.
4. https://www.tensorflow.org/
5. https://pytorch.org/

the MSE between the samples in the time domain is not a robust measure.

The MSE or the MAE are *not robust measures* because small errors in the synthesis of audio signals produce large errors depending on the phase of the signals. Especially when the network output $\boldsymbol{y}_n$ is audio data like music or speech a phase independent loss function should be considered.

2.1.5 Activation Functions

To introduce nonlinearity into a NN, nonlinear activation functions such as in Figure 2.2 are used.

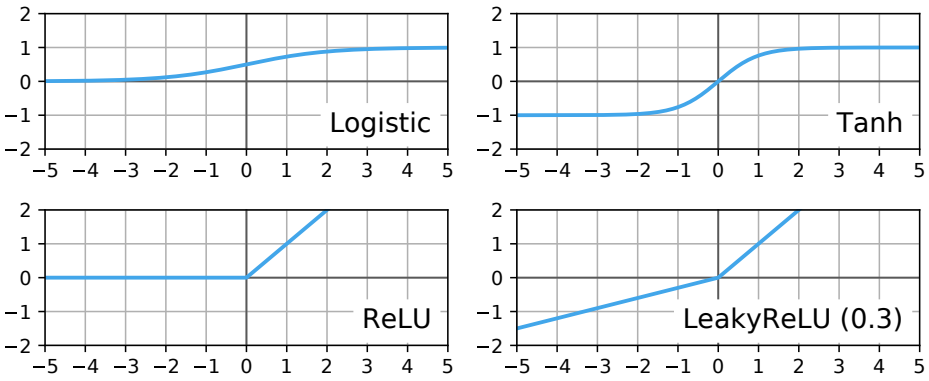


Figure 2.2 – Different popular activation functions from [CFCS17].

These are the currently most popular choices for output layers (top) and hidden layers (bottom). The logistic function which is also called *sigmoid function* can be used after any layer but the *rectified linear unit* (ReLU) showed to have good properties for hidden layers [Gé19]. Output range and scaling, learning behavior or stability are reasons to employ other activation functions such as the hyperbolic tangent, the leaky ReLU or others.

2.2 Neural Network Layers

A NN with reasonable performance at a non-trivial task usually consists of multiple layers. For audio applications "multiple feedforward, convolutional, and recurrent (...) layers are usually stacked to increase the modeling capability." [PLV+19] In this section different neural layers are described which are the building blocks of NN architectures in general and also of neural source separation architectures. Layers are divided into

— an *input layer*,
— an *output layer*, and
— multiple *hidden layers* in between

depending on their position in the NN.

The input layer receives the input data sample $\boldsymbol{x}_n$ and the output layer outputs the NN prediction $\boldsymbol{y}_n$. In between are the hidden layers connected in series consecutively. Input and hidden layers in DNNs demand different activation functions than the output layer. The output layers activation function is determined by the systems task and by the target value range $\boldsymbol{t}_n$.

A *deep neural network* (DNN) is a neural network with multiple hidden layers [PLV+19]. The more complex the problem is which the network should solve, the more hidden layers are needed. In the following the three basic NN layer types

— the dense layer,
— the convolutional layer (pooling layer, dilated convolutional layer), and
— the recurrent layer (RNN, GRU)

are described.

Dense layers are general purpose layers, convolutional layers are often used for two-dimensional inputs with local structures and recurrent layers are often used for time series.

2.2.1 Dense Layer

A *dense layer* in the literature is also called *fully connected layer*, *linear layer*, *linear transform* or *affine transform*. All these names describe a linear map with a subsequent application of an activation function f.

There are two common notations for a dense layer in the literature which are mathematically identical. Starting from eq. (2.3)

$$\begin{aligned}\begin{bmatrix} y_n^{(1)} \\ \vdots \\ y_n^{(D_{out})} \end{bmatrix} &= f\left(\begin{bmatrix} W^l_{1,1} & \cdots & W^l_{1,D_{in}} \\ \vdots & \ddots & \vdots \\ W^l_{D_{out},1} & \cdots & W^l_{D_{out},D_{in}} \end{bmatrix}\begin{bmatrix} x_n^{(1)} \\ \vdots \\ x_n^{(D_{in})} \end{bmatrix} + \begin{bmatrix} b^l_1 \\ \vdots \\ b^l_{D_{out}} \end{bmatrix}\right) = \\ &f\left(\begin{bmatrix} b^l_1 & W^l_{1,1} & \cdots & W^l_{1,D_{in}} \\ \vdots & \vdots & \ddots & \vdots \\ b^l_{D_{out}} & W^l_{D_{out},1} & \cdots & W^l_{D_{out},D_{in}} \end{bmatrix}\begin{bmatrix} 1 \\ x_n^{(1)} \\ \vdots \\ x_n^{(D_{in})} \end{bmatrix}\right)\end{aligned} \tag{2.7}$$

which differs in the position of the bias vector $\boldsymbol{b}$. In the first expression the bias vector $\boldsymbol{b}$ is separated from the weights $W^l_{j,i}$ and in the second expression the bias vector is a part of the weight matrix which requires an extended input vector $\boldsymbol{x}_n = [1, x_n^{(1)}, \cdots, x_n^{(D_{in})}]^T$.

Due to this notational difference there are also two common graphs to represent a dense layer which consists of artificial neurons. The first expression in eq. (2.7) leads to a graph with a separate bias node with an input of 1 for every dense layer and the second expression includes the bias inside the neuron as depicted in Figure 2.3. The artificial neuron j applies its weights $W^l_{j,i}$ to its inputs which are all outputs i from the previous layer and sums the results including its bias b^l_j. One artificial neuron accounts for D_{in} weights and

1 bias as depicted in Figure 2.3. The resulting sum of the bias and the weighted inputs is fed through the activation function f which forms the output $y_n^{(j)}$.

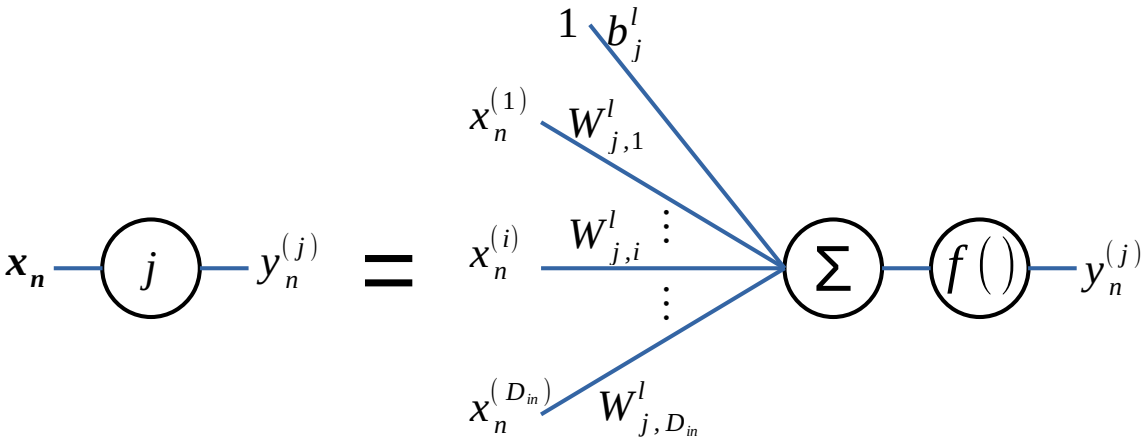


Figure 2.3 – Graph of an artificial neuron with input $\boldsymbol{x}_n = [x_n^{(1)}, \cdots, x_n^{(D_{in})}]^T$, output $y_n^{(j)}$, weights $W_{j,i}^l$ and bias b_j^l. The indices are the data index n, input data dimension D_{in}, neuron index i in the previous layer and neuron index j of the current layer.

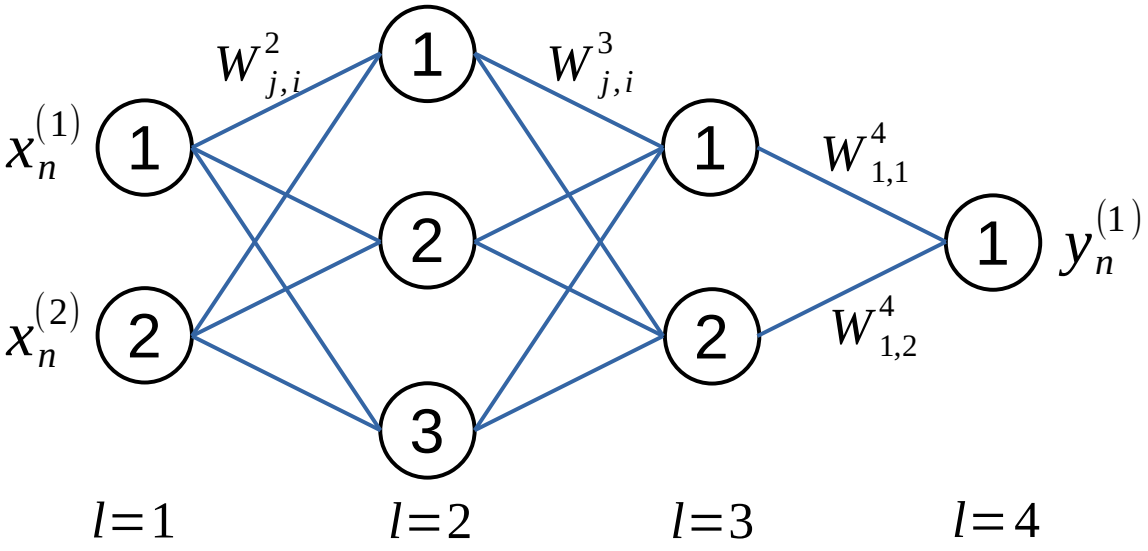


Figure 2.4 – Graph of a simple dense NN with 4 layers, with layer number l and weights $W_{j,i}^l$.

Every neuron j in a dense layer l is connected to every neuron in the previous layer i (hence the term fully connected layer). A simple example of a NN with two hidden layers is depicted in Figure 2.4. It consists of 4 layers numbered with the index l and the data flow is from left to right. Layer 1 is the input layer, layer 4 is the output layer and layers 2 and 3 are hidden layers.

2.2.2 Convolutional Layer

A convolutional layer learns a number of i filter kernels $\boldsymbol{w}[i]$. The layers input signal $x[n]$ is successively convolved with all i filter kernels producing i outputs $\boldsymbol{y}[i, n]$ which are also called *feature maps* (especially in image processing networks). The linear convolution of a one-dimensional time signal $\boldsymbol{x}[n] = [x[n], x[n-1], \cdots, x[n-N+1]]^T$ of length N with the real time-invariant filter kernel $\boldsymbol{w}[i] = [w_{i,0}, w_{i,1}, \cdots, w_{i,M-1}]^T$ with length M is given by

$$y[i, n] = w_{i,m} * x[n] = \sum_{m=0}^{M-1} w_{i,m} x[n-m] = \boldsymbol{w}^T[i]\boldsymbol{x}[n] \tag{2.8}$$

with the convolution operator $*$, the time index n and the filter coefficient index m.

In the most cases $M < N$. Eq. (2.8) calculates one output sample $y[i, n]$. To yield the entire output signal $\boldsymbol{y}[i, n] = [y[i, n], y[i, n-1], \cdots, y[i, n-Q+1]]$ of length $Q = M + N - 1$, the linear convolution can be written as matrix multiplication

$$\boldsymbol{y}[i, n] = W^T[i]\boldsymbol{x}[n] \tag{2.9}$$

with the convolution matrix defined in eq. (2.10).

$$W[i] = \begin{bmatrix} \boldsymbol{w}[i] & & 0 \\ & \ddots & \\ 0 & & \boldsymbol{w}[i] \end{bmatrix} = \begin{bmatrix} w_{i,0} & 0 & \cdots & 0 \\ w_{i,1} & w_{i,0} & & \vdots \\ \vdots & w_{i,1} & \ddots & 0 \\ w_{i,M-1} & \vdots & \ddots & w_{i,0} \\ 0 & w_{i,M-1} & & w_{i,1} \\ \vdots & 0 & \ddots & \vdots \\ 0 & 0 & & w_{i,M-1} \end{bmatrix} \in \mathbb{R}^{Q \times N} \tag{2.10}$$

Eq. (2.9) gives a compact representation of the convolutional layer and is very similar to the dense layer in eq. (2.3). The important difference is that weights get *shared* along a whole input signal $\boldsymbol{x}[n]$ for every filter kernel $\boldsymbol{w}[i]$ in convolutional layers, whereas dense layers demand a weight from every input to every neuron.

In practice convolutional layers are implemented as cross correlation with an added bias [PyT22], mostly followed by an activation function f. Convolutional layers generalize to higher dimensions and they are invariant to input translations. As described in [DV16], in contrast to a dense layer the output shape of a convolutional layer along every axis is dependent on its

— input size (number of input feature maps),
— number of output feature maps i (number of learned filter kernels)
— kernel size,

— stride,
— zero-padding,
— dilation.

The *stride* of a convolution is the hop size between two consecutive kernel positions. [6] This can be viewed as a form of subsampling. *Zero-padding* refers to the number of concatenated zeros at the beginning and the end of an axis. Zero-padding types are commonly *valid* (full zero-padding) and *same* (half zero-padding). A dilation spreads out the kernel elements by skipping input elements to efficiently increase the *receptive field* of a NN using covolutional layers [ODZ+16].

A convolutional neural network (CNN) consists of convolutional layers and dense layers and can be employed for image processing. Additionally it usually includes *pooling layers* and possibly *transposed convolutional layers*. A pooling layer slides a window along its input much like a convolution but calculates a value such as the maximum or the average. Max-pooling layers are very common in combination with convolutional layers to reduce the size of feature maps.

Transposed convolutional layers are commonly used to mirror the operation of convolutional layers in a NN architecture. Based on eq. (2.9) a transposed convolutional layer computes

$$\boldsymbol{y}[i,n] = ((W^T[i])^T \boldsymbol{x}[n] = W[i]\boldsymbol{x}[n]. \tag{2.11}$$

A guide for these layers is given in [DV16].

2.2.3 Recurrent Layer (RNNs)

Conceptual, a recurrent layer which is synonymously to a recurrent neural network (RNN) consists of a NN with a feedback path. Considering a time series $\boldsymbol{x}[n]$ as input to the RNN, such a recurrent relation enables a limited memory regarding the past states of the NN. The *hidden state* $\boldsymbol{h}[n]$ is fed back to form the output or prediction $\boldsymbol{y}[n]$ of the RNN. Therefor, the output is a function of the input and of the last hidden state $\boldsymbol{y}[n] = f(\boldsymbol{x}[n], \boldsymbol{h}[n-1])$. Hence, the hidden state contains information about all past hidden states.

As described in [Mas22] the recurrent relation can be unfolded in time as depicted in Figure 2.5.

At every time step n there are two inputs $\boldsymbol{x}[n]$ and $\boldsymbol{h}[n-1]$ to the NN, and two outputs $\boldsymbol{y}[n]$ and $\boldsymbol{h}[n]$ from the NN. The internal structure of the NN is depicted in Figure 2.6 as a block diagram.

There are three linear layers (linear transformations) depicted as weight matrices W_{xh}, W_{hh} and W_{hy}. The vectorized $tanh()$ acts as activation function. At every time step n,

6. This refers to the visualization of a 2D convolution operation as the consecutive application of a kernel on its input by element-wise multiplication of the kernel weights with the current receptive field of the input. For a rich set of visualizations see [DV16].

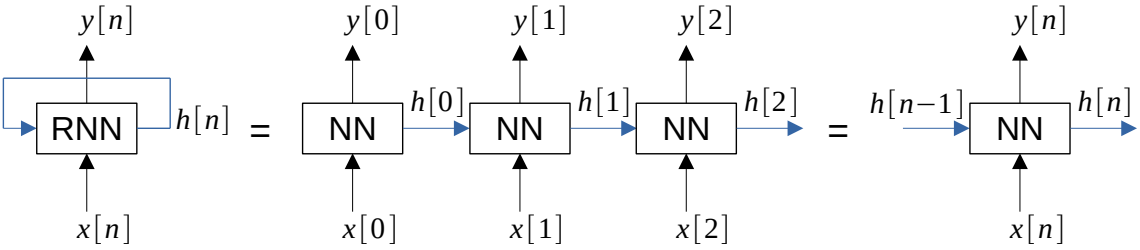


Figure 2.5 – Recurrent neural network feedback unfolded in time.

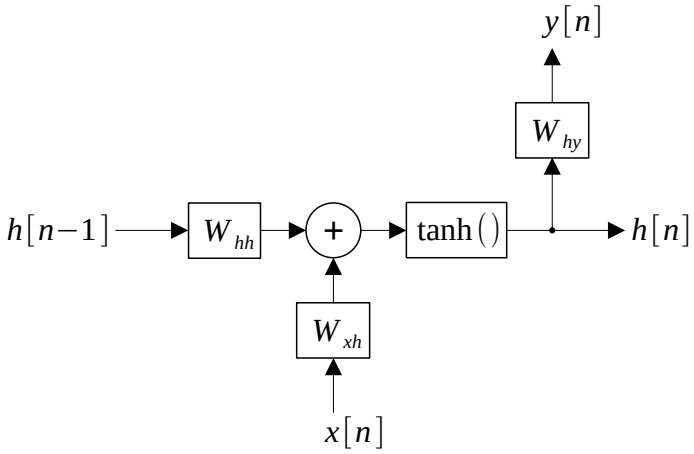


Figure 2.6 – RNN internals.

first the internal state is updated with the inputs

$$\boldsymbol{h}[n] = tanh(W_{hh}^T \boldsymbol{h}[n-1] + W_{xh}^T \boldsymbol{x}[n]) \tag{2.12}$$

and then the output

$$\boldsymbol{y}[n] = W_{hy}^T \boldsymbol{h}[n] \tag{2.13}$$

is calculated.

RNNs are trained via *backpropagation through time* by unfolding the RNN like in Figure 2.5 and applying the error backpropagation algorithm. Unlike the feedforward structures of the dense layers and the convolutional layers, RNNs pose a limitation regarding parallelization because of the inherent feedback structure. Furthermore, the basic RNN described above can suffer from poor learning behavior due to effects regarding deep neural networks (dying/exploding gradient) and its memory often does not reach far enough into the past.

To cure these disadvantages the *gated* RNNs *long short time memory* (LSTM) and *gated recurrent unit* (GRU) have been developed and are usually employed when RNNs are needed. However, in the past years the *transformer* architecture [VSP+17] gradually replaced feedback RNNs and its variants in many applications. This NN architecture uses

attention mechanisms and has no feedback path which makes transformers parallelizable in contrast to RNN variants.

The RNNs described above process input sequences in an unidirectional fashion. Hence current outputs $y[n]$ are exclusively influenced by past sequence elements $x[n-i]$. In non-realtime (offline) applications future sequence elements $x[n+i]$ are available which can also be used to form the current output $y[n]$. RNNs that process the input sequence in both directions are called *bidirectional RNNs* [SP97].

Bidirectional RNNs have two separate layers to process the different directions. One layer W_{hhf} processes the forward direction and the other layer W_{hhb} processes the backward direction. Thereby a forward hidden state $h_f[n]$ and a backward hidden state $h_b[n]$ exist which are both used to compute the output $y[n]$ which is now a function of the whole input sequence with both past and future samples.

2.3 Training Deep Neural Networks

In the past, multiple techniques and strategies had to be developed to solve the main problems that occur when training DNNs. These are

— the *vanishing/exploding gradients* problem,
— underfitting and
— overfitting.

As mentioned above, training NNs works by employing the gradient descent algorithm using backpropagation. As described in [Gé19, ch.11]

> the backpropagation algorithm works by going from the output layer to the input layer, propagating the error gradient along the way. Once the algorithm has computed the gradient of the cost function with regard to each parameter in the network, it uses these gradients to update each parameter with a Gradient Descent step.
>
> Unfortunately, gradients often get smaller and smaller as the algorithm progresses down to the lower [input] layers. As a result, the Gradient Descent update leaves the lower layer's connection weights virtually unchanged, and training never converges to a good solution. We call this the *vanishing gradients* problem. In some cases, the opposite can happen: the gradients can grow bigger and bigger until layers get insanely large weight updates and the algorithm diverges. This is the *exploding gradients* problem, which surfaces in recurrent neural networks (...). More generally, deep neural networks suffer from unstable gradients; different layers may learn at widely different speeds.

The strategies to solve the vanishing/exploding gradients problem are:

— Using the right *weight initialization* according to the employed activation function (i.e. Glorot initialization).
— Using different *activation functions* like ReLU or its variants (SELU, ELU, leakyReLU) instead of the sigmoid function because they do not saturate for positive numbers.

— Using *batch normalization* or *layer normalization*, which zero-center and normalize the data.
— Using *gradient clipping* with RNNs instead of normalization layers to keep the error gradient inside a defined interval.

Measures against underfitting There are numerous measures to avoid underfitting. If the learning curve did not converge fully, the network should be trained for more epochs. Increasing the models complexity (more layers and/or more parameters per layer) or using a different NN architecture should be considered. Since unlabeled data is commonly much more available than labeled data, *unsupervised pretraining* on an auxiliary task can be done. The lower layers (layers closer to the input layer) are then reused for the original task. Using the trained weights of a NN in a different context is referred to as *transfer learning*. Lower layers of NN architectures learn basic feature extraction and can be reused for similar tasks than that of the trained NN.

Measures against overfitting Supplying enough training data to a reasonable complex model for the given task is the usual way to conquer overfitting. If it is not possible to gather more training data, *data augmentation* techniques can be used to generate more training data from the available data. The models complexity can be reduced, if overfitting still occurs. However, in the most cases, regularization techniques are employed to conquer overfitting. These are

— *early stopping*,
— l_1 and l_2 regularization in the loss function,
— *batch normalization* (also acts like a good regularizer),
— *dropout* (one of the most popular for top layers excluding the output layer).

2.4 Neural Network Architectures

This work focuses on NN architectures for sequence processing such as an audio signal. Language and time series also fall into this category. Even images can be interpreted as sequences. There are three types of sequence processing:

1. many-to-one,
2. one-to-many,
3. many-to-many.

The first one, *many-to-one*, can be used for analysis with a classification or regression output layer. The second type, *one-to-many*, essentially generates a sequence from an input. The third and last one, *many-to-many*, takes an input sequence and transforms it into an output sequence. This third type is used for source separation by taking a mixture signal consisting of multiple sources, processing it with the given NN architecture, and outputting the desired source signals. Many-to-many architectures are also called *sequence-to-sequence* architectures ore *seq2seq* models.

2.4.1 Basic Architectures

Multi Layer Perceptron

In the literature a NN architecture consisting of multiple dense layers is called a *multi layer perceptron*. The perceptron is a historical precursor to the artificial neuron, which uses a step function as activation function f [Gé19, ch.10]. Although modern MLPs use different activation functions, they are still called MLP. Sometimes they even include batch-normalization or layer-normalization layers.

This architecture is universally applicable but does not necessary exploit inherent patterns of the input data, like local spacial features in images or recurring similarities in time series. Multi-dimensional input data has to be *flattened*, that is concatenated in one single dimension, for processing it with a dense layer. This process may destroy important local information which can be harnessed with other layers or architectures.

Furthermore dense layers have a big amount of learnable parameters compared to other layers like a convolutional layer. A dense layer contains a total of $D_{out} \cdot D_{in} + D_{out}$ parameters (weights and biases). For this reason, they are commonly used with inputs of lower dimensionality. Hence, MLPs are often used as output layers for classification architectures or in bottleneck sections (see autoencoder below). For inputs with high dimensionality, different architectures should be considered, or a dimensionality reduction method such as principal component analysis may preprocess the input data.

Convolutional Neural Networks

A convolutional neural network (CNN) is a NN architecture which employs convolutional layers, pooling layers and possibly dense layers. Such architectures can harness local multi dimensional patterns such as found in images, in the 2D case. Lower layers near the input learn low level feature detection, and consecutively towards the output, higher level features can be detected. An image classification CNN may learn to detect lines and edges of different angles in the lower layers. In the middle layers the network detects eyes, noses, mouths and ears. Eventually, the top layers can detect whole faces and their output can be used to classify them in some way.

In contrast to dense layers, convolutional layers can contain lower amounts of learnable parameters. A 1D convolutional layer has a number of $i \cdot M$ weights and a 2D convolutional layer has $i \cdot M_1 \cdot M_2$ weights with the filter kernel lengths M_1 and M_2 along the two dimensions. This means, after the first few layers, the dimensionality of the input can be strongly reduced depending on the layers properties, such as the stride. Pooling layers are used to further decrease the dimensionality in CNNs.

The *WaveNet* architecture [ODZ+16] is an example of a convolutional time series processing network. It uses stacked dilated convolutional layers for an exponential *receptive field* growth per layer. The receptive field of a CNN is the size of the local input data region that can influence an output neuron, analogous to the local receptive field in the visual cortex of the human brain [Gé19, ch.13].

RNN Variants

As described in section 2.2.3 RNNs are a NN layer which itself employs linear layers and therefor can be seen as a NN architecture. A common RNN, a LSTM or a GRU can be employed to process sequences in all three ways mentioned above. They have in common that they discard the last hidden state $h[N-1]$.

Figure 2.7 shows a *many-to-one* RNN which takes a sequence $x[n]$ as input and discards all outputs $y[n]$ but the last one.

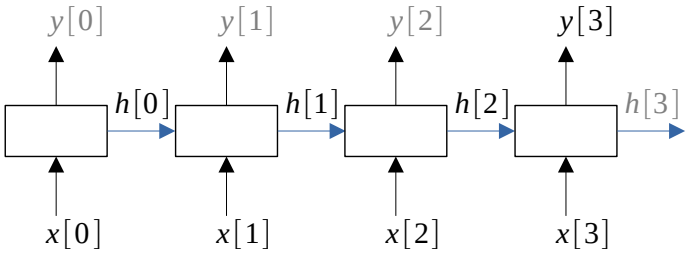


Figure 2.7 – RNN in *many-to-one* configuration as in [Gé19, ch.14]. Grey outputs are discarded.

Figure 2.8 depicts a *one-to-many* RNN which takes one input value $x[0]$ and a series of zeros as input and outputs the sequence $y[n]$.

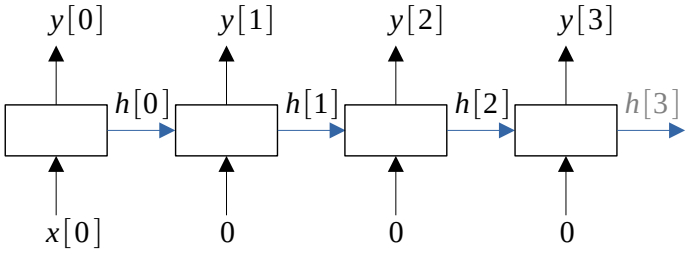


Figure 2.8 – RNN in *one-to-many* configuration as in [Gé19, ch.14]. Grey outputs are discarded.

Lastly, Figure 2.9 shows a sequence-to-sequence RNN, which simply transforms the input sequence $x[n]$ to the output sequence $y[n]$.

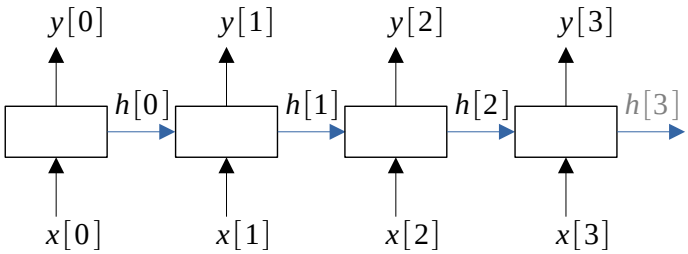


Figure 2.9 – RNN in *sequence-to-sequence* configuration as in [Gé19, ch.14]. Grey outputs are discarded.

Similarly as other architectures, a *deep* RNN consists of multiple consecutive RNN layers.

2.4.2 Architectures for Audio Signal Processing

Architectures for audio signal processing employ generative systems that vary in their audio signal representation. The audio time signal may be used directly but often it is a better decision to preprocess the audio signal to yield audio spectra or spectrograms. Features such as the Mel frequency Cepstral coefficients (MFCC) are also a common choice. These architectures are mostly encoder-decoder (encoder-bottleneck-decoder) or sequence-to-sequence architectures.

Autoencoder

The autoencoder (AE) architecture depicted in Figure 2.10 is a NN that is trained to reconstruct its input $\boldsymbol{x}_n$ at its output $\boldsymbol{y}_n \approx \boldsymbol{x}_n$. This makes it a structure for *unsupervised learning*.

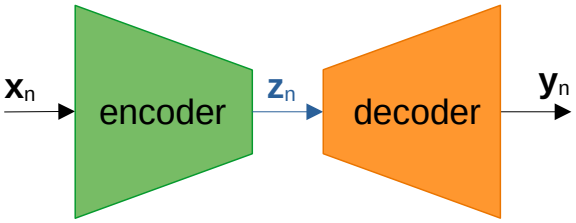


Figure 2.10 – The autoencoder architecture with input $\boldsymbol{x}_n$, latent coding $\boldsymbol{z}_n$ and reconstruction output $\boldsymbol{y}_n$.

While this task may seem trivial for a network with layers with $D_{out} \geq D_{in}$, the encoder of the AE successively decreases the dimensionality of its input with layers with $D_{out} < D_{in}$. The decoder is often a mirrored version of the encoder and restores the original input dimensionality at its output with successive layers with $D_{out} > D_{in}$. The dimensionality reduction forces the encoder to learn to produce a *latent coding* $\boldsymbol{z}_n$ containing the essential information of $\boldsymbol{x}_n$ in such a way, that the decoder is able to reproduce $\boldsymbol{x}_n$ from $\boldsymbol{z}_n$.

An AE can be constructed from any neural layers. This leads to terms like a *dense autoencoder* or a *convolutional autoencoder*. Sometime AE architectures feature *bottleneck layers* between the encoder and the decoder to further process the latent coding. For example, this makes sense to introduce a sequence memory with a RNN in a dense or convolutional AE for sequence processing.

Typical uses of AEs are *dimensionality reduction* or *unsupervised pretraining* [Gé19, ch.15]. In both cases the decoder gets discarded after (pre)training.

A *denoising autoencoder* is trained to produce a noise-free output from its input. This is done by adding noise to the input data and using it as inputs $\tilde{\boldsymbol{x}}_n$. The network is trained to reproduce the original noise free inputs $\boldsymbol{x}_n$.

A *variational autoencoder* is trained like the original AE but produces a latent coding $\boldsymbol{z}_n$ that is used to sample a probability distribution for reproducing the input. After training,

the encoder is discarded and the decoder is used to generate samples, which are similar to the training data, by feeding samples from the probability distribution to its input.

Encoder-Decoder RNN

An *encoder-decoder RNN* depicted in Figure 2.11 in another sequence-to-sequence model. As described in [Gé19, ch.14] it is trained to take an input sequence $x[n]$ and to produce an output sequence $y'[n]$. Unlike the sequence-to-sequence RNN, the encoder-decoder RNN consists of an encoder RNN in many-to-one configuration and a decoder RNN in one-to-many configuration. The outputs of the encoder $y[n]$ are discarded but the last hidden state h_c is used as a *context vector* representing the input sequence. This context vector serves as the hidden starting state of the decoder to produce the output sequence $y'[n]$.

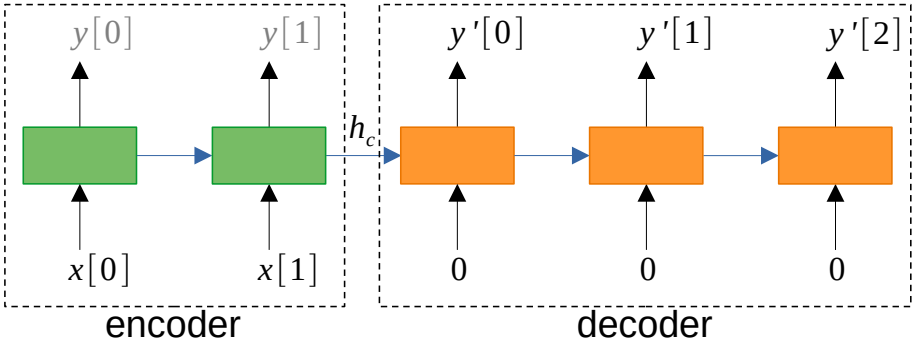


Figure 2.11 – The encoder-decoder RNN architecture with input sequence $x[n]$, context vector h_c and output sequence $y'[n]$. Grey outputs are discarded.

The encoder-decoder RNN can be used to transform a given sequence into a different one, as in machine translation. In this case, the decoder receives the target sequence as input at training and at inference the decoder feeds back its last output to its current input. The difference to an AE is that the encoder-decoder RNN is trained supervised while the AE is normally an architecture for unsupervised learning.

U-Net

The *U-Net* architecture [RFB15] depicted in Figure 2.12 is essentially an AE with *skip connections*. These skip connections can be made by addition or as depicted by concatenation. It is a fully convolutional network using convolutional layers and pooling layers in the encoder on the left side and transposed convolutional layers in the decoder on the right side. In this way the dimensionality is decreased towards the middle and again increased towards the output. The skip connections from the encoder stages to the mirrored decoder stages can preserve details from the input that may get lost by abstraction in a pure AE architecture. Furthermore, they cause gradients to flow directly to lower layers, preventing the vanishing gradients problem.

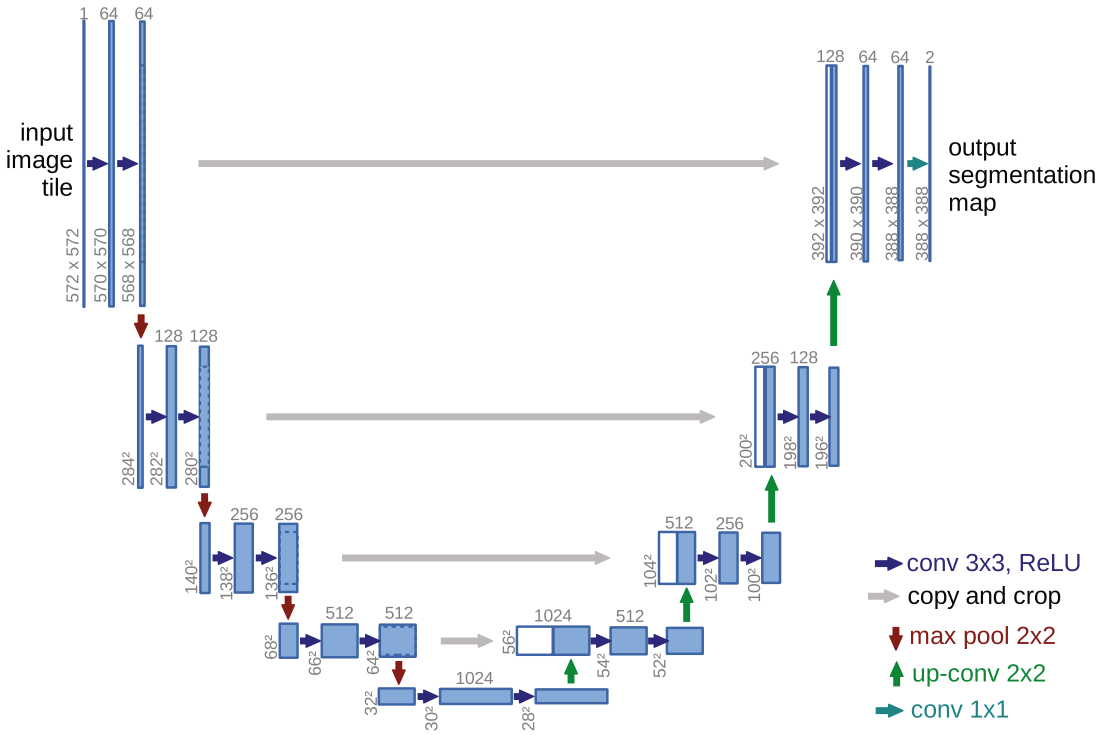


Figure 2.12 – U-Net architecture from [RFB15]. Grey arrows indicate skip connections.

Originally developed as an image processing architecture, the U-Net is a popular choice for music source separation producing state-of-the-art results [MFUS21]. It is also successfully employed in other applications.

Further NN architectures with great potential for audio signal processing and synthesis are *generative adversarial networks* (GANs) [ZXT19, p31], *diffusion models* (which use the U-Net architecture) [HJA20] and the *transformer architecture* [VSP+17]. GANs [KLA+20], [Wan22] and diffusion models [RBL+22] achieve state-of-the-art performance in image generation and the transformer, employed as a large language model [TDFH+22] [BMR+20], achieves state-of-the-art performance in natural language processing tasks.

2.5 Interpretable and Physics-Inspired AI

The literature about *interpretability* and *explainability* of AI systems is highly divers and no general definitions exist for these terms. Different fields use different definitions or even use these terms interchangeably. However considering specifically deep neural networks as in [GBY+18] the terms can be defined as follows:

"The goal of *interpretability* is to describe the internals of a system in a way that is understandable to humans. (...) for a system to be interpretable, it must produce descriptions that are simple enough for a person to understand using a vocabulary that is meaningful to the user" [GBY+18].

Interpretability is tightly linked to *completeness*. "The goal of *completeness* is to describe the operation of a system in an accurate way. (...) When explaining a self-contained computer program such as a deep neural network, a perfectly complete explanation can always be given by revealing all the mathematical operations and parameters in the system" [GBY+18]. Meaningful human understanding together with a complete description in one system representation is not easily manageable for complex systems such as DNNs. Therefor, interpretability and completeness are in a trade-off relationship.

Explainability in terms of data processing answers the question *why* a DNN produces its output from its input and interpretability answers the question *how* it does so. To explain a systems decision, further definitions are possible which are included in [GBY+18].

Neural networks are often described as *universal function approximators*, because they are able to imitate any function given the right weights and biases [Bis06]. This feature is good for generality. However, looking at the approximation of the solution towards a machine-learning problem, the search space is huge. For this reason, big amounts of data are needed to train ordinary DNNs, to solve sophisticated problems.

Considering training data produced by a physical process such as the recording of a musical instrument, the NN may be in some way *inspired* with a model of this physical process to produce its output. If it is possible to integrate a physical model of the source of the training data, then the output of the NN is vastly constrained and the search space, therefor, is strongly reduced. By introducing a physical model into a NN which is capable of synthesizing outputs according to the available training data, the data demand is strongly reduced. The NN can then be referred to as being *physics-inspired AI*.

Chapter 3

Music Source Separation

In natural environments often multiple signal sources are present, which are combined in a linear medium such as air. Measuring the physical quantity at a point in proximity of these sources results in a mixture signal that consists of the sum of the individual source signals. *Source separation* is the reversal of this effect by extracting individual source signals from the observed mixture signal [PLV+19].

In the case of acoustic sound sources these scenarios are experienced in everyday life through listening to human speakers, musical instruments or environmental sounds. Understanding a single person in a conversation with multiple simultaneously talking people and present ambient noise is referred to as the cocktail party problem. Source separation is then the equivalent of solving this problem by extracting the person's voice from the overall perceived sound scene. A similar problem is to aurally perceive individual instruments distinctly at a concert or while listening to recorded music.

While forming the sum of multiple source signals is a simple mathematical operation, the source extraction from the mixture signal is in general not trivial. The more similarities present sources share, the harder it gets to separate them.

> Regarding the different audio domains, in speech it is assumed that the signal is sparse and that different sources are independent from each other. In environmental sounds, independence can usually be assumed. In music there is a high dependence between simultaneous sources as well as there are specific temporal dependencies across time, in the waveform as well as regarding long-term structural repetitions. [PLV+19, p.9]

The goal of *music source separation* (MSS) is to extract individual instrument signals from a musical mixture signal. A popular task in the MSS community is to separate pop songs into the four tracks: vocals, drums, bass and other [MFUS21]. [1] The difficulty of MSS depends on the source type to be separated from the mix. Separation of *uncorrelated sources* such as voice and drums usually is an easier problem than the separation of *correlated sources* such as the individual strings of a guitar.

1. The track referred to as *other* consists of all sounds which are not vocals, drums or bass.

3.1 MSS in the Time Domain

Following [PLV+19] the source separation process in the time domain is described as extracting the *true source signals* $s_{c,i}[n]$ from the *mixture signal* $x_c[n]$. As described above, the mixture signal

$$x_c[n] = \sum_{i=1}^{I} s_{c,i}[n] \tag{3.1}$$

corresponds to the sum of the individual sources with the time sample index n, the source index i, the number of sources I and the channel index c. Since multiple microphones may be used for recording, multiple channels c may be present. The mixture signal $x_c[n]$ is referred to shortly as *mix signal* or just *mix*.

MSS methods that work strictly in the time domain are called *waveform methods* since their input $x_c[n]$ and output $s_{c,i}[n]$ are time domain signals. Another signal representation rarely used in MSS methods is called *complex-as-channels* (CaC) which was proposed by [CKCJ21].

3.2 MSS in the Frequency Domain

Spectrogram methods use a time-frequency transform such as the *short-time Fourier transform* (STFT) for signal representation. In this case a spectrogram method can either synthesize source estimate spectrograms directly (*direct synthesis methods*) or it can estimate a separation mask which is applied to the mix spectrogram, yielding the estimated source spectrogram (*masking methods*).

Source separation in the time-frequency domain is often done via estimating a time-frequency mask. This mask is then used to extract one source from the magnitude spectrogram of the mix signal. Without a corresponding phase, the masked magnitude spectrograms cannot be transformed back into the time domain. In most masking methods for MSS the mixture signal's phase is used to transform the masked spectrograms back into the time domain to yield the extracted time domain signal of the source instrument. This method is widely used and produces results with good sound quality. [2]

The masking operation is an element-wise multiplication $\odot$ (which is also referred to as the *Hadamard product*) in the time-frequency domain of the mixture spectrogram $X_c[k, m] = STFT\{x_c[n]\}$ and the estimated separation mask $\hat{M}_{c,i}[k, m]$ with frequency bin k and time frame index m and results in the estimated source signal

$$\hat{S}_{c,i}[k, m] = X_c[k, m] \odot \hat{M}_{c,i}[k, m]. \tag{3.2}$$

2. Additionally there are phase estimation algorithms like the Griffin-Lim algorithm and its successors to estimate the phase from the magnitude spectrogram.

$\hat{S}_{c,i}[k,m]$ implicitly contains the phase of the complex mix signal $X_c[k,m] \in \mathbb{C}$, because the separation mask has exclusively real elements $\hat{M}_{c,i}[k,m] \in \mathbb{R}$. As notated above spectrograms are commonly obtained via the STFT, because it is efficient and has a perfect inverse transform. Other transforms such as the constant-Q-transform may offer different advantages and disadvantages than the STFT.

3.3 Single Channel Music Source Separation

The general problem with $c > 1$ outlined above is called *multi channel* MSS. *Single channel* MSS uses only one channel $c = 1$ to represent audio material. For single channel masking methods, eq. (3.2) becomes

$$\hat{S}_i[k,m] = X[k,m] \odot \hat{M}_i[k,m]. \tag{3.3}$$

A spectral mask in general (estimated or not) $M_i[k,m]$ can either be a *binary mask* $M_i[k,m] \in \{0,1\}$ or a *soft mask* $M_i[k,m] \in [0,1]$ which is also called a *ratio mask* [VVG18, ch.5]. The best mask an estimation algorithm can obtain is called *oracle mask* or *ideal ratio mask* (IRM) $M_i^{IRM}[k,m]$ in the case of a ratio mask. The IRM acts as an upper bound for the performance of time-frequency domain MSS methods and is obtained from the true source spectrogram $S_i[k,m]$ (e.g. a separately recorded instrument without mic bleed) via

$$M_i^{IRM}[k,m] = \frac{|S_i[k,m]|}{|X[k,m]| + \epsilon} \tag{3.4}$$

with a small ϵ to avoid division by zero [LS19]. The division in eq. (3.4) indicates an element-wise division and is not to be confused with a multiplication of the numerator with the inverse matrix of the denominator.

In this sense, the goal of masking methods is to estimate source masks $\hat{M}_i[k,m]$ that are as close as possible to the IRM $M_i^{IRM}[k,m]$.

3.4 Current Methods

"Many approaches have been researched in the field of music [source] separation such as local Gaussian modelling, non-negative matrix factorization, kernel additive modelling and hybrid methods combining these approaches" [SUTM21]. State-of-the-art music source separation methods employ neural networks exclusively. Deep learning based approaches outperform traditional signal processing methods and the separation quality of such approaches continue to improve [MFUS21].

Many state-of-the-art methods use the time-frequency approach presented in eq. (3.2) since the "structure of natural sounds is more prominent in the time-frequency domain"

than in the time domain, "natural sound sources are *sparse* in the time-frequency domain which facilitates their separation" and convolution operations are done with a simple multiplication in the time-frequency domain [PLV+19, p.8]. However, there is a growing trend in the top performing methods such as [Déf21], [RMD22] and [KCC+21] to employ both representations, the time domain signal and its time-frequency transform. According to Stöter and Uhlich further current trends in audio source separation [SU20] among others are

— the move from supervised to universal source separation,
— dealing with imperfect training data,
— and employing perceptual loss functions.

In August 2021 the music demixing challenge (MDX2021) took place "where the task" was "to separate stereo songs into four instrument stems (Vocals, Drums, Bass, Other)" [MFUS21]. Participants where ranked in two leader-boards which differed in training set size. The top methods of leader-board A which "focused on establishing a fair comparison between models to highlight which one performs best" are:

1. Demucs version 2 [Déf21],
2. KUIELab-MDX-Net [KCC+21],
3. a closed source approach, and
4. a network blending model using X-UMX [SUTM21], a U-Net similar to Spleeter [HKVM20] and a modification of Demucs [DUBB19].

Open-Unmix [SULM19] and CrossNet-Open-Unmix (also called Open-Unmix improved) [SUTM21] were used as baseline methods. The sound demixing challenge 2023 [3] took place at the time of writing from January to March 2023 [Son23].

Performance measurement of MSS methods is commonly done with the *signal to distortion ratio* (SDR) [VGF06] or with its revision, the *scale invariant signal to distortion ratio* (SI-SDR) [LRWEH19]. The SDR was used in MDX2021 [MFUS21] and is also currently employed in SDX2023 [Son23] to rank MSS methods' performance despite the critique in [LRWEH19].

As described in [UPG+17], *network blending* or *network fusion* is the process of linearly combining multiple outputs of MSS systems (taking "the weighted average for each estimated source" [KCC+21, p.4]) and then performing a multi-channel Wiener filter (MWF) post-processing to achieve better results than with the individual outputs. The MWF "ensures that the sum of the four estimates gives the original mixture", considerably reducing artifacts, and hence improving separation performance. Further information on the statistical signal processing of the MWF is given in [UPG+17, p.262].

In the following sections MSS baseline methods and state-of-the-art methods are described. This gives an overview of the field of MSS and also gives insight in the use of deep learning in audio signal processing.

3. `https://www.aicrowd.com/challenges/sound-demixing-challenge-2023`

Open-Unmix

In the preceding work [UPG+17] to Open-Unmix, the authors propose a MSS approach via a feed-forward DNN with MWF post-processing. The use of a bidirectional LSTM trained with data augmentation has been investigated as well which resulted in better model generalization. Data augmentation allowed to avoid overfitting effectively. Further improvement has been achieved with the network blending of the trained feed-forward DNN and bidirectional LSTM.

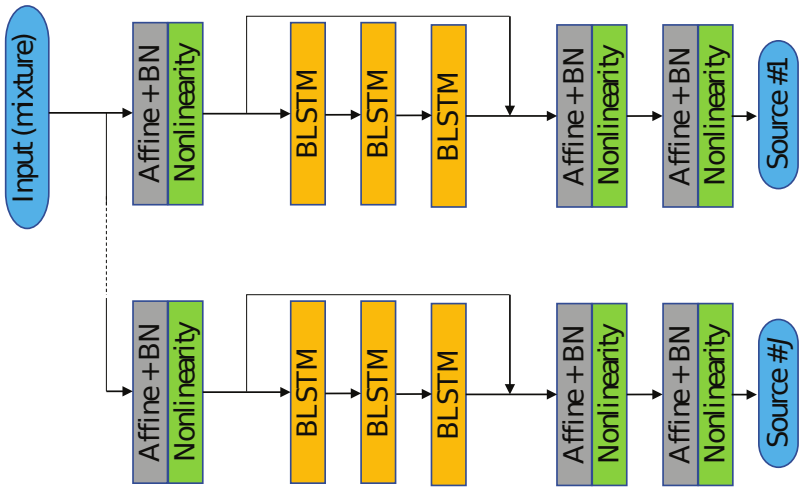


Figure 3.1 – Open-Unmix NN architecture [SULM19] from [SUTM21]. The gray *affine + BN* layer describes a linear layer with a subsequent batch normalization layer. These two layers are followed by a non-linear activation function in green. The gray and green layer effectively form a dense layer with batch normalization. Inputs and outputs are audio spectrograms.

Open-Unmix[4] (UMX) is titled "a reference implementation for music source separation" [SULM19] based on the bidirectional LSTM model described above from [UPG+17]. The UMX architecture is depicted in Figure 3.1. It consists of a dense layer with batch normalization followed by three bidirectional LSTMs that are bypassed with a skip connection and two additional dense layers with batch normalization.

For every source type a separate instrument-specific model is trained. Every UMX instrument model receives a mix spectrogram $X_c[k, m]$ as input and outputs the corresponding separated instrument prediction spectrogram $\hat{S}_{c,i}[k, m]$. Training is done supervised using the MSE loss between predicted sources $\hat{S}_{c,i}[k, m]$ and target sources $S_{c,i}[k, m]$.

In [SUTM21] an improved version of UMX is proposed named *CrossNet-UMX* (X-UMX) using a new *multi domain loss* and a new *combination loss* for *bridging network paths* of all instrument models. This multi domain loss is calculated from both the time-frequency domain output as well as from its inverse transform into the time domain.

As depicted in the X-UMX architecture in Figure 3.2 the *network bridging* is done for "information sharing among sources" [SUTM21] by jointly training the whole network

4. `https://github.com/sigsep/open-unmix-pytorch`

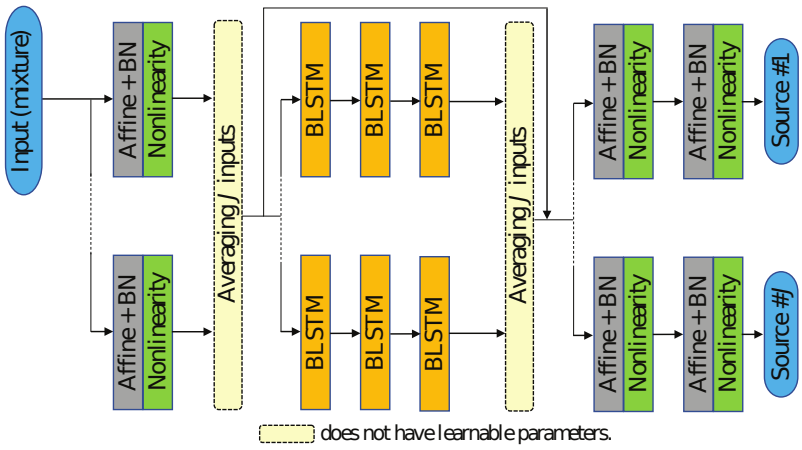


Figure 3.2 – CrossNet Open-Unmix NN architecture from [SUTM21]. The gray *affine + BN* layer describes a linear layer with a subsequent batch normalization layer. These two layers are followed by a non-linear activation function in green. The gray and green layer effectively form a dense layer with batch normalization. Inputs and outputs are audio spectrograms.

on all sources. This is in contrast to the UMX architecture in Figure 3.1 where every source model is trained independently from the others. The authors show that the bridging of all separation models is beneficial and "performance improvements can be gained for most DNN-based source separation methods with introducing almost no additional computational costs at inference time" [SUTM21].

Demucs

The *Demucs v2* waveform/spectrogram hybrid architecture [Déf21] is the successor of the *Demucs* waveform architecture [DUBB19]. As depicted in Figure 3.3 it extends a U-Net architecture [RFB15] with 2 parallel branches. One branch processes the time domain signal (T) and the other branch processes the time-frequency domain signal (Z) [Déf21].

The encoder blocks consist of two 1D convolutional layers with two residual branches in between. Two different activation functions are employed after convolutional layers, the gated linear unit (GLU) [DFAG17] and the Gaussian error linear unit [HG16]. In the time domain branch 1D-convolutions are applied along the time dimension and in the time-frequency domain branch the 1D-convolutions are applied along the frequency dimension. A frequency embedding is injected between the first and second time-frequency domain encoder blocks. The residual branches are "composed of dilated convolutions, and for the innermost [blocks], a biLSTM with limited span and local attention" [Déf21, p.5]. Further details are found in [Déf21] and [DUBB19].

The outputs of the fifth encoder blocks in both branches have the same dimensions. These outputs are added and then encoded to the latent coding with a shared encoder block. The decoder blocks are symmetrical to the encoder blocks and decoder block outputs have the same dimensions as the respective encoder block inputs. After the last decoder blocks

the time-frequency domain prediction is transformed into the time domain and added to the time domain prediction. This sum serves as the final network output which is the predicted source signal from the mix input.

Another advancement of Demucs has been achieved in [RMD22] where the Demucs v2 architecture [Déf21] was updated. The "innermost layers are replaced by a cross-domain Transformer Encoder, using self-attention within one domain, and cross-attention across domains" [RMD22]. This model shows slightly better performance compared to Demucs v2 when trained on MUSDB [RLS+17] and 800 extra songs but performs poorly when trained exclusively on the MUSDB dataset. This method counts to the top performing MSS methods according to [Pap23].

KUIELab-MDX-Net

The *KUIELab-MDX-Net* MSS method employs 3 different NNs:

- a pretrained Demucs v1 [DUBB19],
- a TFC-TDF-U-Net v2 and
- a mixer model.

As depicted in Figure 3.4, one TFC-TDF-U-Net v2 model is trained for every source type (vocals, drums, bass and other). The TFC-TDF-U-Net v2 source predictions together with the mix signal are fed to the mixer model which consists of a single convolutional layer to "further enhance the *independently* estimated sources" [KCC+21]. The mixer model predictions and the Demucs predictions are blended (see network blending above) to form the overall predicted source signals. This architecture blends a time domain model (Demucs v1) with time-frequency domain models (TFC-TDF-U-Net v2).

Figure 3.5 shows the *TFC-TDF-U-Net v2* NN architecture which also indicates its training procedure. This time-frequency domain model receives a mix spectrogram cropped above the target source type frequency range and outputs the corresponding source prediction spectrogram. The output spectrogram gets zero padded to the input FFT size and is transformed back into the time domain. Model training is done supervised with a L1-loss function.

The TFC-TDF-U-Net v2 architecture is based on the U-Net architecture [RFB15]. Three skip connections are implemented by element-wise multiplications instead of concatenation, as in the original U-Net. As its predecessor TFC-TDF-U-Net v1 [CKC+20] and [CKCJ21], this architecture is based on *time frequency convolutions with time-distributed fully connected networks* (TFC-TDF) which were introduced in [CKC+20]. A TFC-TDF block consists of a small NN architecture employing a *time distributed fully-connected network* (TDF) and a *time-frequency convolution* (TFC). Further details are found in [CKC+20].

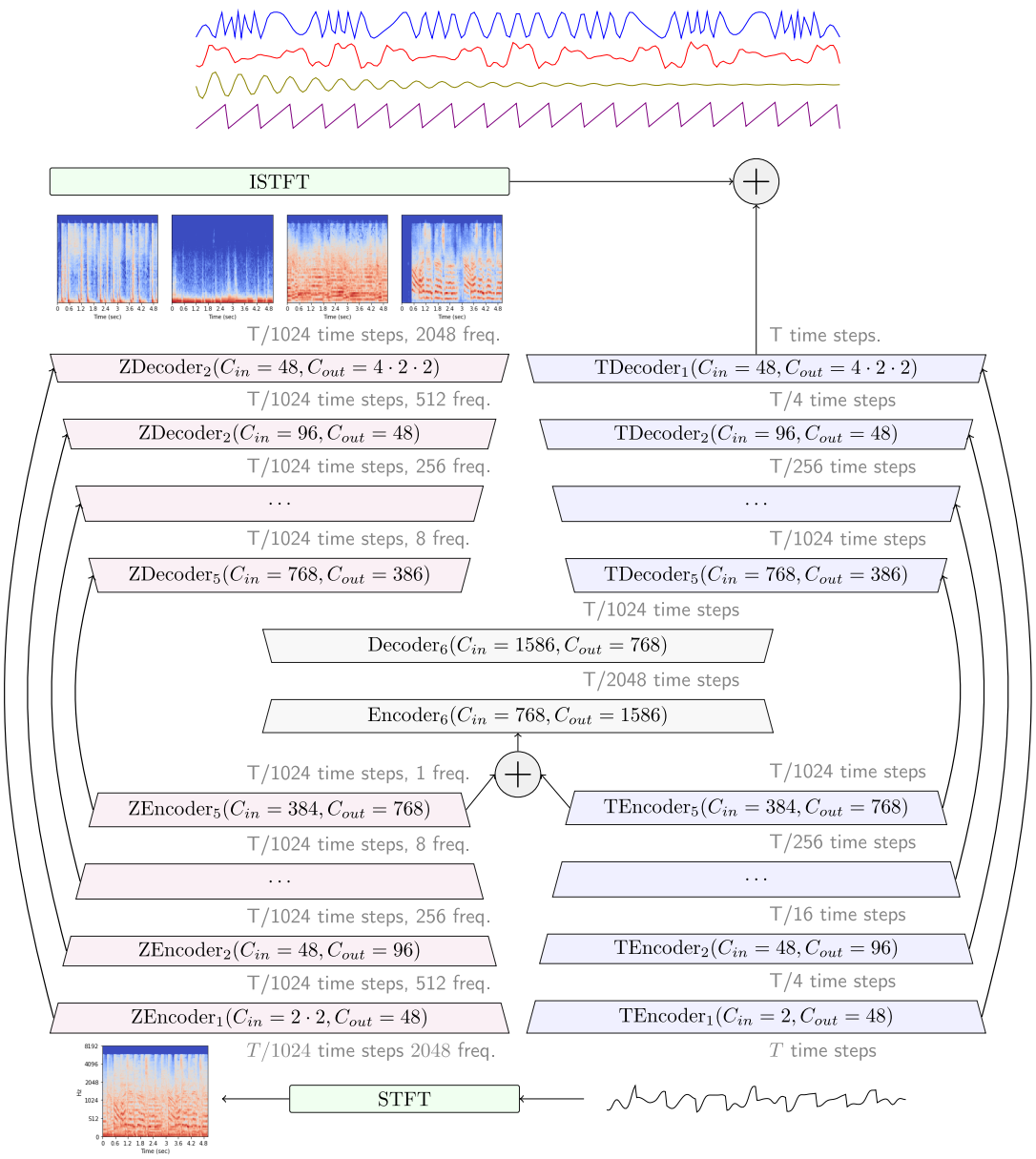


Figure 3.3 – Demucs v2 NN architecture from [Déf21]. T denotes the time-domain branch and Z denotes the time-frequency domain branch. Skip connections as in U-Net are depicted as bent arrows at the left/right side of the encoders/decoders from encoder layers to decoder layers.

Other Relevant Methods

Band-split RNN is "a frequency-domain model that explicitly splits the spectrogram of the mixture into subbands and perform[s] interleaved band-level and sequence-level modeling" [LY22]. A priori or expert knowledge can be used to determine hyperparameters such as the subband bandwidths of the model to increase performance for specific source types. Furthermore a semi-supervised fine tuning pipeline has been proposed to further

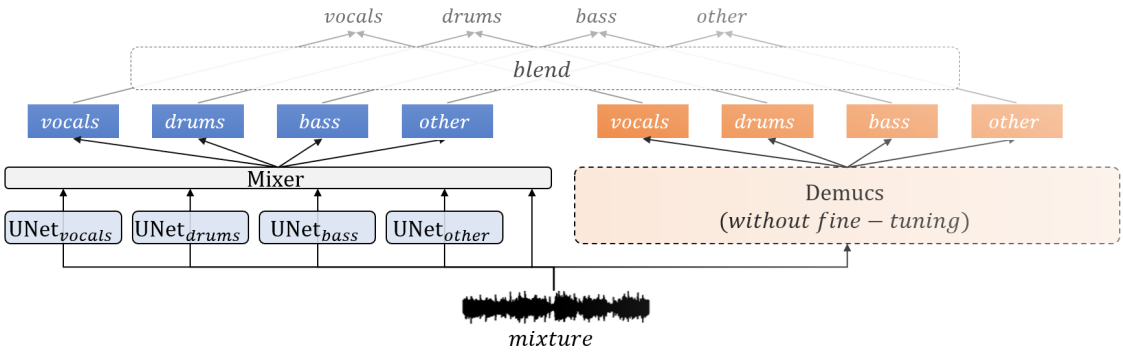


Figure 3.4 – The overall architecture of the KUIELab-MDX-Net from [KCC+21]. The blue blocks labeled with *UNet* represent a TFC-TDF-U-Net v2 model for every source type. The block labeled with *blend* indicates network blending.

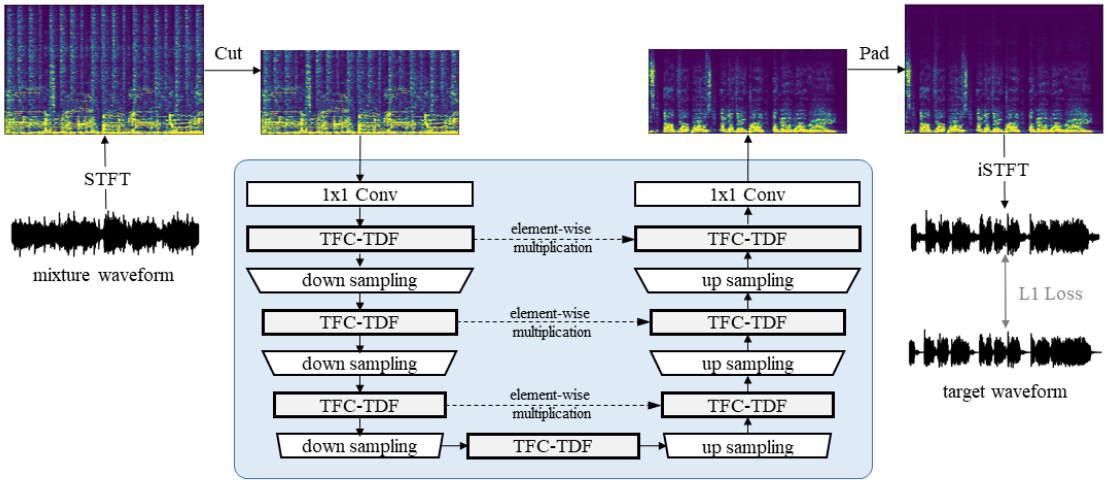


Figure 3.5 – The time-frequency domain NN architecture TFC-TDF-U-Net v2 out of the KUIELab-MDX-Net from [KCC+21].

increase MSS performance. This model (together with the advancement of Demucs v2 [RMD22]) counts to state-of-the-art methods according to [Pap23].

Previous to the MDX2021 the *D3NET* [TM21] architecture produced state-of-the-art performance in MSS. It is described as a "densely connected dilated DenseNet" [TM21] which is "based on dilated convolutions connected with dense skip connections" [Déf21, p.2]. In [CKCJ21] the *LaSAFT* architecture was introduced, along with the *Complex-as-Channels* (CaC) representation which differs from masking and waveform methods. Additionally the authors proposed a MSS architecture based on the conditioned U-Net [MBP19] which employs FiLM layers [PSdV+18] to manipulate a U-Net architecture with additional input data. They also proposed an attention-based frequency transformation block called LaSAFT and an extension to FiLM called GPoCM.

Like X-UMX [5] [SL23], *Spleeter* [HKVM20] is another MSS method available as open source implementation [6] [dee23a]. Similar to other methods it uses a U-Net architecture and is trained on a large body of closed source data from the streaming platform deezer [7] [dee23b].

Sams-Net is a time-frequency domain MSS method employing an attention based NN architecture [LCHL21]. In [SHG21] conditioning techniques with video material regarding a conditioned U-Net [MBP19] were explored for the task of classical music source separation also containing correlated sources. In [MDS21] a method for learning a music signal representation is proposed and compared to the STFT. It employs a denoising autoencoder model including differentiable digital signal processing [EHGR20] which is trained unsupervised to reconstruct the singing voice. Furthermore the model is used for singing voice separation with binary masking. In [SFDRB22] a NN architecture containing differentiable digital signal processing as well is trained unsupervised to perform MSS on vocal ensemble recordings. This method is described in more detail in chapter 4.

5. https://github.com/sigsep/open-unmix-pytorch
6. https://github.com/deezer/spleeter
7. https://www.deezer.com/

Chapter 4

Differentiable Digital Signal Processing

This chapter gives a detailed overview on the topic of *differentiable digital signal processing* (DDSP). DDSP is a technique which was first proposed in [EHGR20] and is generally employed to include signal processing techniques directly inside NNs. The authors of [EHGR20] introduced the DDSP TensorFlow library but DDSP is rather a concept than a specific implementation. However, this introduction laid the groundwork for a series of publications with increasing numbers over time.

The DDSP concept enables the use of traditional signal processing blocks such as filters or oscillators in NN architectures. Every calculation to form the output of a NN has to be differentiable, since NNs are trained with gradient descent methods. Hence the name *differentiable* DSP. The forward pass is defined by the mathematical definition of the desired signal processing system. Then, the backward pass is handled by the employed automatic differentiation systems or machine learning libraries such as TensorFlow or PyTorch. In this way, gradients can flow through the DDSP part and the whole system can be trained end-to-end.

Currently DDSP is exclusively employed for neural audio signal processing, although there are no restrictions to this field. This is most likely due to the fact that it was proposed in this field. As in traditional audio signal processing, applications can be divided into *audio synthesis* and *audio effects*. Other neural audio techniques such as WaveNet [ODZ+16] or SampleRNN [MKG+16] employ standard neural layers such as convolutional layers or RNNs *directly* to process or synthesize audio signals. In general, systems employing DDSP, predict control signals with standard neural layers for controlling the DDSP processing or synthesis part of the network.

A significant difference between these approaches lies in the size of the output space of the NN. While a generic NN (e.g. an MLP) imposes no restrictions on the form of its output signal, a NN employing DDSP *encodes expert knowledge into the system*, restricting its output to the possible outputs of the signal processing part. In this way, the NN with DDSP is already informed of the desired output signals' space before training, whereas the generic NN has to infer the entire context from the training data. The better the DDSP

part of a NN is able to model the training data, the less training data and training time is needed for reaching a certain performance regarding the optimization goal.

Looking at the advantages and disadvantages regarding the employment of DDSP results in a trade-off as in many areas of engineering. A clear advantage of the use of DDSP is the potential to reduce the required training data, training cost and training time to reach a given objective with a NN. Furthermore in many cases it is too costly or even impossible to gather more training data to achieve a solution with a certain quality.

A disadvantage on the other hand is the requirement of expertise for encoding the expert knowledge into the system. The output of the DDSP model has to mirror the training data in such a way, that the entire system is able to effectively learn from it. If the DDSP model deviates too far from the process, which generated the training data, or the NN is not able to predict reasonable control signals for an appropriate DDSP model, the trained model may show poor performance.

For this reason it is very important to have detailed knowledge of

1. the process, which produced the training data,
2. how to model the process mathematically, and
3. how to control the DDSP model in a stable but also capable way

for producing good predictions. Casually this may be expressed as: 'You got to have the expertise first before encoding it into a neural network.' Additionally a DDSP model may restrict the NN output too strong for proper generalization, regarding a specific set of training data. In this case, a subset of the training data may be modeled effectively by the DDSP model but the complete set exceeds its capabilities.

This means, the trade-off lies between systems, which use

— big data, very deep NNs with big amounts of parameters, little expert knowledge requiring little expertise in the field and possibly short development time by using general purpose architectures, or
— little data, shallow NNs with low amounts of parameters, detailed expert knowledge for developing an appropriate specialized signal model that is able to model the data effectively.

Most signal processing systems are explainable because humans associate meaning with their parameters, inputs and outputs and the causality between inputs and outputs is clear. For this reason, the control signals which are the output of a NN and the input to the DDSP model are at least interpretable. Many generative NN architectures feature a variation of a *latent coding* such as an autoencoder, which is not interpretable. Such an interpretable latent coding may be not only a problem for debugging but also a design requirement of the system for some reason. DDSP enables such interpretable latent codings.

4.1 Audio Applications of DDSP

DDSP literature can be broadly categorized into two groups:

1. *sound (re-)synthesis*, and
2. *audio effects*.

Systems performing neural *sound (re-)synthesis* with DDSP are trained to synthesize one or a number of audio signals from an input audio signal, from input controls or from both. Applications of such systems are *source separation*, *deconvolution*, *timbre transfer* or *speech synthesis*.

The task of separating musical sources is handled in detail in chapter 3. Deconvolution refers to the inverse operation of convolution, yielding the unprocessed input signal to a convolution. Timbre transfer applies the timbre of one audio signal to another audio signal, not interfering with pitch and dynamics but adapting playing style. Therefore timbre transfer is more sophisticated than a vocoder effect which applies the spectral envelope (filter) of one signal to another signal (source) and ignores idiosyncrasies of the given instrument.

Neural audio effects are employed in *intelligent music production* [DM19] [MS19] or are trained to perform other audio tasks such as *virtual analog modeling*. In intelligent music production, neural audio effects perform *automatic mixing* or *automatic mastering* including *style transfer*. DDSP is one approach of a couple existing methods to build such neural audio effects.

According to [DM19] intelligent music production systems are in active research for developing *better music production metering and diagnostics*, *more intuitive interfaces and controls*, *intelligent music production assistants*, *fully autonomous agents for different audio tasks*, and *improved interactive audio* (as in virtual reality). These systems are developed for audio engineers but also for the layman to achieve faster or better results in music production.

4.1.1 Neural Sound (Re-)Synthesis via DDSP

Generative audio systems can be designed with common generative NN architectures such as the autoencoder or the GAN. Current systems for sound (re-)synthesis employing DDSP have the overall structure of an autoencoder, although there are no restrictions on the system architecture. The encoder is built with standard neural layers and the decoder consists of either just a DDSP signal model or the DDSP signal model with a preceding neural block. These systems are optimized to synthesize sound according to the training data for performing a variety of different audio tasks.

The Origin

Engel et al., associated with Google Brain [1] [Goo23d] and the Google Magenta project [2] [Goo23c], proposed DDSP first in [EHGR20]. This publication serves as the basis for

1. `https://research.google/teams/brain/`
2. `https://magenta.tensorflow.org/`

all following DDSP literature. Additionally to the DDSP concept, the authors presented the *DDSP Autoencoder* which is a NN architecture trained unsupervised to resynthesize recordings of a specific musical instrument with a harmonic plus noise signal model. Timbre transfer has successfully been achieved and by adding a reverb module into the signal model of the DDSP Autoencoder they also achieved deconvolution. An important part for their success with such a system was to employ the multi scale spectral loss function for training. This publication is presented below in more detail.

DDSP was then used by Engel et al. for pitch tracking by self-supervised inverse audio synthesis [ESH+20]. In *MIDI-DDSP* "detailed control of musical performance" [WMD+21] has been studied with a musical hierarchy on the levels of notes, performance and synthesis. The authors claim, that the method "can reconstruct high-fidelity audio, accurately predict performance attributes for a note sequence, independently manipulate the attributes of a given performance, and as a complete system, generate realistic audio from a novel note sequence" [WMD+21].

Sound Synthesis Techniques

With an established understanding how to incorporate traditional DSP techniques into neural networks, many major sound synthesis techniques have been studied. These are

- additive synthesis,
- subtractive synthesis,
- frequency modulation,
- waveshaping, and
- wavetable synthesis.

The DDSP Autoencoder [EHGR20] already featured a *harmonic plus noise model*, which is a very general signal model capable of synthesizing an extremely wide range of sounds. On the other hand, such a generic source model does not pose as many restrictions on the output space of the overall NN like a more restrictive sound synthesis method. The DDSP Autoencoder signal model can be described as a combination of a restricted form of additive synthesis and subtractive synthesis.

Subtractive synthesis was studied in [MS21] with the application of synthesizer sound matching with DDSP. The authors trained a NN containing a basic DDSP subtractive synthesizer to recreate real-world sounds. They considered a two phase training scheme by first pre-training the system to recreate synthetic sounds with known parameters employing a parameter loss. Then the system is fine-tuned by training it to minimize a spectral loss regarding real-world sounds. In a subjective evaluation they found, "that the proposed method finds better matches compared to baseline models" [MS21].

Neural waveshaping synthesis was explored in [HSF21] by introducing the *neural waveshaping unit* (NEWT). The authors compiled a signal model consisting of the NEWT, a DDSP "noise synthesizer and reverb and found it capable of generating realistic musical instrument performances with only 260k total model parameters, conditioned on F0 and loudness features." [HSF21] The proposed method performed competitively around baseline methods in terms of sound quality but significantly outperformed them in generation

speed achieving real-time capability.

Differentiable wavetable synthesis has been studied in [SHC+22] by learning a "dictionary of one-period waveforms i.e. wavetables, through end-to-end training" [SHC+22]. The authors claim to achieve high-fidelity sound synthesis by only using a dictionary of 10 to 20 wavetables and to achieve real-time capability with their proposed wavetable synthesis method. They also give application examples such as high quality pitch-shifting and data-efficient neural audio sampling.

Frequency modulation (FM) synthesis was studied in [CMS22] by first exploring design constraints to train a DDSP FM synthesizer. The authors subsequently presented "Differentiable DX7 (DDX7), a lightweight architecture for neural FM resynthesis of musical instrument sounds in terms of a compact set of parameters." [CMS22] Their work takes "steps towards enabling continuous control of a well-established FM synthesis architecture from an audio input." The DDX7 with a subsequent DDSP reverb was trained to resynthesize instruments performing classical music. According to the authors the features of this method are the already established interface of FM synthesis for sound design, the use of a small model with 6 oscillators or less and its possible on-the-fly timbre manipulation and real-time capability.

In [RMR22] "a *polyphonic differentiable model for piano* sound synthesis, conditioned on Musical Instrument Digital Interface (MIDI) inputs" was proposed. The inputs to this NN architecture employing DDSP are pitch, velocity, a piano ID, pedal data and the target audio. Similar to the harmonic plus noise signal model with subsequent reverb from the DDSP Autoencoder [EHGR20] it uses an *additive synthesis* plus noise model with subsequent reverb as a signal model. Acoustic knowledge of the piano was encoded into the system which was then trained to resynthesize piano performances with recorded MIDI data. The authors evaluated the trained model in terms of sound quality with a listening test and found, that their proposed method succeeded a different neural approach but was still inferior to a physical modeling piano synthesizer. Their polyphonic approach was inspired by DDSP based music source separation methods [KNK+22] and [SFDRB22].

Music Source Separation (MSS)

MSS is done via resynthesizing the sources from a given mix signal with a DDSP source model. Currently, there are two independent approaches to MSS with DDSP.

Schulze-Forster et al. performed *singing voice separation of choir recordings* with a DDSP *source-filter signal model* in [SFDRB22]. They developed a NN architecture to predict control signals for an excitation source and to parameterize an IIR filter for singing voice synthesis. The system is informed with the fundamental frequencies of the sources in the mix and is trained unsupervised. The proposed method features high data efficiency without the need for ground truth data and achieves "good separation quality even when trained on less than three minutes of audio" [SFDRB22]. This MSS method is described below in more detail since it is the a central precursor to this work.

Kawamura et al. developed a DDSP "mixture model for synthesis parameter extraction" from a mix of harmonic sounds [KNK+22]. Their polyphonic model consists of multi-

ple pretrained DDSP Autoencoders [EHGR20]. This model is then fitted to polyphonic instrument mixes.

Speech Synthesis

DDSP was also deployed for *neural speech synthesis*. In [FGKC20] speech was resynthesized intelligibly with a NN architecture similar to the DDSP Autoencoder. The input speech signal is transformed into a Mel spectrogram and then fed to a NN, similar to the decoder in [EHGR20] for predicting synthesis controls. The authors synthesized speech with a DDSP harmonic plus noise model from the ground truth pitch and the predicted control signals. Although "the quality of the synthesized speech can be improved" the system offers useful control capabilities due to the interpretable latent coding.

Webber et al. developed the *Autovocoder* in [WVBW+22] which allows "fast waveform generation from a learned speech representation using" DDSP. They obtained a frequency domain representation with a fast inverse transform for replacing the Mel spectrogram. The authors claim, that their approach is 5 to 14 times faster than comparable methods.

4.1.2 Neural Audio Effects via DDSP

Neural audio effects using DDSP usually contain one or a combination of audio effects such as a filter (equalizer), nonlinear distortion, a compressor, reverberation, etc. implemented with DDSP. A NN generates control signals for the DDSP effect(s) optimized in a way to achieve some goal regarding the input audio signal. Inputs to the NN can be the direct input signal of the audio effect(s), derived features from the input signal, other inputs with a relation to the optimization goal, or a combination of these.

As outlined in [RWSB21] audio effects can be difficult to use or may not be powerful enough to accomplish a desired task. In the past, several methods have been proposed to develop adaptive or intelligent audio effects or to emulate analog audio effects. Recent methods employ deep learning and can be categorized into

1. end-to-end direct transformation methods,
2. parameter estimators, and
3. DDSP methods.

Therefor, the DDSP methods are a subset of the overall neural audio effects methods. These neural audio effects using DDSP are outlined in the following.

Kuznetsov et al. explored "*differentiable IIR filters* for machine learing applications" in [KPE20]. Analogous to the circumstance that a 1D convolutional layer can be seen as a form of nonlinear FIR filter, the authors establish the link between IIR filters and RNNs. Then they present three IIR filter topologies with *parameter ranges ensuring stability*. Finally they successfully employ a differentiable IIR filter in a black-box virtual analog modeling system to emulate an analog guitar distortion effect. Although this publication

is categorized here as a neural audio effect, IIR filters are often part of a synthesis model such as in [SFDRB22].

In [RWSB21] DDSP with black-box audio effects was explored. The authors named their system *DeepAFx* which enables end-to-end training of NNs containing "*arbitrary* stateful third-party audio effects as layers". They trained a deep encoder which analyses the input signal to generate effect controls in a supervised fashion. The system is optimized to perform an audio task with this third-party audio effect according to the training data. They demonstrate the capabilities of their system "with three automatic audio production applications: tube amplifier emulation, automatic removal of breaths and pops from voice recordings and automatic music mastering" [RWSB21].

Nercessian et al. studied "lightweight and interpretable neural modeling of an audio distortion effect using hyperconditioned differentiable biquads" in [NSW21]. Steinmetz et al. explored "automatic multitrack mixing with a differentiable mixing console of neural audio effects" in [SPPS21]. Colonel and Reiss managed the "reverse engineering of a recording mix with" DDSP in [CR21]. Steinmetz, Bryan and Reiss studied "style transfer of audio effects with" DDSP in [SBR22] where they presented "a framework that can impose the audio effects and production style from one recording to another", yielding "audio effect control parameters that enable interpretability and user interaction" [CR21]. Automatic DJ transitions have been achieved in [CHL+22] using DDSP audio effects in GANs.

Lee et al. developed DDSP artificial reverberation models for the use in deep NNs in [LCL22]. They replaced IIR components with FIR approximations in their models to enable fast parallelized training. Furthermore they trained a system to recreate recorded room impulse responses with their parametric reverb model.

Speech Processing

DENT-DDSP [GCC22] was developed to allow performance enhancement of systems such as automatic speech recognition by *explicit distortion modeling* for speech. Explicit distortion modeling serves as a feature compensation step for speech processing. This system is fully explainable, "requires only 10 seconds of training data to achieve high fidelity" and the authors claim that their proposed method succeeds all of their baseline methods in terms of multi-scale spectral loss.

4.2 Precursors

In this section two important precursors to this work are presented in detail. First [EHGR20] by Engel et al. is introduced to give an understanding of the origin of DDSP literature. Then the unsupervised MSS method [SFDRB22] by Schulze-Forster et al. is described in detail, since it was the precursor to this work.

4.2.1 The Origin of DDSP

Engel et al. first explored *differentiable digital signal processing* (DDSP) in [EHGR20]. DDSP enables the direct use of classic signal processing techniques in NNs. The authors describe that "DDSP enables an interpretable and modular approach to generative modeling" [EHGR20].

They argue that the practical success of neural networks "is largely due to the use of strong structural priors such as"

— convolution,
— recurrence, and
— self-attention.

"These architectural constraints promote generalization and data efficiency to the extent that they align with the data domain." [EHGR20] DDSP is not limited to the audio domain but is universally applicable. However it was first used in audio synthesis which inspired others in this field to employ DDSP in their work.

The majority of neural audio synthesis models generate output signals directly in the time domain or in the frequency domain with a successive transform to the time domain. Models with strided convolutional layers suffer from a phase alignment problem, models employing a Fourier representation suffer from spectral leakage and autoregressive models are incompatible to loss functions which make use of human perceptions. Furthermore these are generally large networks with a demand of big amounts of training data.

Analysis-synthesis models such as vocoders are motivated by physics and perception. Systems have been developed to extract control signals for a synthsis model with hand-tuned heuristics. However all these systems which contain signal processing elements did not achieve end-to-end training which is a desirable goal for manageable DNN development. DDSP makes it possible to build fully differentiable synthesizers and audio effects. In this way, expert knowledge can be encoded into a NN with a signal model acting as a strong inductive bias.

Engel et al. proved the capabilities of DDSP by implementing the *DDSP Autoencoder* NN architecture. It was trained to resynthesize a specific monophonic musical instrument with a harmonic plus noise signal model, implemented with DDSP. With this DNN containing a DDSP signal model they achieved

— independent control over pitch and loudness,
— realistic extrapolation to pitches outside of the training range,
— blind dereverbaration by adding a reverb to the signal model,
— timbre transfer by turning a singing voice into a violin,
— while using fewer parameters than comparable NNs.

Figure 4.1 [3] gives an overview on the NN architecture of the DDSP Autoencoder. It is

3. Note that the encoder consists of two parts. A f_0 tracker is used to produce $f_0[m]$, a deterministic (not trainable) loudness measure is used to calculate the loudness $l[m]$ and the $\boldsymbol{z}$ encoder depicted in Figure 4.2 produces $\boldsymbol{z}[m]$. The authors most likely merged the CREPE f_0 tracker and the $\boldsymbol{z}$ encoder since they made one experiment not mentioned here, where they jointly trained f_0 tracking with the resynthesis task.

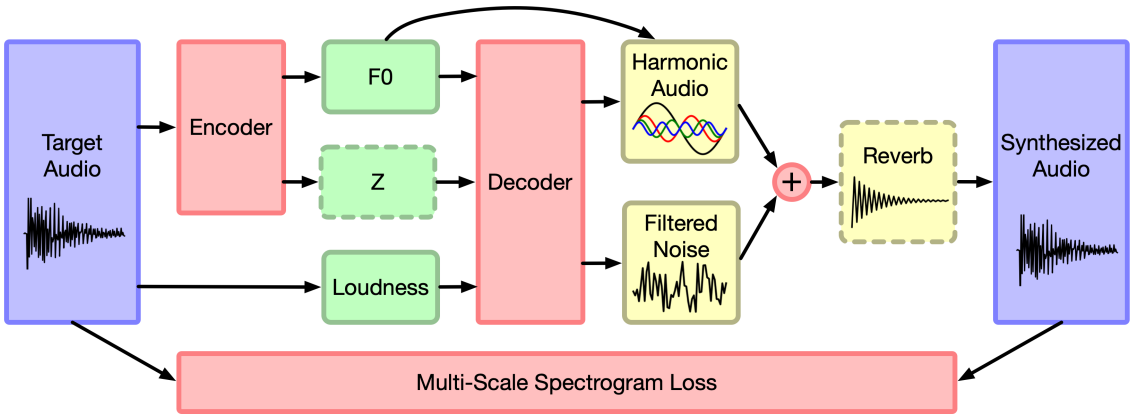


Figure 4.1 – DDSP Autoencoder NN architecture from [EHGR20, p.6]. Audio signals are in blue boxes, "red components are part of the neural network architecture", deterministic DDSP modules are in yellow boxes, the interpretable latent codings are in green boxes.

trained *unsupervised* by receiving an input $x[n]$ and resynthesizing the output $y[n] = \hat{x}[n]$. The NN is optimized to achieve similarity $\hat{x}[n] \approx x[n]$ with a phase-independent loss function.

From the input $x[n]$ a latent coding $(f_0[m], \boldsymbol{z}[m], l[m])$ is produced where m is the time frame index, $f_0[m]$ is the fundamental frequency of the input signal $x[n]$, $l[m]$ is the loudness and $\boldsymbol{z}[m]$ is the residual vector which effectively encoded timbre. Hence the latent coding is interpretable. $f_0[m]$ is produced from $x[n]$ with a pretrained CREPE [KSLB18] pitch tracker. The harmonic oscillators' fundamental frequency is directly controlled by the analyzed pitch track $f_0[m]$. $l[m]$ is calculated with a deterministic A-weighted loudness measure as in [HERG19]. $\boldsymbol{z}[m]$ is derived from the neural encoder architecture depicted in Figure 4.2

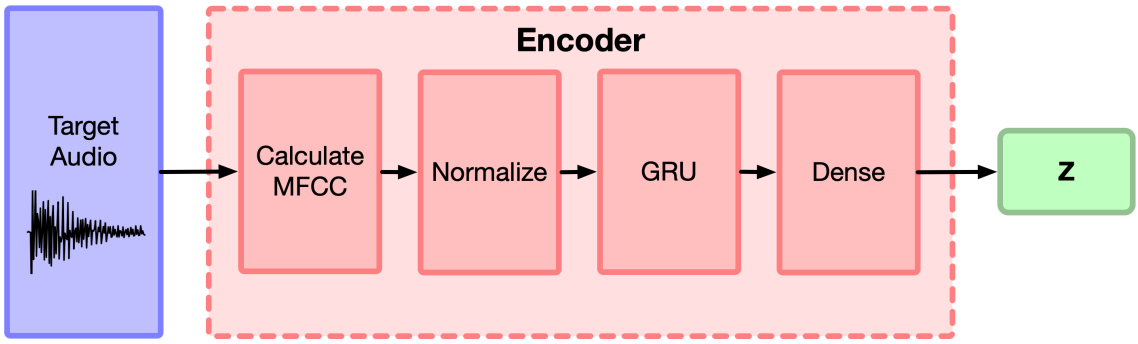


Figure 4.2 – The $\boldsymbol{z}$ encoder NN architecture from [EHGR20, p.16]. MFCC denotes the mel frequency cepstral coefficients.

The decoder network depicted in Figure 4.3 produces control signals for the DDSP harmonic plus noise signal model from the latent coding $(f_0[m], \boldsymbol{z}[m], l[m])$. It contains multilayer perceptron (MLP) blocks depicted in Figure 4.4.

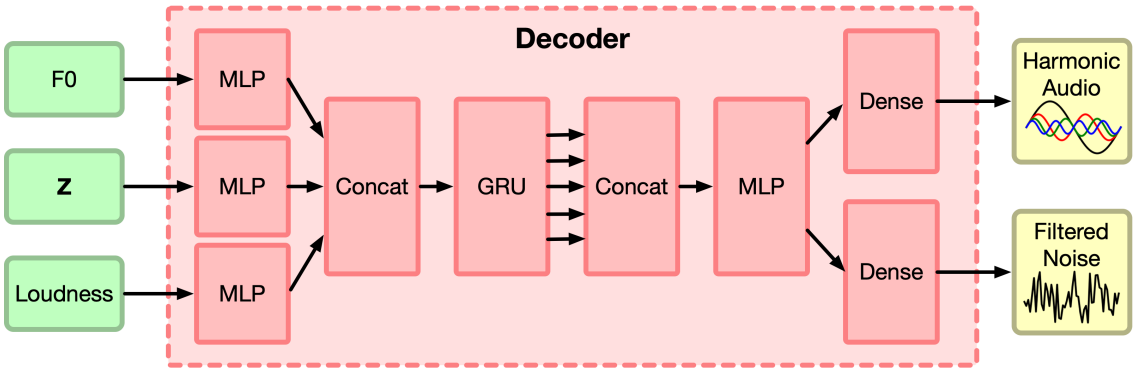


Figure 4.3 – The decoder of the DDSP Autoencoder NN architecture from [EHGR20, p.16].

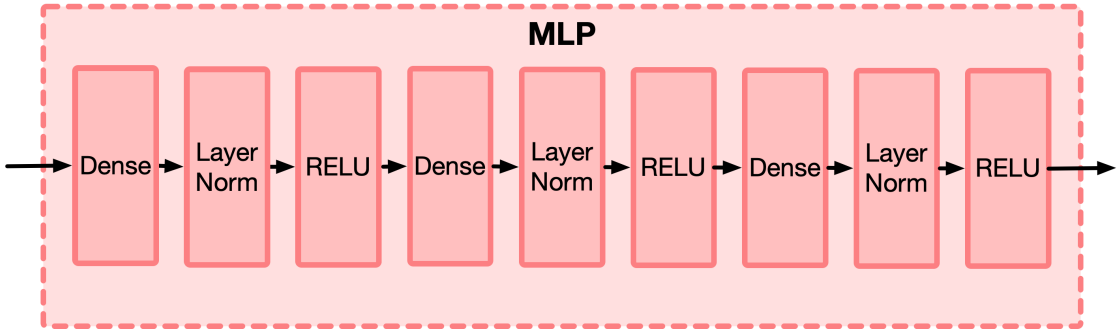


Figure 4.4 – Multilayer perceptron (MLP) NN architecture used in the decoder from [EHGR20, p.16].

DDSP Components

As depicted in Figure 4.1 the DDSP Autoencoder uses a harmonic plus noise signal model with a subsequent reverb. Employing this signal model encodes expert knowledge about the training data into the system. Therefor, the output space is restricted to harmonic signals with variable filtered noise amounts which corresponds to a wide range of musical instruments. On the other hand, it would be impossible to effectively model inharmonic sounds with this approach which may require different signal models such as a frequency modulation model as in [CMS22], or a generic additive synthesis without the harmonic restriction. Such an additive synthesis model was used in part in [RMR22] for modeling string inharmonicity.

The harmonic plus noise signal model consists of the a sine oscillator bank with added filtered noise. The noise filter is parameterized via the *frequency sampling method*, resulting in a time variable FIR filter. Reverberation is implemented as convolution reverb.

Harmonic Oscillator The harmonic oscillator is given by

$$y[n] = \sum_{k=1}^{K} A_k[n] sin(\phi_k[n]) \tag{4.1}$$

with the partial number $k \in K$, the time variable amplitude $A_k[n]$ of the k-th partial and its instantaneous phase $\phi_k[n]$. This phase is obtained by integration via

$$\phi_k[n] = 2\pi \sum_{m=0}^{n} f_k[m] + \phi_{0,k} \tag{4.2}$$

with the summation ('integration') index m, the instantaneous frequency $f_k[n]$ of the k-th partial and the initial phase $\phi_{0,k}$ which can be fixed, randomized or learned.

A harmonic signal is then obtained by restricting the partial tones to integer multiples of the fundamental frequency $f_k[n] = kf_0[n]$. "Thus the output of the harmonic oscillator is entirely parameterized by the time-varying fundamental frequency" $f_0[n]$ "and harmonic amplitudes" $A_k[n]$. [EHGR20, p.4]

FIR Filter Design Parametrization for a time variable FIR filter is achieved with the frame-wise NN prediction of the FIR transfer function $H[k]$. In this way filtering is done according to the optimization goal set with the loss function of the NN. In the case of the DDSP Autoencoder the lower output of the decoder network in Figure 4.3 is used as transfer function $H[k]$ for the filter which generates filtered noise. To "control the time-frequency resolution trade-off of the filter" $H[k]$ is windowed with a Hann window before filtering. Filtering is done in the frequency domain $y[n] = iDFT\{H[k]X[k]\}$ with the transformed input signal $X[k] = DFT\{x[n]\}$ for every frame m.

The reverb module in Figure 4.2 is implemented as a FIR filter in the same way as the time-variable FIR filter above. Equally its transfer function is predicted by a NN. The difference between them is, that the reverb has a much longer impulse response $h[n] = iDFT\{H[k]\}$ than the time variable FIR filter and that it is time-invariant.

Multi-scale spectral loss The employed loss function is the *multi scale spectral loss* L_{MSS} which is calculated as a sum of losses

$$L_{MSS} = \sum_{j=1}^{J} L_j \tag{4.3}$$

for a number J of different FFT sizes. The individual losses are given by the sum of the MAEs

$$L_j = ||X_j - \hat{X}_j||_1 + ||log(X_j) - log(\hat{X}_j)||_1 \tag{4.4}$$

with the L1 distance $|| \cdot ||_1$, the target spectrogram $X_j[k,m] = |STFT_j\{x[n]\}|$ and the prediction spectrogram $\hat{X}_j[k,m] = |STFT_j\{\hat{x}[n]\}|$ with FFT size index j. L_j reduces the mean absolute difference of the linear and logarithmic spectrograms to a single scalar loss value. More specifically it is calculated from the spectrograms by calculating a mean over the time frames $m \in M$ and the frequency bins $k \in K$ with

$$L_j = \frac{1}{K}\sum_{k=1}^{K}\frac{1}{M}\sum_{m=1}^{M}|X_j[k,m]-\hat{X}_j[k,m]|+\frac{1}{K}\sum_{k=1}^{K}\frac{1}{M}\sum_{m=1}^{M}|log(X_j[k,m])-log(\hat{X}_j[k,m])| \tag{4.5}$$

as implemented in their repository [Goo23b] [4]. It is important to employ such a phase independent loss function for proper training results. In this way e.g. a slightly delayed perfect reconstruction $\hat{x}[n] = x[n-\epsilon]$ does not produce a big loss which matches the human perception of sound.

Results

The trained DDSP Autoencoder is capable of high-quality neural audio synthesis and timbre transfer [5]. The authors found, that synthesis had higher quality when using the pretrained CREPE f_0 tracker compared to jointly learning f_0 analysis with sound resynthesis. They trained the DDSP Autoencoder to resynthesize only 13 minutes of solo violin performances, recorded in a consistent room environment. In this configuration the $\boldsymbol{z}$ encoder was not used but the pretrained CREPE pitch tracker was used.

The trained DDSP Autoencoder achieved high quality violin synthesis with extrapolation to pitch ranges outside the training data. The interpretable latent coding allows independent control over loudness and pitch for either resynthesis to perform timbre transfer turning singing voice into a violin or by synthesizing audio from MIDI notes via feeding appropriate envelopes to the input of the decoder network.

Additionally by incorporating the reverb module into the signal model they achieved blind deconvolution of the solo violin. This was simply done by bypassing the reverb at resynthesis. Furthermore the estimated room impulse response was then available as the impulse response of the reverb module.

Whereas the DDSP Autoencoder with pretrained CREPE has 240k to 7M parameters, other audio synthesis models have 15M parameters upwards [EHGR20, p.17]. This makes it lightweight and efficient both in terms of training data (around 10 to 15 minutes of unlabeled instrument recordings) and model training.

4. `https://github.com/magenta/ddsp/blob/main/ddsp/losses.py`

5. An overview of the results is given in `https://magenta.tensorflow.org/ddsp` [Goo23b] and additional audio examples are given here `https://storage.googleapis.com/ddsp/index.html` [Goo23a]. Recently a VST plugin was released here `https://magenta.tensorflow.org/ddsp-vst` [Goo23c]

4.2.2 Unsupervised Audio Source Separation Using Differentiable Parameteric Source Models

Inspired by DDSP [EHGR20], Schulze-Forster et al. proposed a novel unsupervised MSS method for singing voice separation in [SFDRB22]. They modeled each source with a DDSP source-filter model and trained a NN to reconstruct the input mixture signal as the sum of the synthesized sources by predicting control signals for the source models with known fundamental frequencies. At inference time, separation quality is improved by soft-masking the mixture signal with the masks obtained from the resynthesized sources. With this approach they achieved good separation quality as well as high data efficiency at the task of singing voice separation.

First the authors state that state-of-the-art neural MSS methods (introduced in section 3.4) are not able to separate *homogeneous sources* such as multiple instruments of the same family in one recording. Furthermore these systems require labeled data, namely clean recordings of the separate target source signals. The creation of these separate stems is laborious, expensive and in some cases it is even impossible. This fuels the demand for unsupervised methods requiring only mix signals for training.

Training Procedure The proposed method achieves unsupervised learning by resynthesizing the mixture signal with separate DDSP source-filter models for each source. The DNN observes the input mix together with the sources fundamental frequencies f_0 which are estimated with a polyphonic pitch tracker from the mix. With these inputs, it generates control signals for the DDSP source models in such a way, that the sum of the synthesized sources approximates the mix signal according to the loss function.

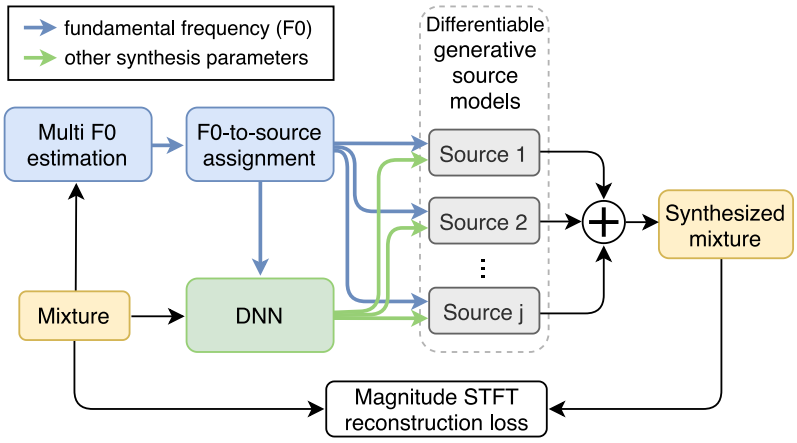


Figure 4.5 – Training procedure overview from [SFDRB22, p.3].

Figure 4.5 shows an overview of the unsupervised training procedure of this MSS method. The mixture signal $x[n]$ is fed to a DNN. This DNN estimates control signals for the separate source models in such a way that they synthesize separate source signals $\hat{s}_i[n]$, forming a mixture signal $\hat{x}[n]$ that minimizes the loss function. From the input mixture $x[n]$ the fundamental frequencies f_0 are estimated with a polyphonic pitch tracker and

assigned to the corresponding sources. The f_0 information is used not only to directly synthesize source signals with the appropriate pitch but also to inform the DNN. This is a common approach in DDSP literature, since it enables the creation of source signals with good pitch accuracy and to predict pitch dependent controls. The employed loss function is the multi scale spectral loss L_{MSS} as in [EHGR20], which is defined in eq. (4.3) and (4.4).

Source Model To model the singing voice the authors employed a source-filter signal model. Such a model is motivated by the physiology of the human speech process. Thereby the glottis produces an excitation signal $e[n]$ which is filtered by the vocal tract. The vocal tract is modeled with a time-varying all-pole filter of order O and filter coefficients a_o. The z transform of the source model is given by

$$\hat{S}_i(z) = E_i(z)\frac{1}{A_i(z)} \tag{4.6}$$

with the z transform of the estimated source $\hat{S}_i(z) = \mathcal{Z}\{\hat{s}_i[n]\}$, the z transform of excitation signal $E_i(z) = \mathcal{Z}\{e_i[n]\}$, the transfer function $\frac{1}{A_i(z)}$ of the time-varying IIR all-pole vocal tract filter, and the source index i. This source model is described in the time domain as

$$\hat{s}_i[n] = e_i[n] - \sum_{o=1}^{O} a_{o,i}\hat{s}_i[n-o]. \tag{4.7}$$

The excitation signal $e[n]$ is formed with a harmonic plus noise model as defined in eq. (4.1), which was used in [EHGR20] as an overall signal model for musical instruments. They notate the harmonic plus noise model as

$$e[n] = (\alpha[n]h[n]) * r[n] + (w[n] * d[n])g[m] \tag{4.8}$$

with the convolution operator $*$, the time sample index n, the time frame index m, and the time-varying amplitude $\alpha[n]$ of the harmonic signal $h[n]$. $r[n]$ and $d[n]$ are finite impulse responses. $w[n]$ corresponds to white noise and $g[m]$ is the frame-constant noise amplitude. The harmonic signals partial amplitudes are fixed and $r[n]$ and $d[n]$ are time-invariant to provide a global spectral shape. Eq. (4.8) only emulates the excitation, whereas the short time variations produced by articulation of words are entirely modeled with the all-pole filter $\frac{1}{A(z)}$.

With this complete definition the source model parameters are given by ($f_0[n]$, $a_o[m]$, $\alpha[n]$, $r[n]$, $g[m]$, $d[n]$). The fundamental frequency f_0 is provided per frame ($f_0[m]$) to the model but is then upsampled to audio rate ($f_0[n]$). A visualization of the source model with intermediate spectrograms is depicted in Figure 4.6.

Regarding this specific source model, the authors claim that their "proposed method is not specific to any particular source model and any parametric model may be used as long as it can be formulated in a differentiable way." [SFDRB22, p.3]

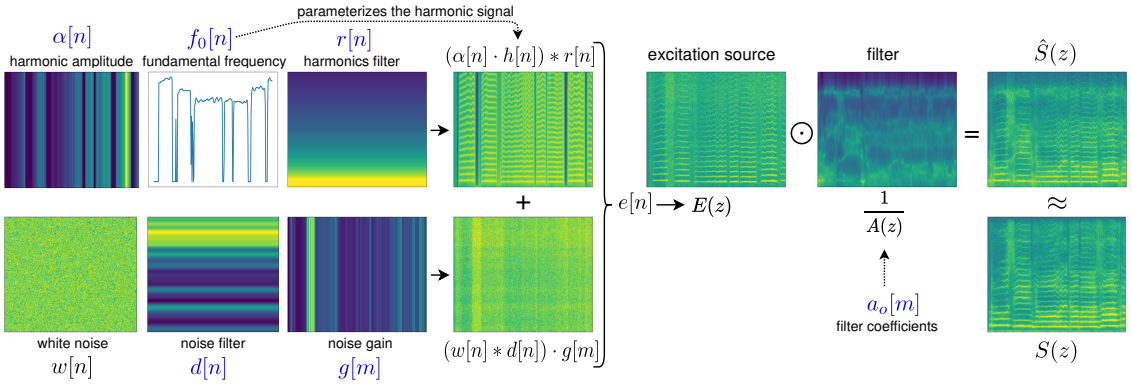


Figure 4.6 – Visualization of the processing steps in the source-filter model taken from [SFDRB22, p.4]. Model parameters are written in blue. "Although most components are visualized through magnitude spectrograms, processing is not necessarily done in the time-frequency domain" [SFDRB22, p.4].

The DNN The control signals for the source models $a_o[m]$, $\alpha[m]$, $g[m]$, and $d[m]$ "are obtained with a DNN and" $r[n]$ "is fixed manually" [SFDRB22, p.6]. This DNN is depicted in Figure 4.7.

The DNN architecture consists of a mixture encoder which is similar to the z encoder from [EHGR20] in Figure 4.2 and a decoder to generate a *latent source representation* from the mixture signal and the sources' f_0 tracks. The latent source representation is the equivalent to the latent coding of a standard autoencoder NN architecture. From this latent source representation, the different control signals or their precursors for further processing are predicted with simple dense or recurrent layers.

In Figure 4.7 the input mixture signal is present as a logarithmic spectrogram $log(|STFT\{x[n]\}|)$ with M time frames and K frequency bins. The mixture encoder first applies a custom scale and shift operation (custom layer normalization) to the logarithmic spectrogram for normalization. "Each spectrogram is normalized by subtraction of its mean and division by its standard deviation. Then, each frequency bin is scaled and shifted by dedicated learned scalars." [SFDRB22] Such a first step is common in feature engineering to gain expressive features for the NN to effectively learn from them.

Then three unidirectional GRUs are applied, followed by a linear layer, yielding the *latent mixture representation*. This learned representation of the mixture signal spectrogram may focus on the important parts of the spectrogram for solving the problem at hand. For the output of the mixture encoder, the latent mixture representation is duplicated I times for every source i.

The source fundamental frequencies $f_0[m]$ are converted to MIDI note numbers and then scaled to an interval between 0 and 1 to act as a valid neural input. In the decoder both the duplicated latent mixture representation and the scaled source pitches are processed separately by a MLP. This MLP "consists of three repetitions of linear layer, layer normalization (...), Leaky ReLU activation" [SFDRB22, p.5]. The MLP outputs are concatenated to combine mix and f_0 information and then fed through a unidirectional GRU and another

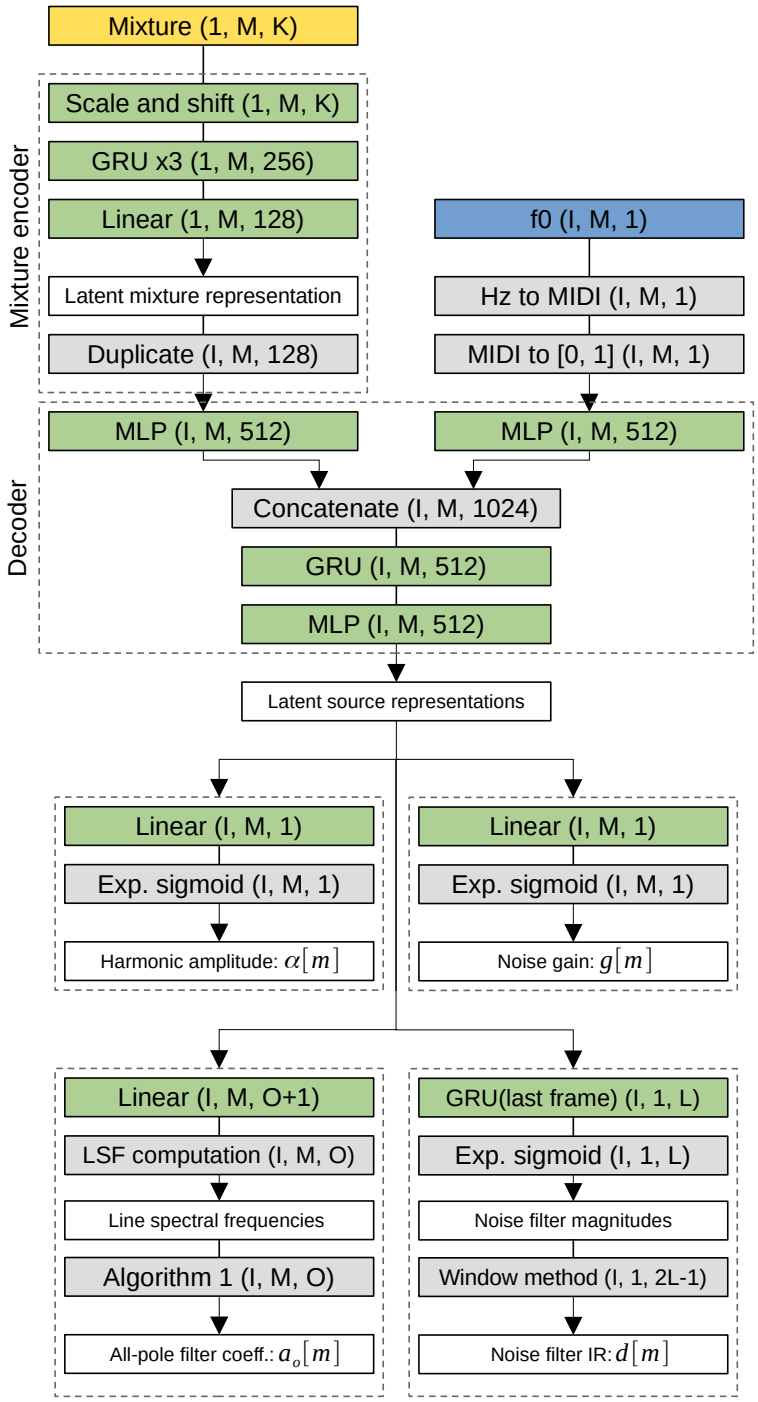


Figure 4.7 – NN architecture. "Transformations with learnable parameters are shown in green, predefined processing steps in gray, (intermediate) outputs in white boxes. The output shape of a transformation is shown in the right part of the box." [SFDRB22, p.5]

MLP. The output of the decoder forms the *latent source representations*.

From this latent source representation, 4 different control signals for the source models are predicted. The amplitude $\alpha[m]$ of the harmonic signal and the noise amplitude $g[m]$

are predicted via a dense layer consisting of a linear layer with an *exponential sigmoid*

$$y = y_{max} \cdot \sigma(x)^{log(10)} + 10^{-7} \tag{4.9}$$

as output activation function. It is a modified version of the standard sigmoid function from eq. (2.5) also employed in [EHGR20]. The exponential sigmoid is used for predicting logarithmic parameters such as amplitude or frequency.

Like the other outputs, the last frame of the unidirectional GRU output is fed through an exponential sigmoid activation for obtaining the impulse response of the FIR noise filter $d[m]$, as in [EHGR20]. By parameterizing the noise filter from the last output frame of the GRU, it contains information about the whole source signal encoded in the mixture signal. The noise filter magnitudes are kept time-invariant, hence $d[n] = d[m]$.

Finally the IIR filter coefficients $a_o[m]$ are predicted by also applying a linear layer to the latent source representations, but then using a filter parametrization technique using *line spectral frequencies* (LSF). For further information about LSF and the (differentiable) algorithm (algorithm 1 in Figure 4.7) to obtain the IIR filter coefficients see [SFDRB22, p.6].

Results The available data consisted of choir recordings of Bach chorals and Barbershop quartet with ground truth, which are separate recordings of individual singing voices. The Bach choral voices are soprano, alto, tenor and bass and the Barbershop voices are tenor, lead, baritone and bass. These recordings were separated into the *full training set* consisting of 91 minutes and 20 seconds of audio material and a *full validation set* with 9 minutes and 10 seconds of audio material. A *small training set* and *small validation set* has been formed, respectively, with 2 minutes and 40 seconds for training and 2 minutes and 20 seconds of validation. The *test set* comprised chorals of a total length of 6 minutes and 48 seconds which are not included in the previous sets.

Two sets of experiments have been conducted by the authors. The first set of experiments conducted training for separating a number of 2 sources present in the mix. The second set of experiments conducted training for separating a number of 4 sources. In each set 4 different training configurations regarding the training set and learning type have been used:

— US-F: unsupervised training with the full dataset,
— US-S: unsupervised training with the small dataset,
— SV-F: supervised training with the full dataset,
— SV-S: supervised training with the small dataset.

Supervised training was performed by calculating the sum of the MSS loss for every individual source estimate and its target source.

Mixture spectrograms were computed with a FFT size of 512 and a hop size of 256 resulting in 257 frequency bins. The soft masking to produce masked source estimates from the directly synthesized source estimates was done with a FFT size of 2048 and a hop size of 256 samples.

$r[n]$ was fixed to a 6dB/octave lowpass characteristic with a cutoff frequency at 200Hz. The order of the all-pole filter was set to $O = 20$. Training was done with the ADAM optimizer, "a batch size of 16 and a learning rate of 0.0001" [SFDRB22] stopping after 200 consecutive epochs.

Evaluating the trained models on the test dataset with the scale invariant signal-to-distortion ratio (SI-SDR) [LRWEH19] of the *masked sources*, the authors recognized the following points.

— Separation quality was higher for 2 sources than for 4 sources in general.
— "The proposed unsupervised method (US-F, US-S) performs better than the baselines." [6] [SFDRB22, p.9])
— The proposed unsupervised method "achieves almost the same performance whether isolated target sources are available for training or not." [SFDRB22, p.9]
— The "performance of the proposed method does not drop drastically when the amount of training data is decreased by 97% (US-F vs. US-S and SV-F vs. SV-S)." [SFDRB22, p.9]

In summary, the unsupervised learning approach produced good separation quality with very high data efficiency trained on less than 3 minutes of audio. It achieved equivalent performance to the supervised approach. This was achieved by encoding expert knowledge into the source models via DDSP and effective usage of the sources pitch tracks.

Directly synthesized sources performed worse than the baselines but the authors note that "source estimates generated by parametric models are a worthwhile goal for future research" for "tasks such as timbre or style transfer, transposition, and melody editing" [SFDRB22, p.10].

6. Baselines were a learning free non-negative matrix factorization approach and a U-Net approach trained on the same datasets.

Chapter 5

MSS of Guitar Recordings Using DDSP

On the basis of [SFDRB22] (which was described in section 4.2.2), the concept of this work was to perform MSS employing a new source model. Since a machine learning systems design always involves the selection of data for training, validation and testing, it was reasonable to base the decision towards which sources should be modeled on available data with ground truth. Furthermore, open source or open access datasets are preferable, since they are free and enable perfect reproducibility for other researchers. For this reasons I decided to use the *Guitarset* [XBP+18] dataset for training, validation and testing. This dataset is described in more detail below in section 5.3.2.

In this chapter, first the results from [SFDRB22] with their method for singing voice separation was recreated. As a preliminary step towards MSS a DDSP physical string model was used as source model in a modified version of the DNN from [SFDRB22] to resynthesize signals with one neurally predicted parameter. Subsequently, 3 experiment series for MSS using DDSP were conducted which are referred to as experiment series A, B and C. The repository for the preliminary experiment is available on the Git IEM server `https://git.iem.at/s1061531/umss-guitar-prex` and the repository for experiment series A, B and C is available on the Git IEM server at `https://git.iem.at/s1061531/umss-guitar`. The results of the experiments are presented in the subsequent chapter 6.

5.1 Recreation of the Original Method

As a first step, the results of [SFDRB22] for singing voice separation have been recreated. Since the authors used proprietary datasets for training and validation a replacement had to be found. I chose to employ the *Choral Singing Dataset* [1] [Zen19] introduced in [CGGMDL18] for training, validation and testing. Originally this dataset was used solely for testing, but considering the data efficiency of the method at hand, less than 3 min-

1. `https://zenodo.org/record/2649950#.Y-jjaRzMKi0`

utes of training data should be sufficient [SFDRB22, p.1] and the performance should approach the results of the authors of [SFDRB22]. Pitch tracking was done by employing the CREPE pitch tracker [SFDRB22] on the separate voices.

The three songs of the Choral Singing Dataset have been split:

— *Nino Dios* with a length of 1min 40sec used as *training set*.
— *Locus Iste* with a length of 3min used as *validation set*.
— *El Rossinyol* with a length of 2min 20sec used as *test set*.

Two MSS models have been trained. One model for separating a mixture of 2 voices and one model for separating 4 voices.

5.2 Preliminary Experiment: Karplus-Strong Resynthesis

As a preliminary step towards MSS employing a DDSP physical guitar model I decided to employ the physical model first to reproduce a single output signal for a fixed frequency and one free parameter predicted by the NN.

As the signal model a simple *Karplus-Strong physical string model* [KS83], [Smi13, Ch. The Karplus-Strong Algorithm] was used. The Karplus-Strong model is a Waveguide model [Smi13] as mentioned in [KVT98]. It is the most simple way to implement a rough physical model of a vibrating string.

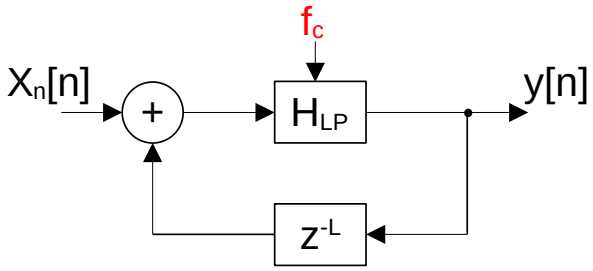


Figure 5.1 – The block diagram of a simple Karplus-Strong physical string model. The neural predicted parameter f_c (cutoff frequency) of the lowpass filter H_{LP} is shown in red.

Figure 5.1 shows a block diagram of a simple Karplus-Strong algorithm. It consists of a delay line z^{-L} with length L and a lowpass filter H_{LP} inside a feedback loop. The excitation signal is a white noise burst $x_n[n]$. The length of the delay line L plus the group delay of the feedback filter determines the fundamental frequency f_0 of the output signal. The note lengths or damping of the string is influenced by the cutoff frequency f_c of the lowpass filter. Since it is a lowpass, high frequencies get damped very quickly and low frequencies are sustained longer in the output signal.

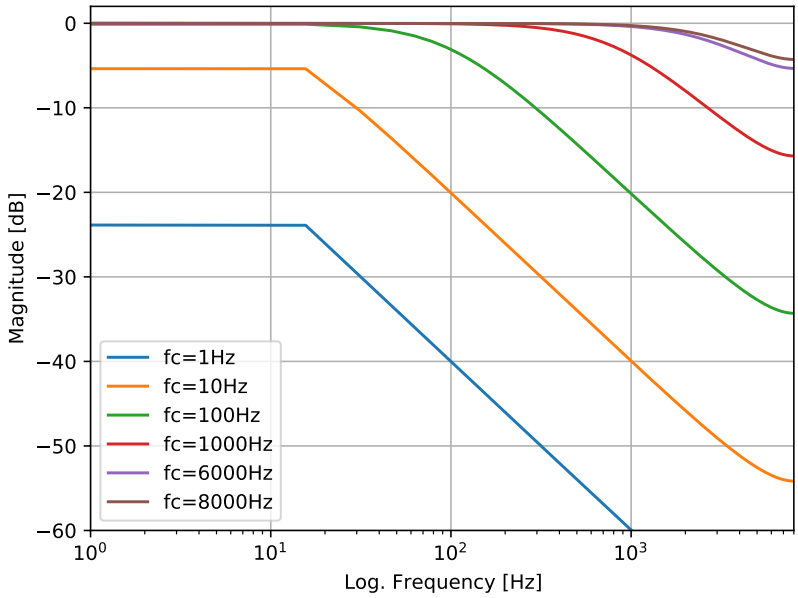


Figure 5.2 – Magnitude response of lowpass filter $h_{LP}[n]$ at $f_{sr} = 16kHz$ for different cutoff frequencies f_c.

The output signal $y[n]$ is given by

$$y[n] = x_n[n] + h_{LP} * y[n - L - 1] \tag{5.1}$$

with the time sample index n, the convolution operator $*$ and the lowpass impulse response $h_{LP}[n]$ given by the difference function in eq. (5.2). The lowpass of the source model is a simple IIR filter from [Wik23b]

$$y[n] = \alpha x[n] + (1 - \alpha)y[n - 1] \tag{5.2}$$

with the coefficient

$$\alpha = \frac{2\pi\Delta_{sr}f_c}{2\pi\Delta_{sr}f_c + 1}$$

and the sampling interval $\Delta_{sr} = 1/f_{sr}$.

The lowpass transfer function is given by $H_{LP}(z) = \frac{\alpha}{1-(1-\alpha)z^{-1}}$ and its magnitude response is shown in Figure 5.2. For $f_c < 100Hz$ at $f_{sr} = 16kHz$ the resulting -3dB cutoff frequency does not coincide with the set f_c and the whole signal is attenuated. However, such a behavior may be a useful feature as a string damping filter, approximating physical behavior.

The Fractional Delay Line

An arbitrary f_0 leads to a fractional delay of $L = f_{sr}/f_0 = T/\Delta_{sr} \in \mathbb{R}$ samples with the audio sample rate f_{sr} and the fundamental period T. Hence, to synthesize arbitrary notes from a continuous fundamental frequency interval, a fractional delay line [Smi13, Ch. Delay-Line and Signal Interpolation] is needed. In this experiment, the fractional delay line is implemented via a ring buffer with linear interpolating read. The linearly interpolated fractional sample $\hat{x}[\tau]$ is given by

$$\hat{x}[n+\eta] = (1-\eta)x[n] + \eta x[n+1] \tag{5.3}$$

with the fractional sample $\eta \in [0, 1]$ and $\tau = n + \eta$.

Since all these operations are differentiable, the whole Karplus-Strong string model from eq. (5.1) is differentiable and can be used as a source model in a NN for source separation.

Training Procedure

The goal of this work was to implement unsupervised learning of source separation by resynthesizing the mix signal [SFDRB22]. However, in this preliminary experiment, the goal was to *resynthesize a single source signal* by predicting a single parameter f_c for every input signal. Therefor f_c is the only free neural predicted parameter in this experiment.

The input signals are synthesized notes by the Karplus-Strong string model itself, with fixed f_0 and fixed f_c. For synthesis also the *note onset is fixed*. Hence, the system is informed when to synthesize the note. As in [SFDRB22] the employed loss function is the *multi scale spectral loss* [EHGR20] defined in eq. (4.3). Therefor the DNN is trained to find the fixed cutoff frequency of the generated dataset.

Dataset and Hyperparameters

Two different datasets have been synthesized with the Karplus-Strong source model. One with a randomly seeded excitation signal and one which used the same excitation signal for synthesis. The input and output time signals have a length of 1 second at a sampling rate of $f_{sr} = 16kHz$. The Karplus-Strong model was set to a fixed fundamental frequency of $f_0 = 440Hz$ and to a fixed loop filter cutoff frequency of $f_c = 6000Hz$ to create the dataset. The training set consisted of 16 examples and the validation set consisted of 4 examples.

— batch size: 4
— optimizer: Adam (as in [SFDRB22])
— learning rate: $1 \cdot 10^{-3}..1 \cdot 10^{-6}$
— input FFT size: 512 samples (as in [SFDRB22])
— input FFT hop size: 256 samples (as in [SFDRB22])
— L_{MSS} FFT sizes j: 2048, 1024, 512, 256, 128, 64 samples (as in [SFDRB22])

The Neural Network Architecture

The NN architecture from [SFDRB22] depicted in Figure 4.7 served as a base architecture. It has been modified to the architecture depicted in Figure 5.3 for predicting a lowpass cutoff frequency from the input signal. All GRU layers are unidirectional. The input signal is fed to the first layer as a logarithmic spectrogram as described in section 4.2.2 with M time frames and K frequency bins. For synthesis the predicted cutoff frequency f_c for the lowpass filter in the source model is retrieved from the last layer.

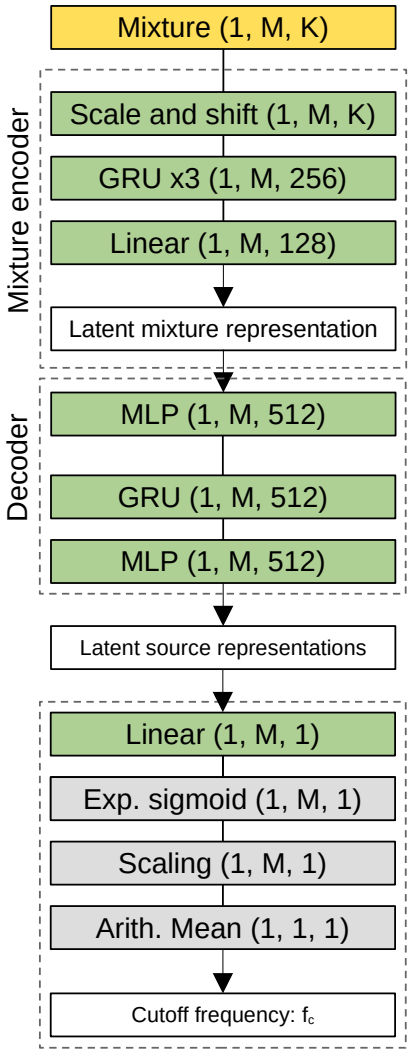


Figure 5.3 – NN architecture for the preliminary experiment (modified from [SFDRB22]). Layer output dimensions are notated in parantheses with M time frames and K frequency bins.

As in [SFDRB22] the mixture encoder produces a latent representation of the input signal for the decoder. Although in this experiment the input is not a source mixture but a single source signal, the system is intended to be expanded to multiple sources. Hence, the labeling in parantheses with a leading 1 in Figure 5.3. The decoder and the output layers produce the predicted value of f_c corresponding to the input signal.

The scaling of the neural prediction $p \in [0, 1]$ is done as follows

$$f_c = p(\frac{f_{sr}}{2} - f_{min}) + f_{min}$$

with a minimum frequency $f_{min} = 20Hz$. The last operation in the NN is an arithmetic mean across all time frames M to yield a single f_c value for synthesizing one output prediction $y[n]$ from one input example $x[n]$.

5.3 Experiment A Series

In this experiment series, guitar string signals are separated by synthesizing source estimates with a simple Karplus-Strong physical string model. Again, the cutoff frequency f_c is predicted by the DNN according to its mixture input. Two experiments have been conducted in this series. One experiment has been conducted with a physical modeling sampling rate of 16kHz (the same as the dataset sampling rate) and experiment with the double sampling rate of 32kHz. This was due to the need for an extended range of f_c for sound synthesis, producing notes with longer sustain. Since the maximum cutoff frequency of the lowpass is bounded by the Nyquist frequency, a higher sampling rate enables the simulation of lower string damping and hence longer sustain than at lower sampling rates.

5.3.1 Training Procedure

The goal of this experiment series is to approach unsupervised music source separation by resynthesizing the mix signal as in [SFDRB22]. This procedure is depicted in Figure 5.4. The deep neural network (DNN) is fed with the mix signal $x[n]$ and it is informed with the sources fundamental frequency $f_{0,i}$, which was analyzed with the pretrained CREPE [KSLB18] monophonic pitch tracking network. From these inputs the NN predicts the synthesis control signals c_i for a physical string model for every string i. Note onsets are analyzed with a simple algorithm from f_0 and its confidence, obtained from the pitch tracker.

The physical source model produces the *synthesized sources* $\hat{s}_i[n]$ from the neurally predicted control signals c_i. These sources are summed to form the *predicted mixture signal* $\hat{x}[n]$. This mix is then used to calculate the cost function L_{MSS} for training the DNN. Since the network is informed with f_0 information tracked by CREPE [KSLB18] from the target sources $s_i[n]$ as in [EHGR20] this is not true unsupervised learning but it is an approach to it.

5.3.2 Dataset: Guitarset

The *Guitarset* [XBP+18] is a dataset of single string classical guitar recordings featuring

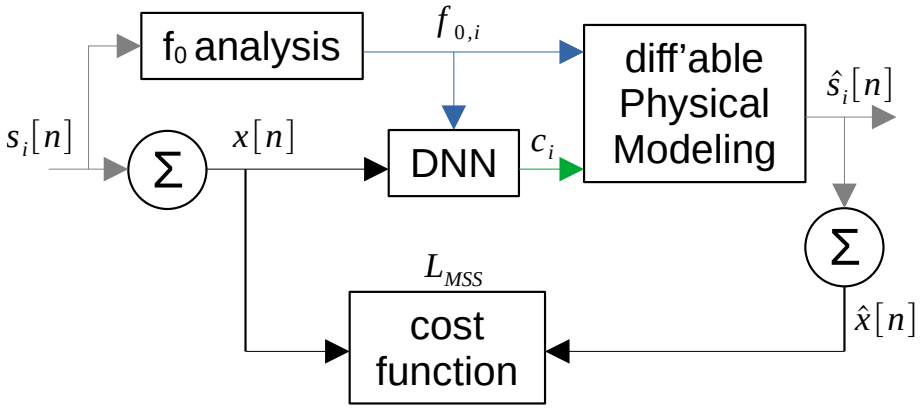


Figure 5.4 – Block diagram of the overall training procedure. With the source (guitar string) index i, the target sources $s_i[n]$, the (predicted) directly synthesized sources $\hat{s}_i$, the fundamental frequencies $f_{0,i}$, the target mix signal $x[n]$, the synthesized mix signal $\hat{x}[n]$, and the predicted control signals c_i for source synthesis.

— 5 different musical genres: Bossa Nova, Funk, Jazz, Rock, Singer Songwriter
— 2 different playing styles: Comping and Soloing

There are 36 recordings per style and genre, which makes a total of 360 recordings with a varying duration of 20 to 30 seconds. Debleeded hexaphonic pickup recordings have been used for the experiments. This guitar recording dataset provides the ground truth (monophonic single string recordings) to generate polyphonic mixture signals for training, validation and testing. Pitch tracking was performed using the CREPE pitch tracker [KSLB18] on the individual string signals as in [EHGR20].

Consecutive recordings are loaded for training without shuffling batches. Therefor the recordings are presented to the model in the original musical order. Every epoch the recordings order of the training set is randomized. For these experiments a subset of the Guitarset [XBP+18] is compiled with the following settings.

— dataset: Guitarset
— sample rate: 16kHz
— training set: 8 audio files
— validation set: 2 audio files
— test set: 2 audio files
— playing style: comping
— genres: Bossa Nova
— allowed-strings: all 6 guitar strings
— example-length: 2s (length of 1 training example in a batch)
— pitch tracker: CREPE

5.3.3 Source Model

As in the preliminary experiment above, a simple Karplus-Strong string model as depicted in Figure 5.5 is used as source model. The lowpass was transferred to the feedback path and it was extended with a highpass to mitigate DC offsets. The highpass cutoff frequency is fixed to 5Hz.

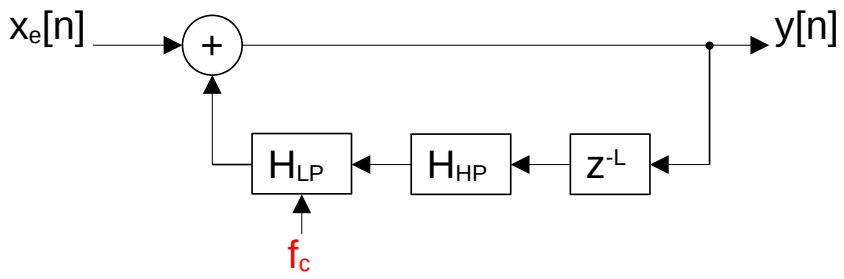


Figure 5.5 – Block diagram of the Karplus-Strong physical string model. The lowpass cutoff frequency f_c is the only neurally predicted control signal for this source model.

The excitation signal $x_e[n]$ is a white noise burst of 5ms triggered by the note onsets. The output signal is given by

$$y[n] = x_e[n] + h_{LP}[n] * h_{HP}[n] * y[n - L - 1]. \tag{5.4}$$

The same fractional delay line z^{-L} as above defined in eq. (5.3) was used in this experiment. Also the lowpass filter from eq. (5.2) is used as $h_{LP}[n]$. The highpass $h_{HP}[n]$ is given by a simple IIR filter with difference equation

$$y[n] = \alpha y[n-1] + \alpha(x[n] - x[n-1]) \tag{5.5}$$

from [Wik23a] with the coefficient

$$\alpha = \frac{1}{2\pi\Delta_{sr} f_c + 1}. \tag{5.6}$$

The highpass transfer function is given by $H_{HP}(z) = \frac{\alpha - \alpha z^{-1}}{1 - \alpha z^{-1}}$ and its magnitude response is shown in Figure 5.6.

5.3.4 Onset Detection

For triggering notes, a note onset detection method was developed. The idea behind this method was to rely not on the target source signals, but rather on pitch information, which may later be provided from a polyphonic pitch tracker for achieving true unsupervised learning. The f_0 and f_0 confidence obtained from CREPE [KSLB18] served as basis

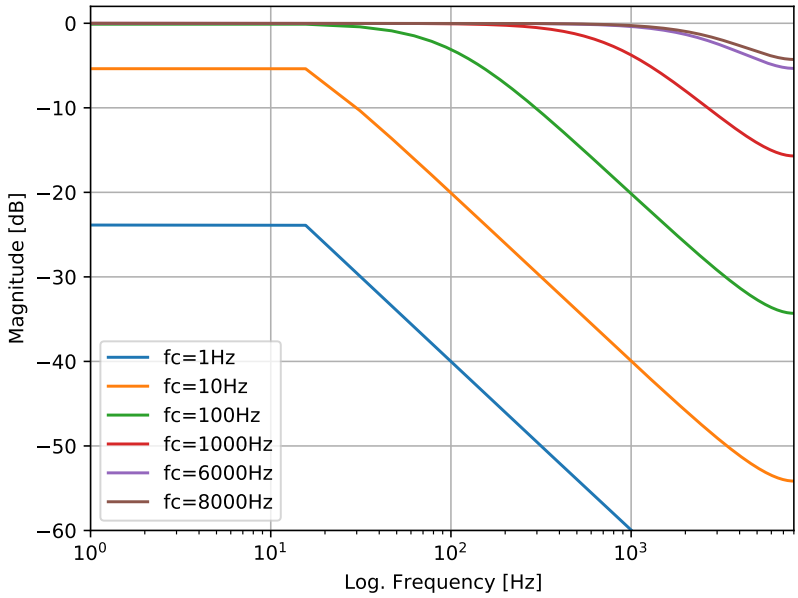


Figure 5.6 – Magnitude response of the highpass $h_{HP}[n]$ at $f_{sr} = 16kHz$ for different cutoff frequencies f_c.

for the onset detection. The following algorithm has been developed and used in the experiments [2].

Define a confidence threshold $c_{th} \in [0, 1]$ and get the fundamental frequency $f_0[m]$ and its confidence $c[m]$ for every frame m of the current training example. Set f_0 to zero if the confidence is below the threshold

$$f_{0out}[m] = \begin{cases} f_0[m] & c \geq c_{th} \\ 0 & else \end{cases} . \tag{5.7}$$

Restrict the signal to an interval of $[0, 1]$

$$x_{onoff}[m] = \begin{cases} 1 & f_{0out}[m] > 0 \\ 0 & else \end{cases} . \tag{5.8}$$

Take the backward difference $x_{diff}[m] = x_{onoff}[m] - x_{onoff}[m-1]$ and distinguish between note onsets and offsets by clipping the signal

$$x_{on}[m] = \begin{cases} x_{diff}[m] & x_{diff}[m] \geq 0 \\ 0 & else \end{cases} . \tag{5.9}$$

2. In hindsight this algorithm was not an ideal choice for this problem, since more sophisticated methods exist.

Finally get the onset frame indices $m_{on} = arg(x_{on}[m] \neq 0)$. In the experiments a confidence threshold of $c_{th} = 0.4$ has been used.

5.3.5 Neural Network Architecture

The NN architecture from preliminary experiment 2 in Figure 5.3 is extended with f_0 information as in [SFDRB22]. This NN architecture is depicted in Figure 5.7. The target mixture magnitude spectrogram along with the fundamental frequency f_0 of every target source is fed to the first layers. From this input, f_c is predicted for every frame of the source estimate. f_c is scaled to a range from 0Hz to the Nyquist frequency $f_{sr}/2$. Again the multi scale spectral loss L_{MSS} is employed as loss function.

5.3.6 Source Masking

The *directly synthesized sources* $\hat{s}_i[n]$ can also be used for masking the mix signal $x[n]$ to obtain higher quality *masked sources* $\tilde{s}[n]$. This is done by obtaining soft masks from $\hat{s}_i[n]$.

The signals are transformed into the frequency domain $\hat{S}_i[k,m] = STFT\{\hat{s}_i[n]\}$, $\hat{X}[k,m] = STFT\{\hat{x}[n]\}$ with frequency bins k and time frames m. The mask is obtained from the source estimates magnitude spectrogram and the magnitude spectrogram of the estimated mix by element-wise division

$$mask_i[k,m] = \frac{|\hat{S}_i[k,m]|}{\sum_i |\hat{S}_i[k,m]| + \epsilon}$$

[SFDRB22, p.6] with $\epsilon = 10^{-12}$ to avoid dividing by zero. The masked source spectrogram $\tilde{S}_i[k,m]$ is computed via

$$\tilde{S}_i[k,m] = mask_i[k,m] \odot X[k,m]$$

with the element wise product $\odot$, implicitly containing the phase information of the mix signal. Finally the time signal is retrieved $\tilde{s}[n] = iSTFT\{\tilde{S}_i[k,m]\}$.

In this experiment source masking is exclusively used at inference for producing the masked sources $\tilde{s}[n]$. Only the synthesized source estimates $\hat{s}_i[n]$ are used in training for forming the predicted mixture signal and computing the loss from it.

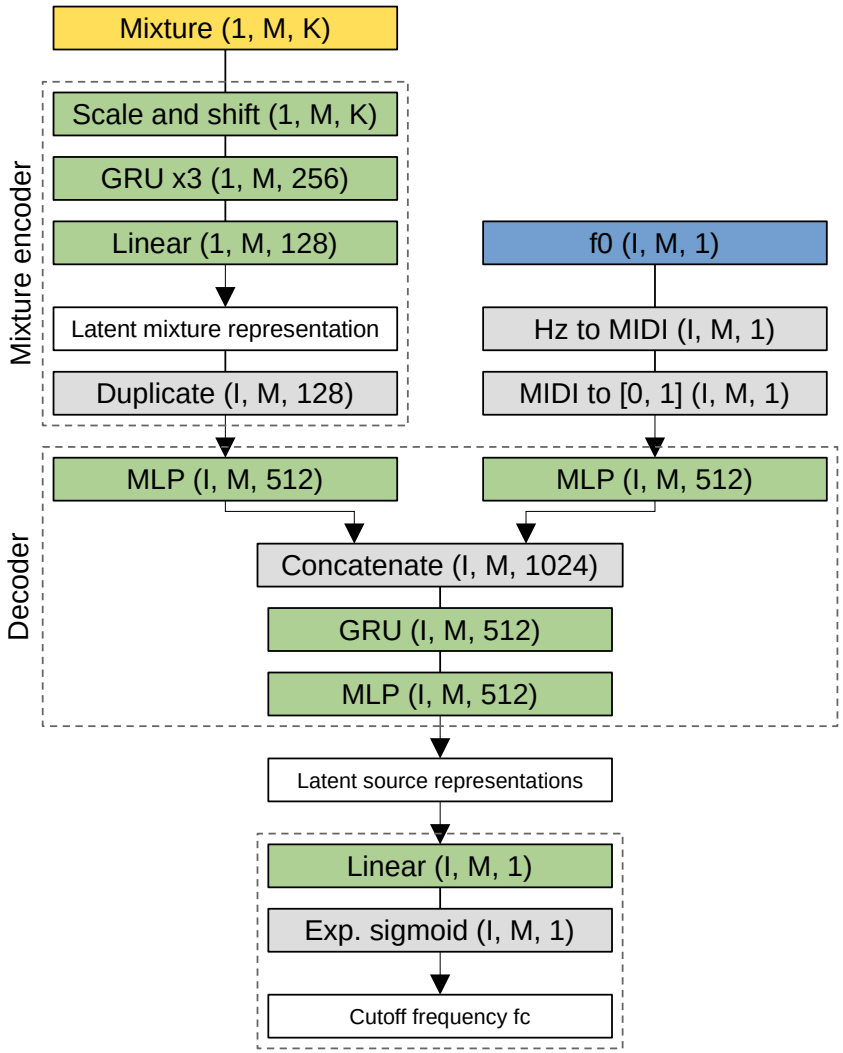


Figure 5.7 – NN architecture (adapted from [SFDRB22]). M is the number of time frames, K is the number of frequency bins and I is the number of sources (strings). "Transformations with learnable parameters are shown in green, predefined processing steps in gray, (intermediate) outputs in white boxes. The output shape of a transformation is shown in the right part of the box." [SFDRB22]

5.4 Experiment B Series

As a next step the simple Karplus-Strong string model from above is extended as described in [JS83]. The experiment B series contains 5 experiments with successive extensions to the physical source model. The dataset was the same as in the experiment A series. Since validation losses in Figure 6.11 did not significantly drop after the second epoch in the previous experiments, the models of the B series were only trained for 2 epochs. This was done due to training time efficiency for yielding results which enable an estimation of the models performances.

Experiment Overview

— ExB0: Baseline without neural predictions.
— ExB1: a ... excitation amplitude and false onset filtering.
— ExB2: a and ρ ... feedback coefficient.
— ExB2.1: Follow up experiment to ExB2 with non-linear feedback clipping for stability.
— ExB3: a, b ... stretching factor (coefficient for the feed-forward filter).
— ExB4: a, b, r ... excitation dynamics (coefficient for the excitation filter).

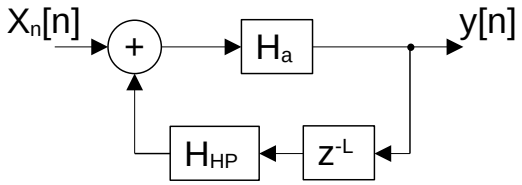


Figure 5.8 – Block diagram of the signal model employed in ExB0.

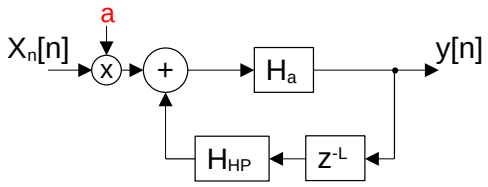


Figure 5.9 – Block diagram of the signal model employed in ExB1.

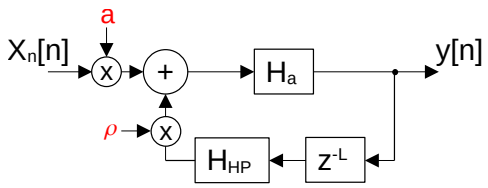


Figure 5.10 – Block diagram of the signal model employed in ExB2.

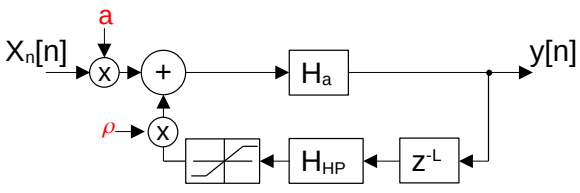


Figure 5.11 – Block diagram of the signal model employed in ExB2.1.

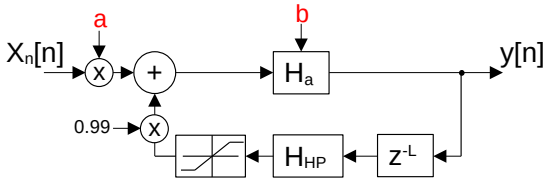


Figure 5.12 – Block diagram of the signal model employed in ExB3.

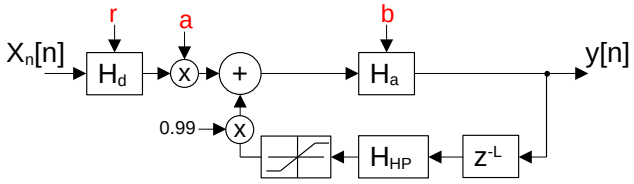


Figure 5.13 – Block diagram of the signal model employed in ExB4.

Experiment B0 Figure 5.8 shows the simple Karplus-Strong signal model employed in experiment ExB0. It is the same model as in ExA but with a different string damping lowpass filter H_a. In this experiment the time invariant lowpass H_a is implemented with the difference equation

$$y[n] = 0.5x[n] + 0.5x[n-1] \tag{5.10}$$

as defined in the original Karplus-Strong algorithm [KS83].

Experiment B1 In experiment ExB1 the baseline was extended with a neurally predicted control signal a as shown in Figure 5.9. a is multiplied with the excitation signal $x_n[n]$ to control the *excitation amplitude* of played notes. Furthermore the NN may be capable of filtering out wrong onsets generated by the onset detection.

Experiment B2: ExB2 in Figure 5.10 is extended with the neurally predicted feedback coefficient ρ, which is used for *note decay shortening* and causing *note ends*.

Experiment B2.1: ExB2.1 depicted in Figure 5.11 is the follow up experiment to Ex2 which showed stability problems at training time. Hence to guarantee stability the feedback path values are clipped [3] to $[-1, 1]$. Since experiments involving the feedback factor ρ showed worse performance than ExB1, ρ has been omitted in the following experiments.

Experiment B3: In ExB3 the signal model in Figure 5.12 receives the neurally predicted excitation amplitude a and the neurally predicted *decay stretching* coefficient b.

3. Clipping is done with a "hard tanh function" [PyT23] which is differentiable in the practical Autodiff sense like the ReLU function that is used excessively in DNNs. Hereby, the gradient of the constant parts of the function is set to zero including the two mathematically non-differentiable points.

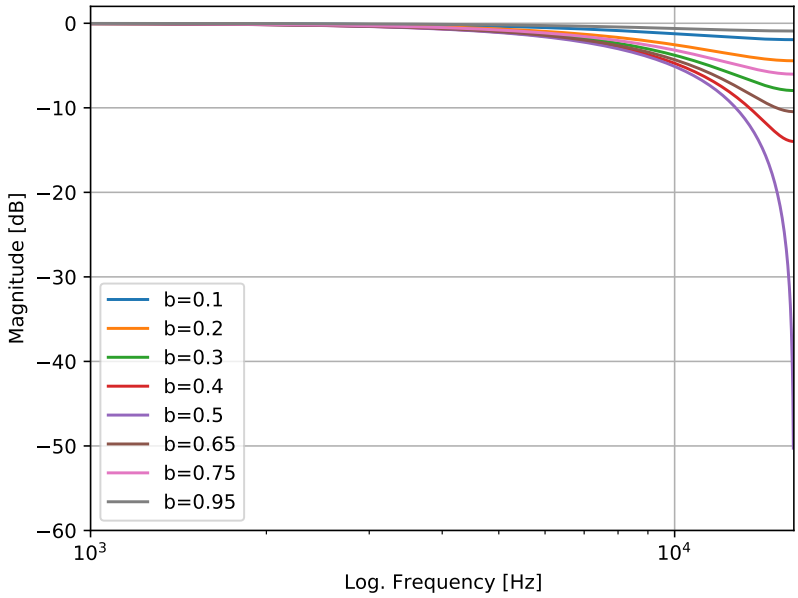


Figure 5.14 – Magnitude response of the original string damping filter $H_a(z)$ from [KS83] at $f_{sr} = 32kHz$ for different coefficients b.

Additionally the feedback coefficient is fixed to $\rho = 0.99$. The decay stretching one-zero FIR lowpass filter H_a is time variant in this experiment and is implemented with the difference equation

$$y[n] = (1 - b)x[n] + bx[n - 1] \tag{5.11}$$

as proposed in [JS83]. H_a influences the note decay and timbre. Its transfer function is given by $H_a(z) = (1 - b) + bz^{-1}$ with magnitude response

$$|H_a(e^{j\theta})| = \sqrt{((1 - b) + bcos(\theta))^2 + (bsin(\theta))^2}. \tag{5.12}$$

It has unit magnitude at $b = 0$, since $|H_a(e^{j\theta})| = \sqrt{1} = 1$ and at $b = 1$, since $|H_a(e^{j\theta})| = \sqrt{cos^2(\theta) + sin^2(\theta)} = 1$, which is also identifiable from the difference equation in eq. (5.11). Hence, $b = 0$ and $b = 1$ are avoided for ensure stability of the Karplus-Strong Feedback loop. The magnitude response of is shown in Figure 5.14. Since it is a one-zero FIR filter, the attenuation is subtle above $1kHz$ at a sampling rate of $f_{sr} = 32kHz$.

Experiment B4: ExB4 extends the signal model of ExB3 with the excitation dynamics filter H_d as depicted in Figure 5.13. The one-pole IIR lowpass filter H_d is given by the difference equation

$$y[n] = (1 - r)x[n] + ry[n - 1] \tag{5.13}$$

as proposed in [JS83].

Figure 5.15 shows the NN architecture used for the experiment B series. Again it is an extension of the NN architecture used in experiment series A.

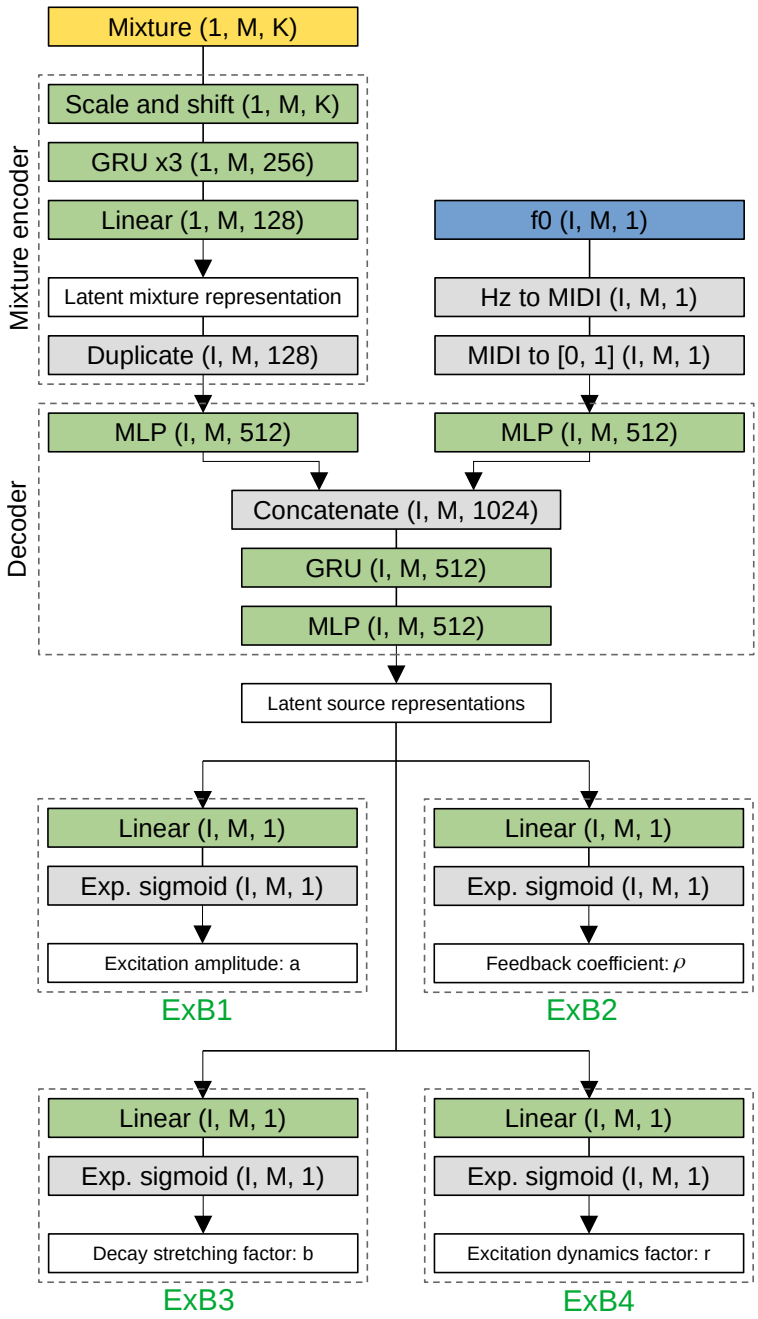


Figure 5.15 – NN architecture for the experiment B series. The experiments introducing the neural control signal are indicated in green text.

5.5 Experiment C Series

In this experiment series, the same training procedure and dataset as in experiment series A and B were used. The goal was to improve the model from ExB4.

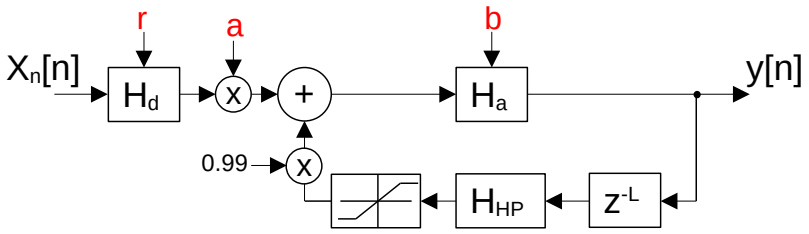


Figure 5.16 – Block diagram of the signal model used in ExC1.

The source model for ExC1 is depicted in Figure 5.16. It is the same source model used in ExB4. However in ExC1 the fundamental frequency f_0 at the time of the onset was held constant up to the next onset. The idea behind this strategy was that fast f_0 changes after the excitation may result in the strong note amplitude decrease, observed in experiment series A and B. Since the Karplus-Strong model is essentially a comb filter, a fast detuning after the excitation of this filter may cause a loss of excitation energy.

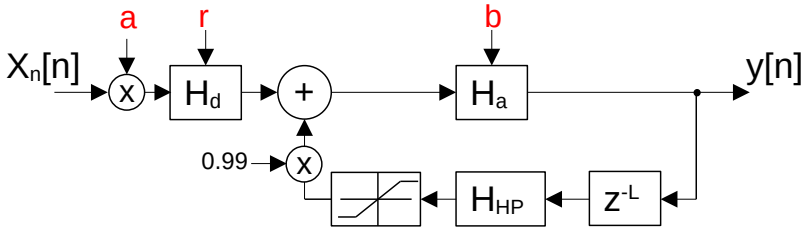


Figure 5.17 – Block diagram of the signal model used in ExC2.

The source model for ExC2 is depicted in Figure 5.17. Here the multiplication with a and the filter H_d are swapped. In this experiment the onset detection was deactivated but the model was excited with constant white noise $x_n[n]$ of amplitude 1. The excitation amplitude a was predicted with 64 samples per time frame m and not like previously with one sample per time frame. This is an attempt for the DNN to neurally predict note onsets by its own.

The neurally predicted control signal a may contain discontinuous parts which do not correspond to the physical actuality. Hence, to smooth these potential discontinuities, the lowpass for excitation dynamics was also used to smooth a.

Figure 5.18 shows the NN architecture used in experiment series C. For ExC1 the parameter $A = 1$ and for ExC2 $A = 64$.

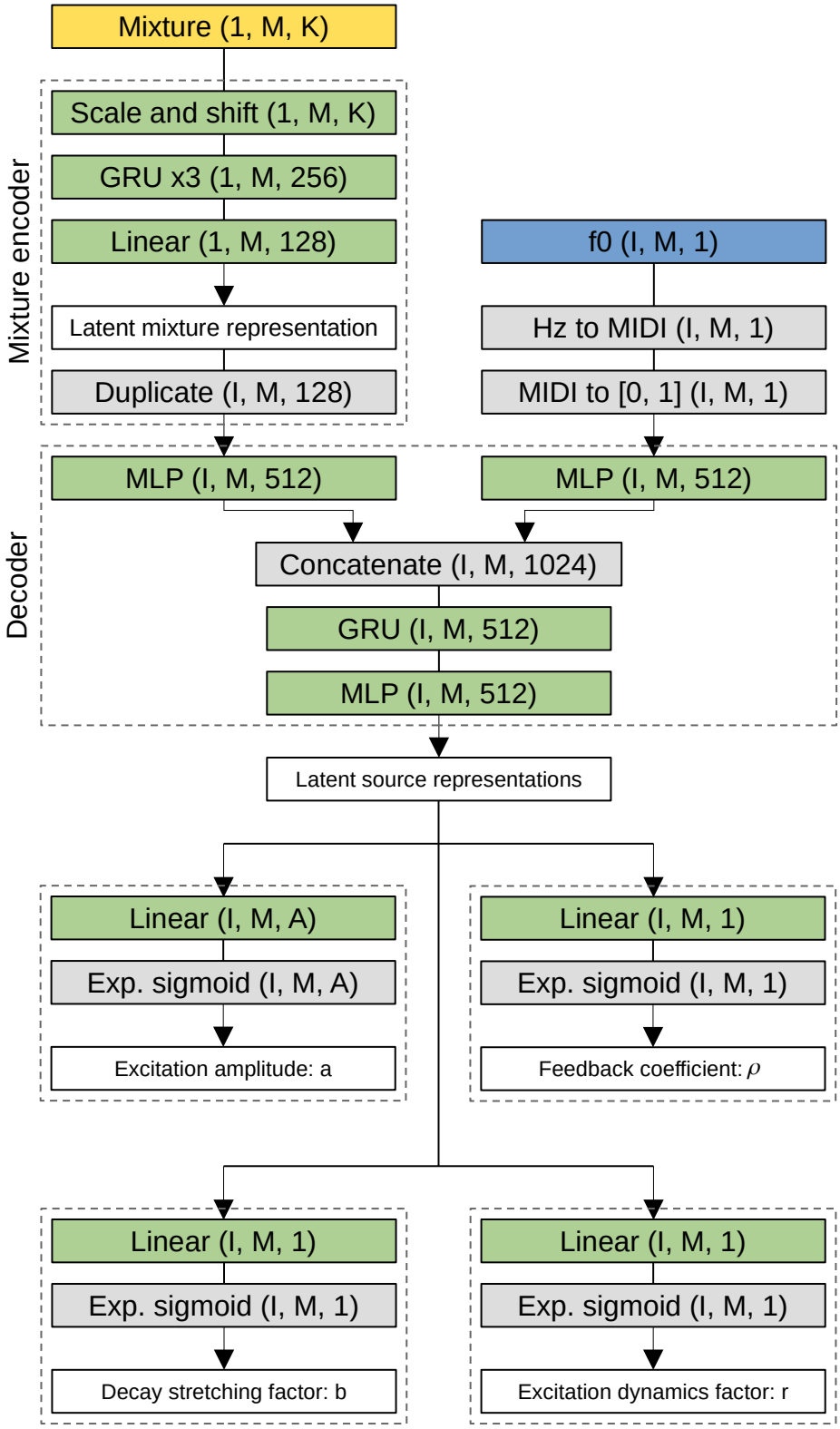


Figure 5.18 – NN architecture of the experiment C series.

5.6 Metrics

As performance measures the metrics *scale invariant signal-to-distortion-ratio* (SI-SDR) and the *Mel Cepstral distance* (MCD) have been used. The SI-SDR was used in [SFDRB22] as a MSS performance measure and the MCD serves as a timbre similarity measure.

As defined in [LRWEH19] the SI-SDR is given via

$$SI\text{-}SDR = \frac{\sum_n (\alpha x[n])^2}{\sum_n (\alpha x[n] - \hat{x}[n])^2} \tag{5.14}$$

and in dB

$$SI\text{-}SDR_{dB} = 10 log_{10}(SI\text{-}SDR) \tag{5.15}$$

with the parameter

$$\alpha = \frac{\sum_n x[n]\hat{x}[n]}{\sum_n x^2[n]}. \tag{5.16}$$

The MCD as defined in [Kub93] is the arithmetic mean over all time frames of the euclidean distance of the Mel frequency Cepstral coefficients (MFCC) between two signals. The MCD per time frame m of signals $x[n]$ and $y[n]$ is given via

$$MCD_{x,y}[m] = \sqrt{\sum_k (MFCC_x[m,k] - MFCC_y[m,k])^2} \tag{5.17}$$

with the time frame index m and the coefficient index k. The MCD is then calculated with

$$MCD_{x,y} = \frac{1}{M} \sum_m MCD_{x,y}[m] \tag{5.18}$$

with the number of time frames M.

Chapter 6

Results

In this chapter, the results of the conducted experiments are presented which are described above, in chapter 5. Every section is closed with an interpretation of these results.

6.1 Recreation of the Original Method

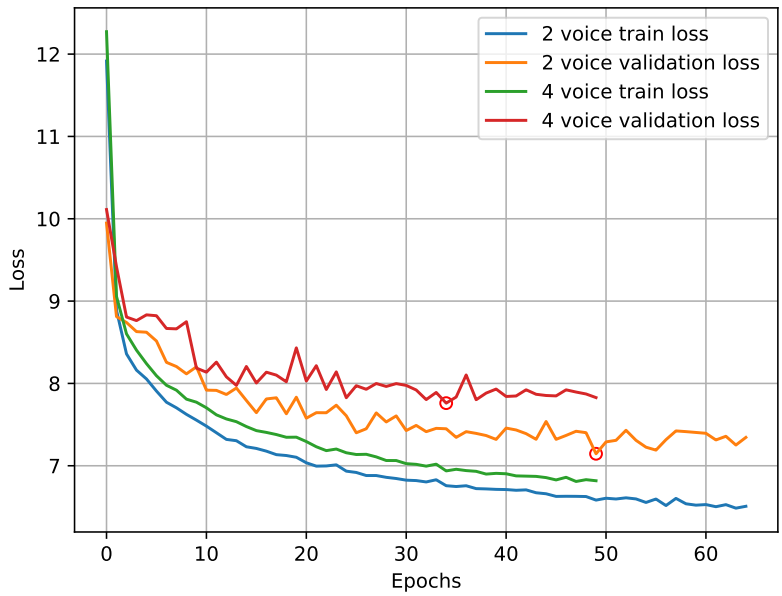


Figure 6.1 – Learning curves for recreating the results from [SFDRB22]. The best epoch with minimal validation loss is indicated with a red circle.

Model configurations have been the same as described in [SFDRB22]. The models have been trained until early stopping was triggered with a patience of 15 epochs. Batch sizes were chosen as 16 for the two-voice-model and 14 for the 4-voice-model to max out

available RAM. The two-voice-model was trained longer (50 epochs) than the 4-voice-model (35 epochs) and accomplished a lower validation loss in its best epoch. This was to be expected since the separation of 2 voices poses a simpler problem than the separation of 4 voices from a mix. Figure 6.2 shows the resulting SI-SDR for the trained models.

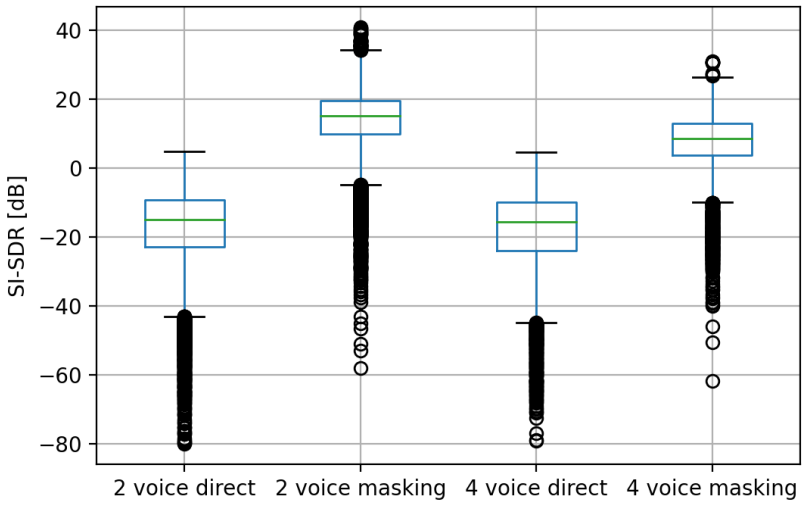


Figure 6.2 – Box plot of the SI-SDR of the trained models separating the test set for recreating the results in [SFDRB22].

Overall the masked sources performed better than the directly synthesized sources, which was to be expected from theory. The recreated 2-voice-masking model achieved a mean SI-SDR of 14.1dB whereas the supervised and unsupervised 2-voice-masking models from [SFDRB22] achieved a mean SI-SDR of 13.5dB. The recreated 4-voice-masking model achieved a mean SI-SDR of 7.3dB whereas, from [SFDRB22], the supervised 4-voice-masking model reached 6.5dB SI-SDR and the original unsupervised 4-voice-masking model had 6.7dB SI-SDR.

Hence, the results of [SFDRB22] have been successfully recreated. There are at least two reasons why the recreated models show a higher SI-SDR of approximately 1dB than the original models, which suggests a slightly better separation performance.

1. Better pitch information has been used in the recreation by tracking monophonic sources with CREPE instead of performing multiple f_0 analysis as in [SFDRB22].
2. The same dataset has been used for training and testing. Although different songs were used for training and testing, they share common features such as musical style, recording style, room impulse response and others.

For that reason the performance of the recreated models and the models of the original publication [SFDRB22] can be estimated as equal, although using different datasets and f_0 trackers.

6.2 Preliminary Experiment: Karplus-Strong Resynthesis

For the synthetic string sound resynthesis, the first models have been trained with randomly seeded white noise as excitation signal. This lead to not exactly reproducible sounds, since the form of the excitation signal has a major impact on the timbre of the synthesized string sound. For this reason also a *reproducible Karplus-Strong model* has been implemented which uses always exactly the same excitation signal for sound synthesis. The excitation signal always consists of white noise which is a random signal, but the random excitation is randomly seeded whereas the reproducible excitation signal stays always the same. Both model variants were used to create a dataset for resynthesis. Models have then been trained with the following configurations.

— Random excitation dataset with random excitation synthesis model.
— Reproducible excitation dataset with random excitation synthesis model.
— Reproducible excitation dataset with reproducible excitation synthesis model.

In the following, the learning curves are shown. Furthermore the average predicted cutoff frequency f_c per epoch for the validation data is depicted (right y-axis). The target $f_c = 6000Hz$ is indicated as a green horizontal dashed line. Models have been trained with different learning rates, because this had a big influence on training behaviour. The model names are numbered in the arbitrary order they were trained.

Random Excitation for Input and Output

The models in this section used the random excitation dataset for training as well as randomly seeded excitations for synthesis.

The first models training result is depicted in Figure 6.3. This model has been trained with a *learning rate* of $1 \cdot 10^{-3}$ and shows an erratic training behavior. The training loss of Model 1 shows a decreasing trend and its validation loss decreases at the beginning, but increases at the end. The predicted f_c overshoots at the beginning and continues around an offset from the target value.

Model 9 was trained with a learning rate of $5 \cdot 10^{-6}$, which turned out to be an appropriate value for this configuration. Its training loss in Figure 6.4 decreases fast in the first epochs and continues around a limit. The validation loss decreases first but increases later. The predicted f_c approaches the target value in the first epochs and settles around an offset from it.

Reproducible Excitation for Input and Random Excitation for Output

The models in this section used the reproducible excitation dataset for training, but randomly seeded excitation signals for synthesis.

Model 8 has been trained with a learning rate of $1 \cdot 10^{-3}$. Training and validation loss in

Figure 6.5 decrease at the beginning and fluctuate around a limit. Also the f_c prediction approaches the target value and fluctuates around it.

The next three models are trained with decreasing learning rates. Model 4 (Figure 6.6) has a learning rate of $1 \cdot 10^{-5}$, model 6 (Figure 6.7) has a learning rate of $5 \cdot 10^{-6}$ and model 5 (Figure 6.8) has a learning rate of $1 \cdot 10^{-6}$.

Reproducible Excitation of Input and Output

The models in this section used the reproducible excitation dataset for training and reproducible excitation for synthesis. Model 7 has a learning rate of $1 \cdot 10^{-3}$ and model 10 has a learning rate of $5 \cdot 10^{-6}$. Model 7's losses in Figure 6.9 drop to an intermediate value and then drop to a final value. Its predicted f_c overshoots the target value and settles at an offset of about 900Hz from it. Model 10's losses in Figure 6.10 drop immediately to a limit and the predicted f_c does not overshoot but also settles at an offset from the target value at about 700Hz from it.

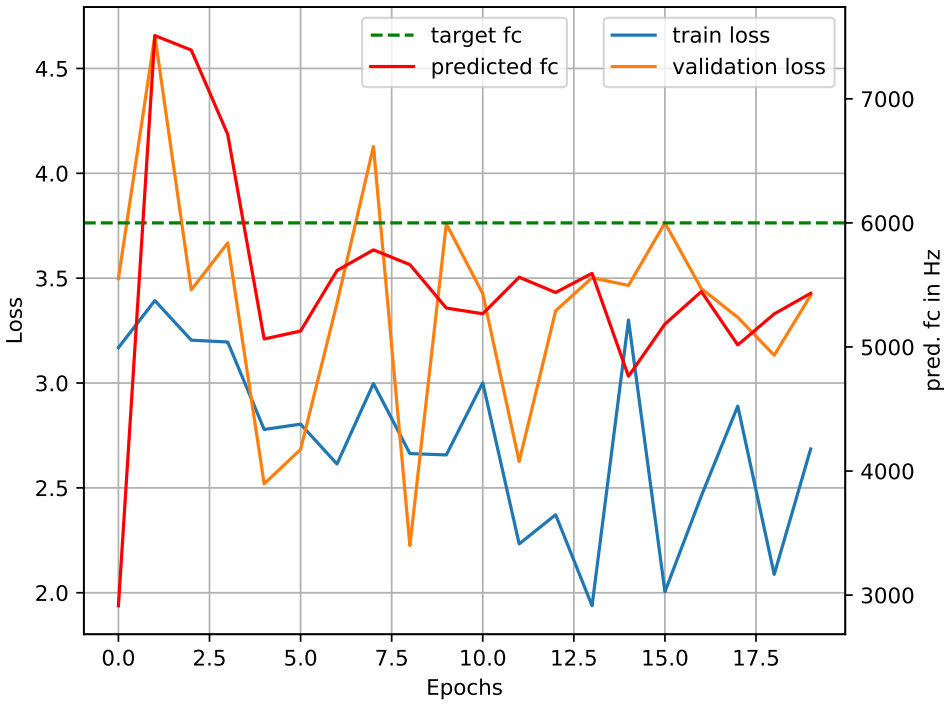


Figure 6.3 – Model 1 training (random in, random out), lr = 10^{-3}.

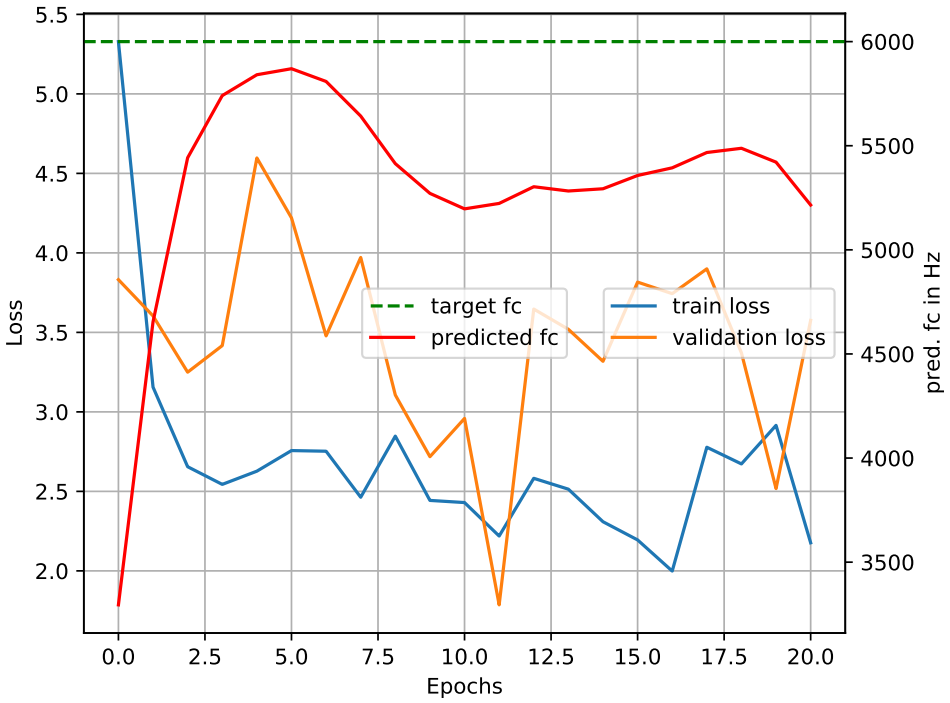


Figure 6.4 – Model 9 training (random in, random out), lr = $5 \cdot 10^{-6}$.

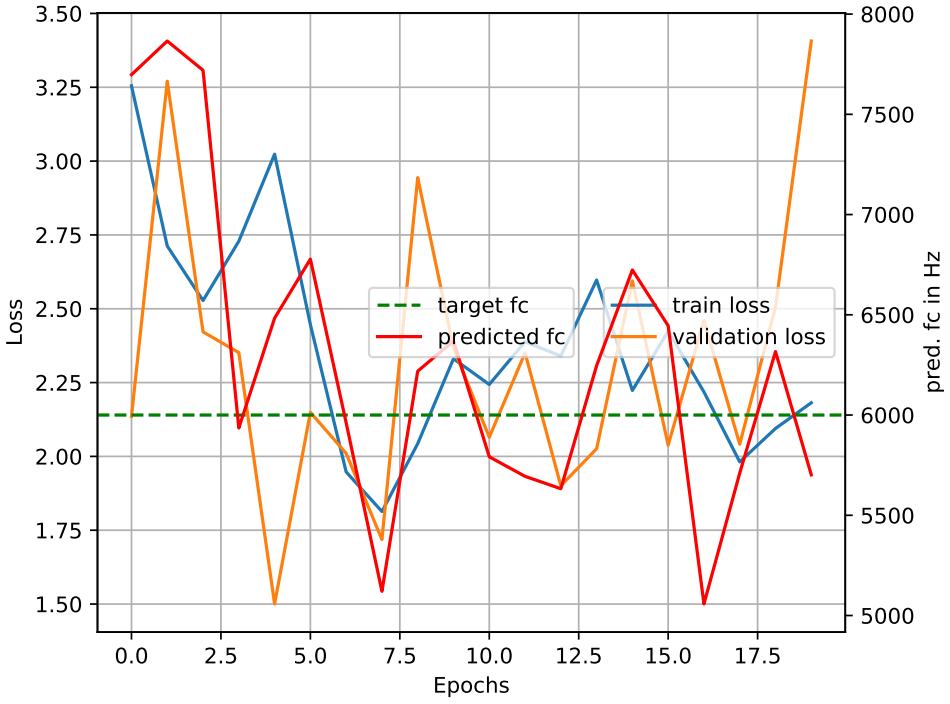


Figure 6.5 – Model 8 training (reproducible in, random out), lr = $1 \cdot 10^{-3}$.

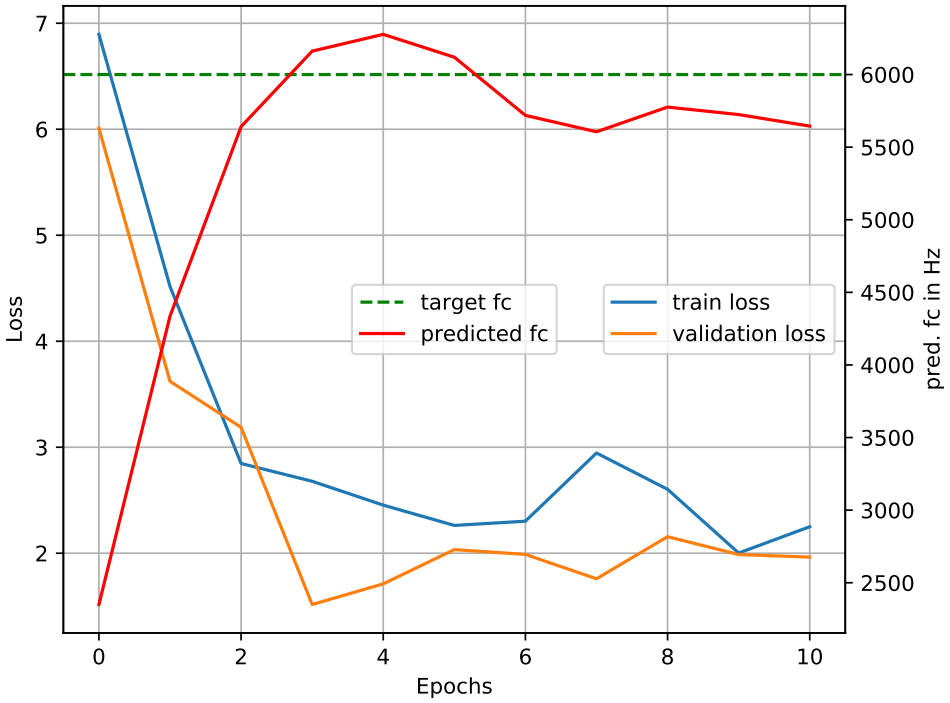


Figure 6.6 – Model 4 training (reproducible in, random out), lr = $1 \cdot 10^{-5}$.

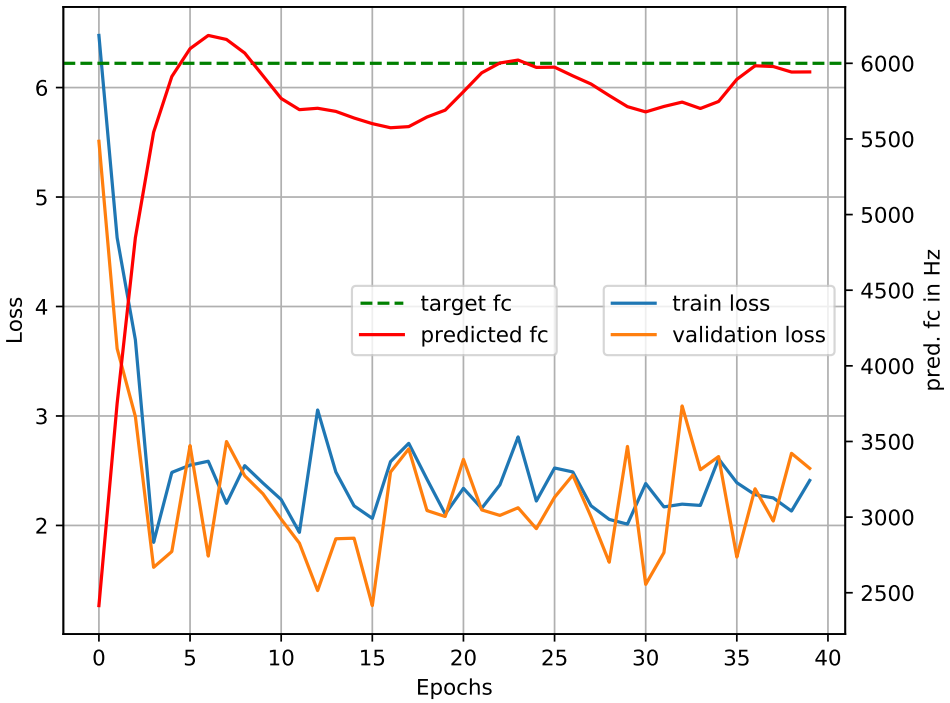


Figure 6.7 – Model 6 training (reproducible in, random out), lr = $5 \cdot 10^{-6}$.

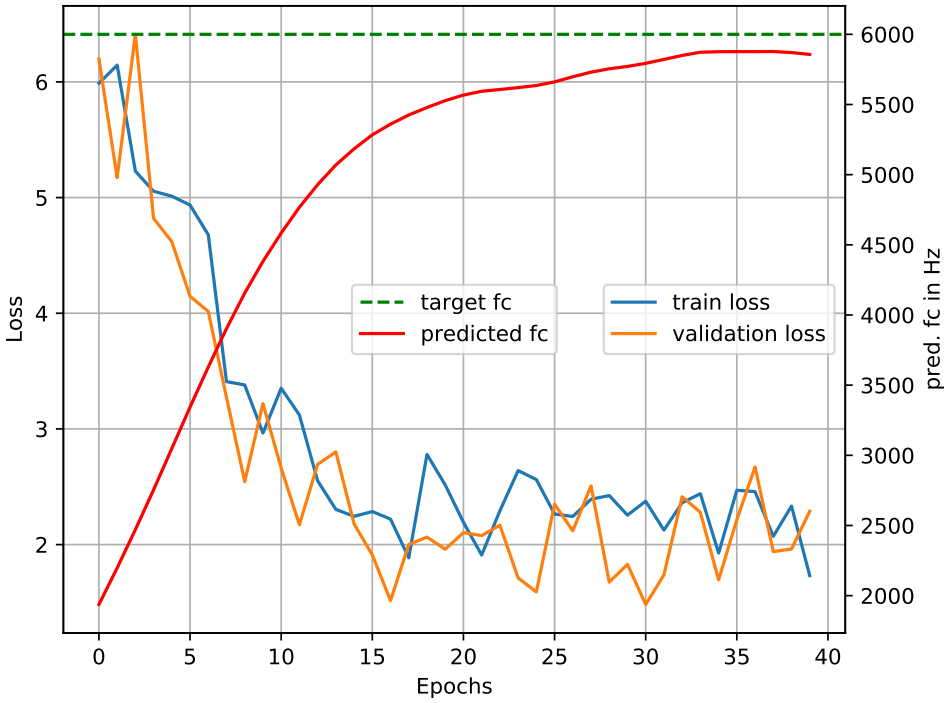


Figure 6.8 – Model 5 training (reproducible in, random out), lr = $1 \cdot 10^{-6}$.

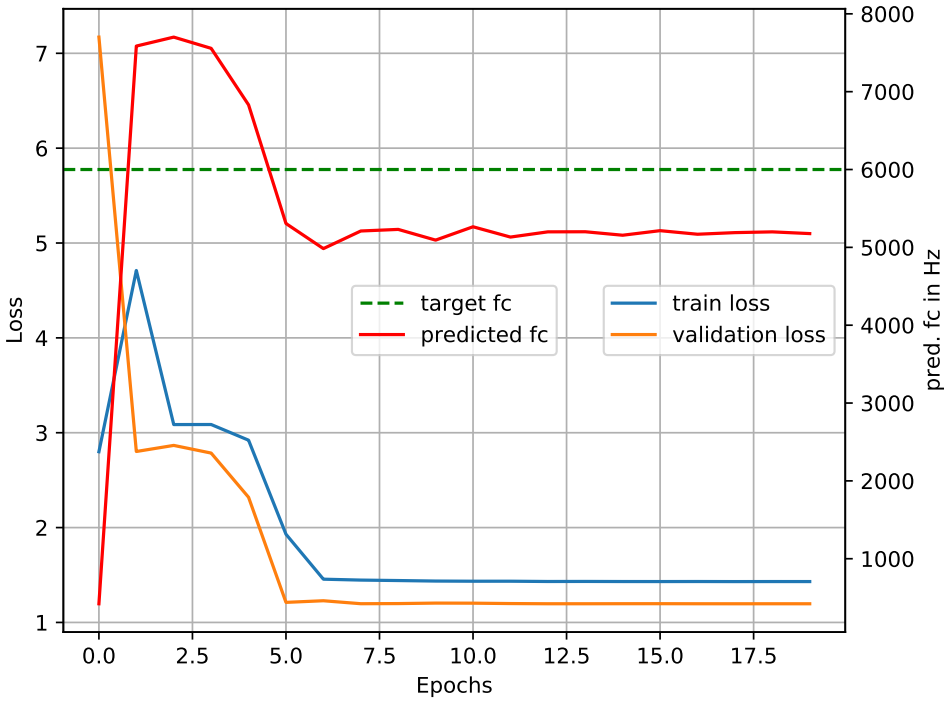


Figure 6.9 – Model 7 training. reproducible in, reproducible out, lr = $1 \cdot 10^{-3}$.

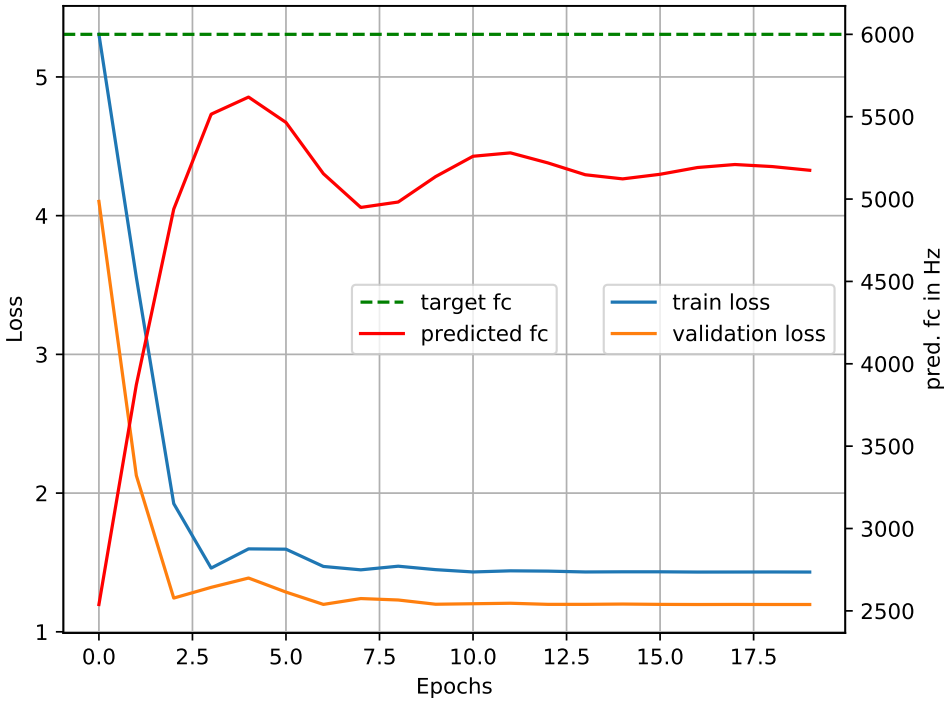


Figure 6.10 – Model 10 training (reproducible in, reproducible out), lr = $5 \cdot 10^{-6}$.

Models 1 and 9 using random excitation signals for input and output show a similar behavior. Both predicted f_c stay at an offset from the target value. A noticeable feature in these graphs is that a good prediction of f_c does not lead to a small validation loss. This indicates, that the output signal of the model can not reproduce the dataset signals with the right f_c prediction. Therefor, the reproducible excitation signal has been implemented to eliminate a sound influencing factor which is not accessible to the Karplus-Strong model.

Models 8, 4, 5, and 6 used reproducible excitation for input and random excitation for output. The strong fluctuations between epochs of model 8 around a target value are a sign of a too high learning rate. In a gradient descent based learning procedure the model parameters are updated according to the gradient on the error surface multiplied with the learning rate or step size to find a minimum. When this step size is too big, the steps overshoot the optimal solution (the minimum) and perform a random walk around it. This behavior is visible as performance fluctuations which can be decreased by choosing a smaller learning rate (step size).

Model 4, 5 and 6 show a similar behavior using smaller learning rates than model 8. The training and validation losses look like typical learning curves settling at a loss of about 2. Model 4 and 6 show an overshoot of the predicted f_c target value and model 5 slowly approaches the target value. The best performing model from this section is model 6 since it settles at the closest f_c predictions.

Models 7 and 10, using reproducible excitation for input and output, settle both at an f_c offset. This is an interesting result, since exactly the same excitation signal was used for creating the dataset and synthesizing the predictions.

Most learning curves in these experiments look rough. This is due to a small training set and even smaller validation set for fast training and validation.

These three experiments regarding the form of the excitation signal resulted in different learning behaviors. By taking the distance of predicted f_c as a performance measure, interestingly the models with reproducible input and output signals performed worst. They show smooth learning curves since the amount of randomness in the overall system is minimized but they may tend to get stuck in a local minimum of the error surface.

The models using reproducible input signals and randomly seeded output signals perform best with model 6 being the best model overall. The introduced randomness of the randomly seeded excitation signal may help the overall system to not get stuck in a local minimum but approaching the global minimum of the error surface and converge to the optimum solution to the stated problem.

None of the trained models shows diverging or unstable training behavior and all average predicted lowpass cutoff frequencies approach the target value in some kind. The results for this preliminary experiment show that the simple Karplus-Strong physical string model *is differentiable*. Furthermore the results show that it is possible to employ the DDSP signal model in a NN architecture such as in [SFDRB22] which can *effectively learn* from the training data.

6.3 Experiment A Series

The described models of the experiment A series have been trained for 5 epochs with the Adam optimizer and a learning rate of 10^{-4} using a batch size of 2 and loss FFT sizes $j =$ 2048, 1024, 512, 256, 128, 64. For the input mixture transform a FFT size of 512 with a hop size of 256 was used. The resulting learning curves are depicted in Figure 6.11.

Two metrics have been used to evaluate the performance of the trained model on the test set . These are the *scale independent source-to-distortion ratio* (SI-SDR) [LRWEH19] as in [SFDRB22] and the *Mel Cepstral distance* (MCD) [Kub93]. White noise source estimates have been used as a lower baseline and masking the mix signals with the ideal ratio mask (IRM) as described in section 3.3 has been used as an upper baseline. The resulting SI-SDR is depicted in Figure 6.12 and the MCD is depicted in Figure 6.13.

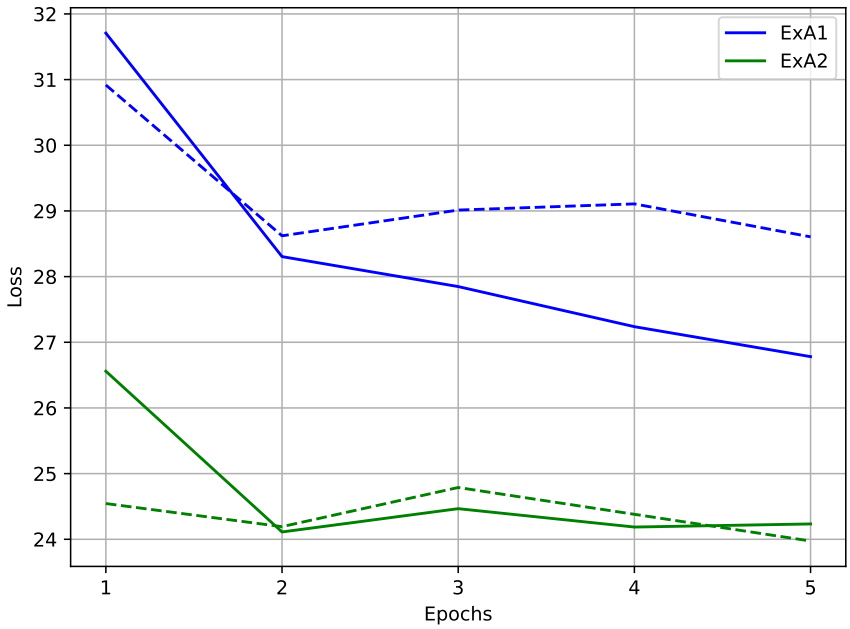


Figure 6.11 – Learning curves for ExA1: 16kHz and ExA2: 32kHz physical modeling sampling rate. Training loss in solid lines and validation loss in dashed lines.

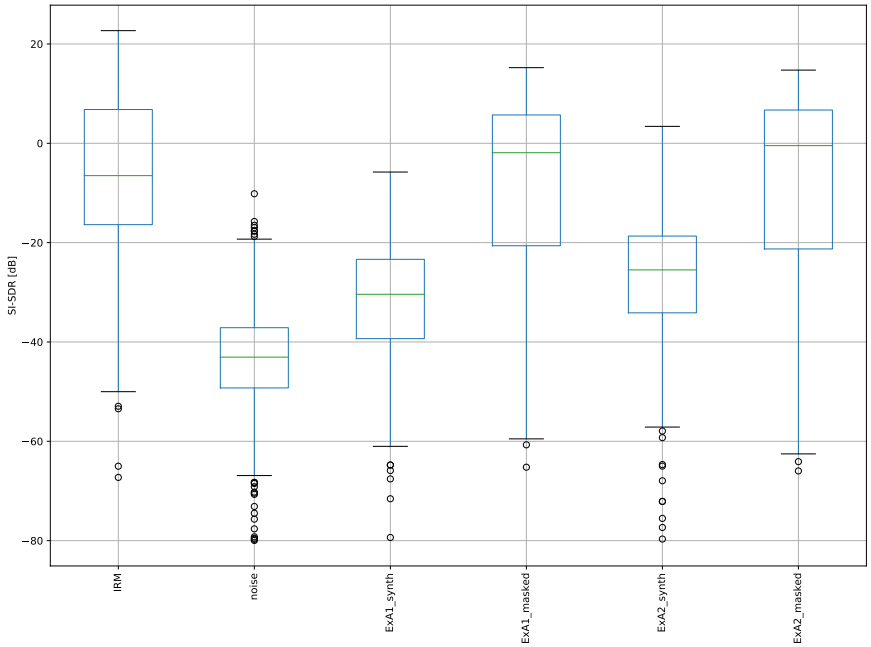


Figure 6.12 – SI-SDR box-plot for the lower baseline *noise*: white noise with amplitude 1, the upper baseline *IRM*: ideal ratio mask, *ExA1*: 16kHz physical modeling sampling rate, and *ExA2*: 32kHz physical modeling sampling rate. *synth*: directly synthesized sources $\hat{s}_i[n]$ and *masked*: masked sources $\tilde{s}_i[n]$.

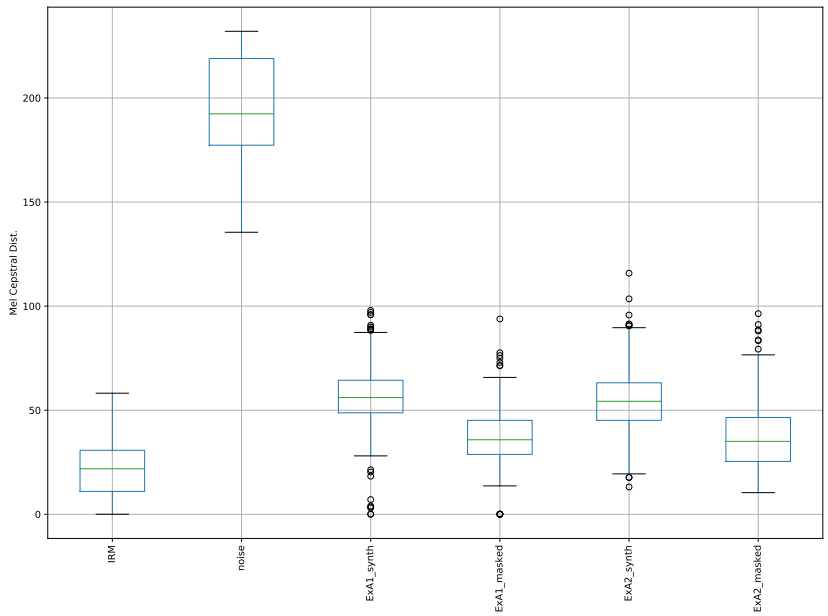


Figure 6.13 – MCD box-plot for the lower baseline *noise*: white noise with amplitude 1, the upper baseline *IRM*: ideal ratio mask, *ExA1*: 16kHz physical modeling sampling rate, and *ExA2*: 32kHz physical modeling sampling rate. *synth*: directly synthesized sources $\hat{s}_i[n]$ and *masked*: masked sources $\tilde{s}_i[n]$.

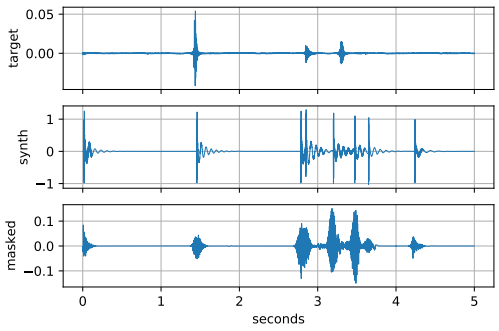
Synthesized Sources and Predicted f_c Synthesized and masked source estimates were computed from a song of the test set [1]. These output source signals $\hat{s}_i[n]$ and $\tilde{s}_i[n]$ are depicted in Figure 6.14 and 6.15 together with the target sources $s_i[n]$ for all strings. In this figure and all following the first 5 seconds of the file `02_BN1-129-Eb_comp_hex_cln.wav` from the test set is used for visualization. String numbering is done in the common guitar string numbering scheme from the highest string 1 (e), to string 2 (b), to string 3 (g) up to the lowest string 6 (e).

In Figure 6.16 and 6.17 the pitch tracker outputs $f_{0,i}$ and $f_{0,i}$ confidence are depicted together with the synthesized sources $\hat{s}_i[n]$ and the interpretable control signal for the cutoff frequency $f_{c,i}$ and the target source $s_i[n]$. For better scaling the predicted $f_{c,i}$ are depicted on a logarithmic scale in Figures 6.18 and 6.19 together with $\hat{s}_i[n]$.

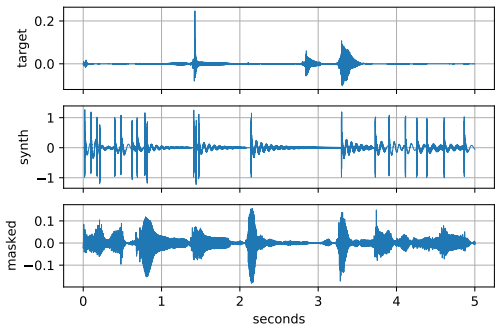
Training and validation loss of ExA1 is generally higher than for ExA2. Both learning curves decrease rapidly from the first to the second epoch. The training loss of ExA1 is steadily decreasing whereas the training loss of ExA2 stays around the limit of the second epoch.

The SI-SDR suggests equal performance of the IRM and the masked sources for ExA1 and ExA2. The median of the masked source estimates lies above the IRM median. As expected, the masked sources $\tilde{s}_i[n]$ achieve higher SI-SDR than the directly synthesized sources $\hat{s}_i[n]$.

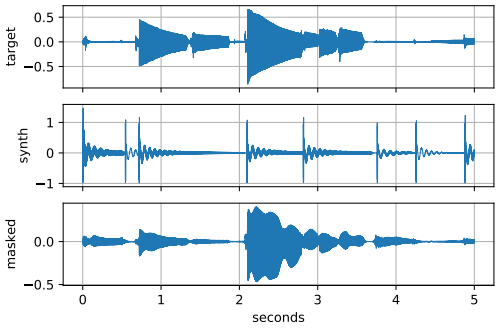
1. The file `02_BN1-129-Eb_comp_hex_cln.wav` has been used for inference.



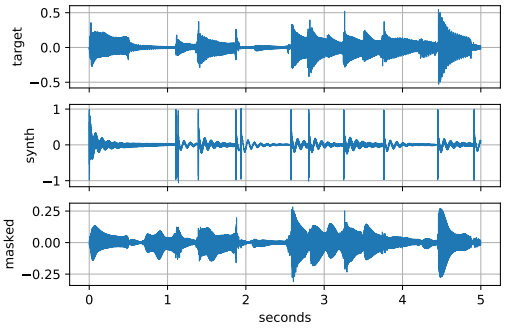
String 1



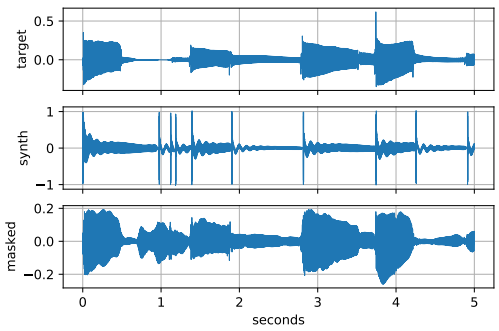
String 2



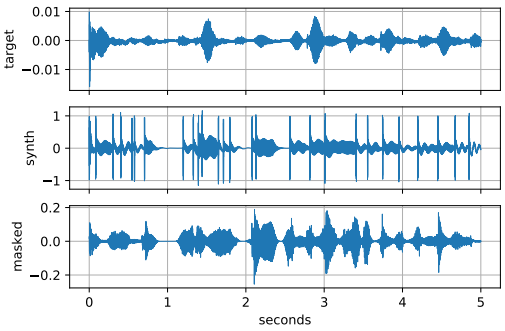
String 3



String 4

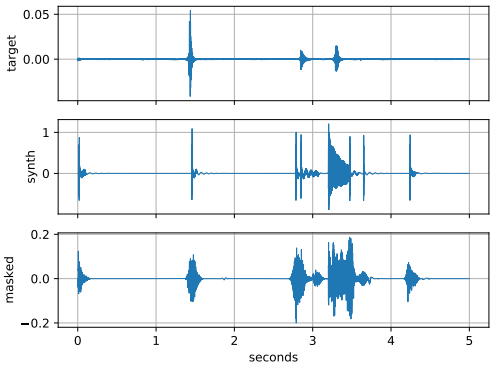


String 5

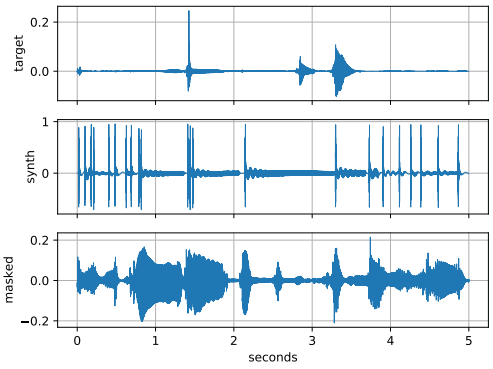


String 6

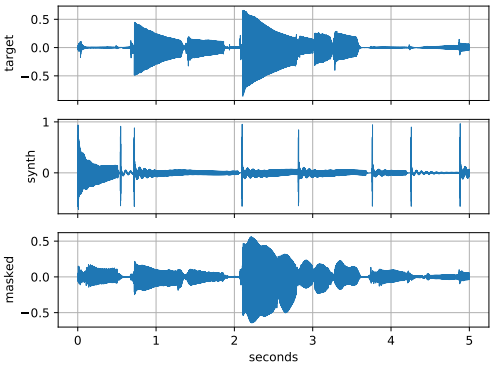
Figure 6.14 – ExA1 (16kHz) source estimate comparison for all strings. Target source (top), synthesized source estimate (middle) and masked source estimate (bottom).



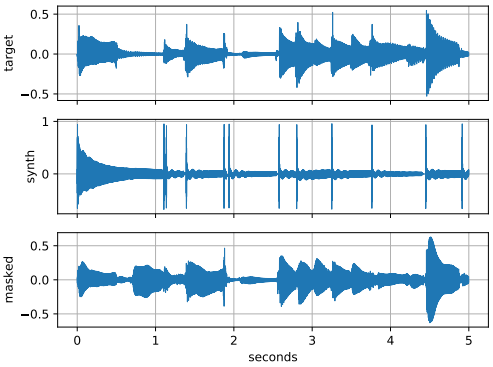
String 1



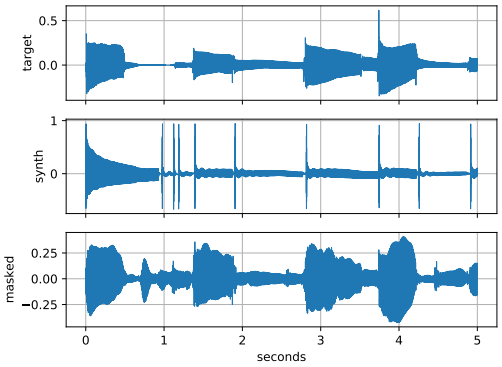
String 2



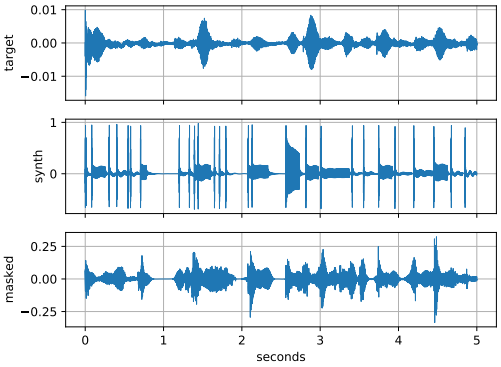
String 3



String 4

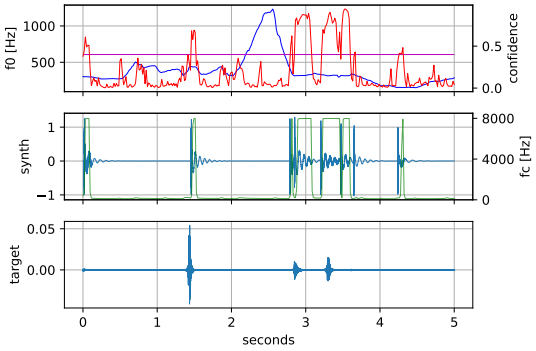


String 5

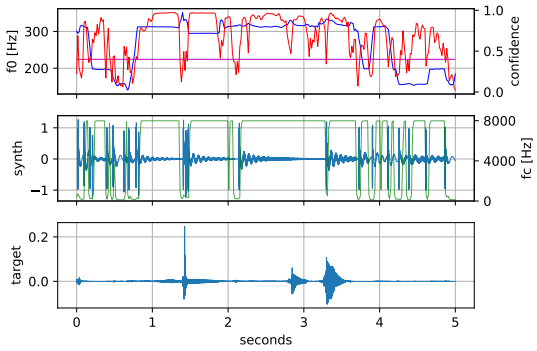


String 6

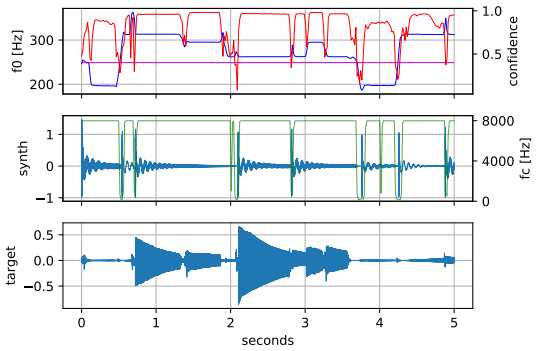
Figure 6.15 – ExA2 (32kHz) source estimate comparison for all strings. Target source (top), synthesized source estimate (middle) and masked source estimate (bottom).



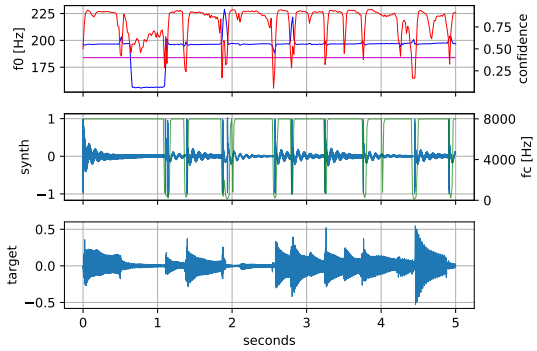
String 1



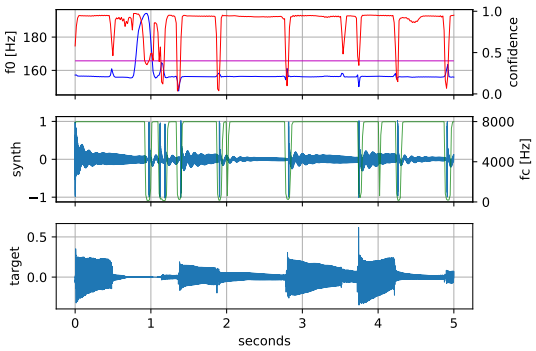
String 2



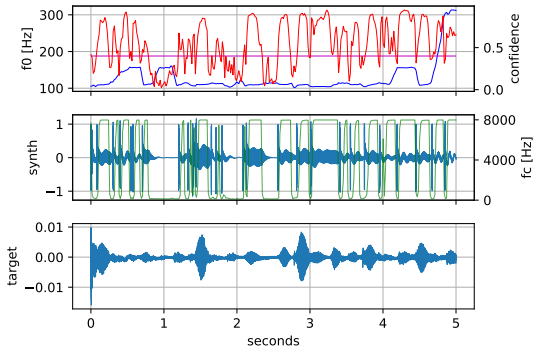
String 3



String 4

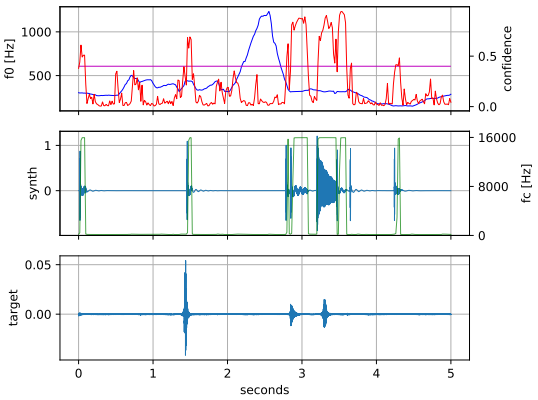


String 5

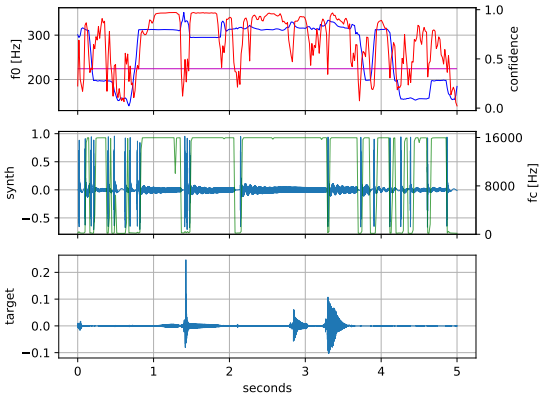


String 6

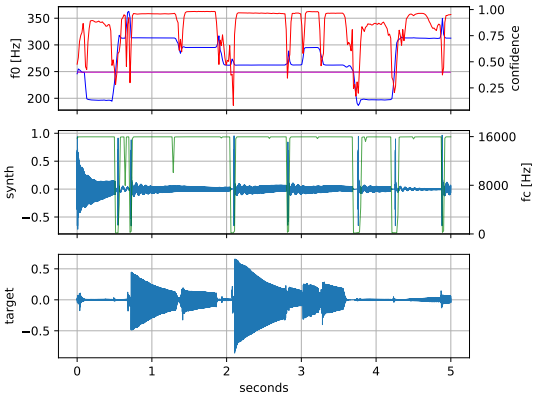
Figure 6.16 – ExA1 control signals and synthesized source estimate with comparison to the target source for all strings. f_0 in blue, f_0 confidence in red and the confidence threshold $c_{th} = 0.4$ in magenta (top), predicted f_c in green with synthesized source estimate in blue (middle), target source (bottom).



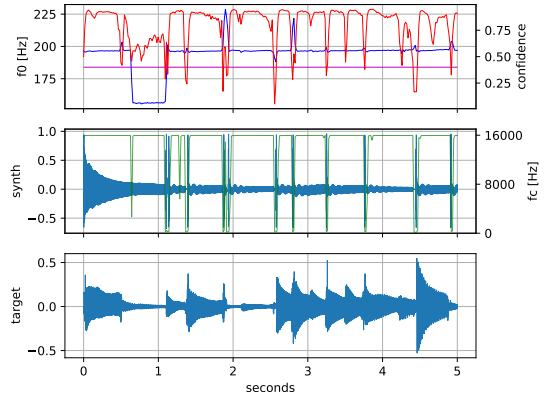
String 1



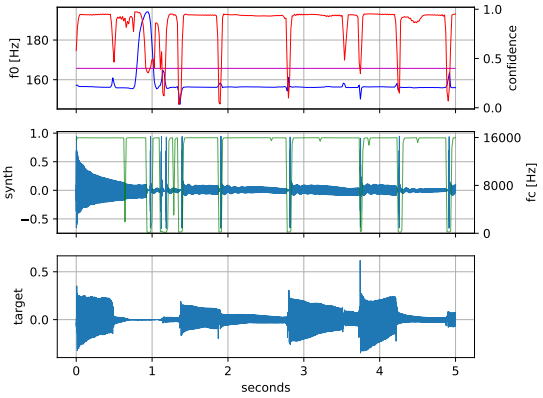
String 2



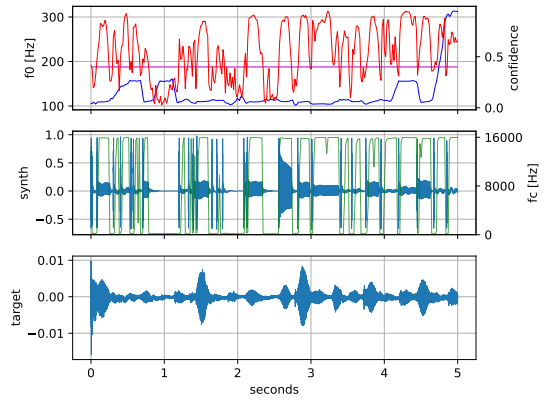
String 3



String 4

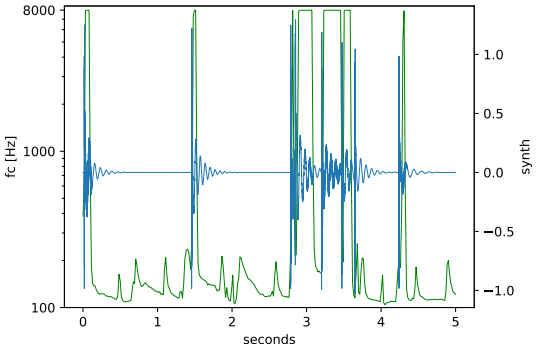


String 5

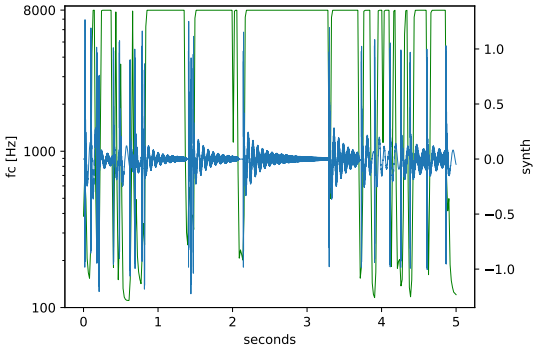


String 6

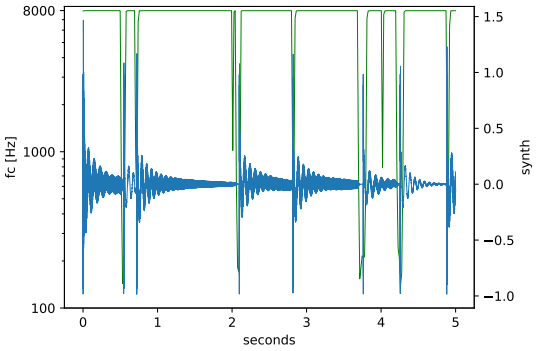
Figure 6.17 – ExA2 control signals and synthesized source estimate with comparison to the target source for all strings. f_0 in blue, f_0 confidence in red and the confidence threshold $c_{th} = 0.4$ in magenta (top), predicted f_c in green with synthesized source estimate in blue (middle), target source (bottom).



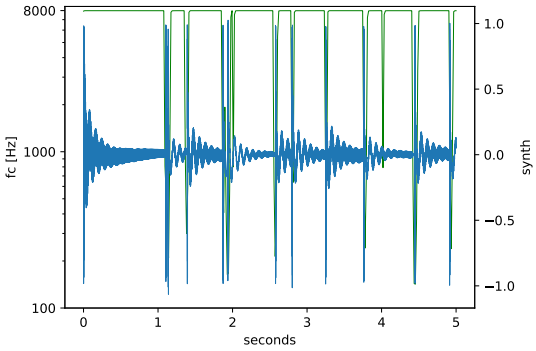
String 1



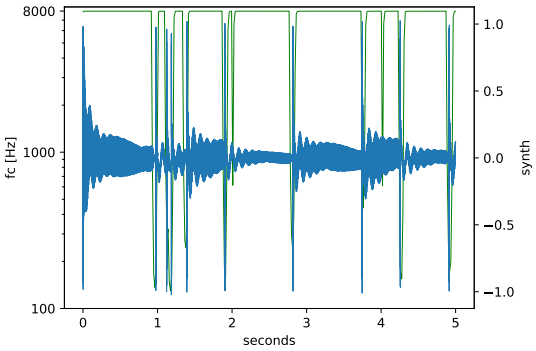
String 2



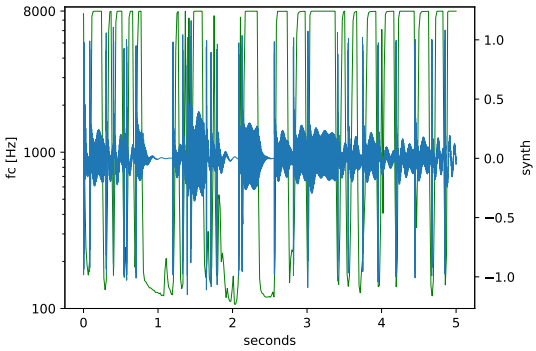
String 3



String 4

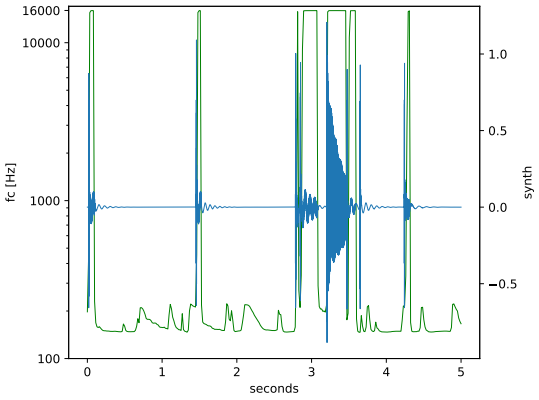


String 5

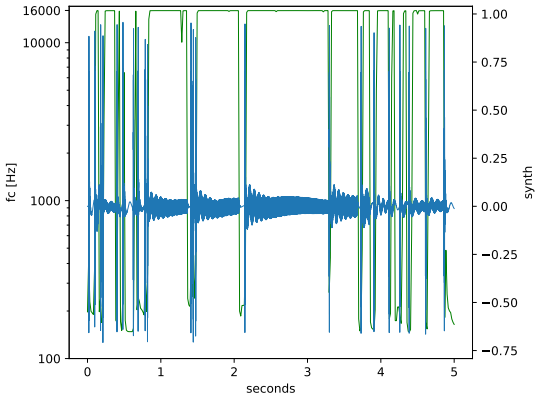


String 6

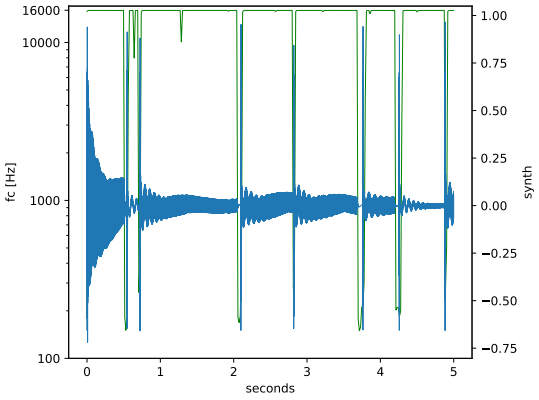
Figure 6.18 – ExA1 f_c at a logarithmic scale in green with the synthesized source estimate in blue.



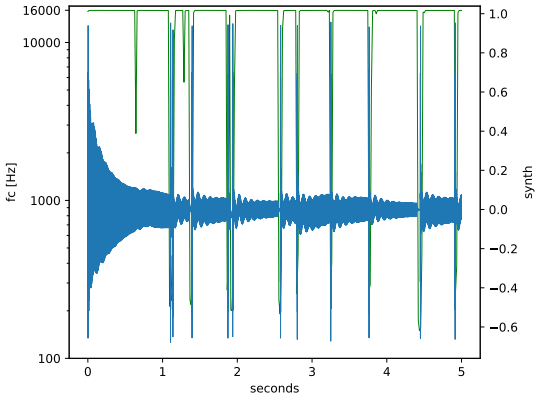
String 1



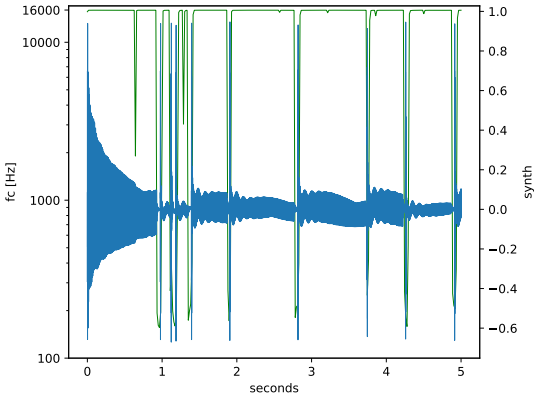
String 2



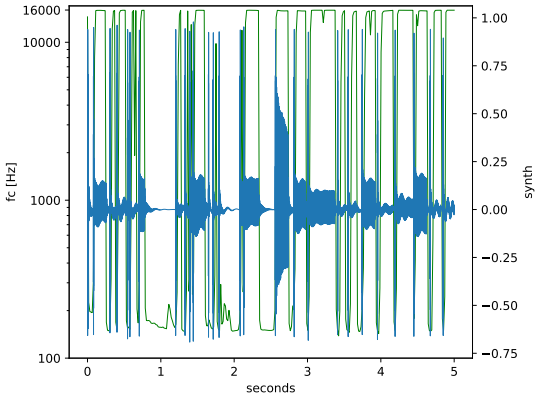
String 3



String 4



String 5



String 6

Figure 6.19 – ExA2 f_c at a logarithmic scale in green with the synthesized source estimate in blue.

The MCD suggests that synthesized and masked source estimates have better performance than the lower baseline (noise) and lower performance than the upper baseline (IRM). Again $\tilde{s}_i[n]$ achieve higher MCD than $\hat{s}_i[n]$ and show similar results for ExA1 and ExA2.

Looking at the source signal comparisons it is noticeable overall that the onset detection produces a lot of false onsets (especially in silent parts where the f_0 detection is difficult) and sometimes skips true onsets. This is clearly visible for string 1, 2 and 6 which all have low volume across this example. For string 3, 4 and 5 the onset detection works reasonably well.

The synthesized sources show that ExA2 is capable of producing notes with longer sustain than ExA1 due to the extended f_c range up to 16kHz. The masked sources are similar to the target signals. However, especially at passages with low amplitude note signals are synthesized due to the false onsets. The results of the low quality onset detection are false note triggering and occasionally missed notes. This shows that the onset detection is not ideal and has a major impact on synthesis quality.

Looking at the control signals it is visible that due to the onset detection the synthesized source estimate gets excited every time the f_0 confidence passes the confidence threshold from a low to a high confidence, as intended. At low confidence parts, the f_0 track tends to deviate from the surrounding high confidence parts.

The neurally predicted f_c controlling the lowpass in the feedback path of the Karplus-Strong string model switches mostly between the minimum (0Hz) and maximum (ExA1: 8kHz, ExA2: 16kHz) value. Overall f_c is low at the excitation times and high at the steady-state oscillation part of the notes. It looks like the DNN learned to soften the excitation by reducing f_c and sustaining the string oscillation by increasing f_c. By looking at f_c on a logarithmic scale, it is visible that f_c never falls below 100Hz for all strings. f_c fluctuates around 150Hz at parts where it is low for strings 1 and 6 whose target sources have low volume and low f_0 confidence. Otherwise f_c is clipped at the maximum f_c and forms short notches at the excitations.

Overall it can be observed that this simple DDSP Karplus-Strong string model can be successfully employed inside a DNN for MSS. Furthermore the DNN is able to learn from the data to predict a reasonable control parameter for synthesizing source signals from the mixture signal. This is a proof of concept for employing a DDSP physical string model inside a NN for MSS.

6.4 Experiment B Series

Table 6.1 shows the mean epoch losses for all experiments in the experiment B series, with ExB2 and ExB2.1 having the highest losses. Figure 6.20 shows the batch losses for all experiments with ExB2 and ExB2.1 having high spikes at the beginning batches. Figure 6.21 shows a SI-SDR box plot and Figure 6.22 shows a MCD box plot for all experiments of the B series.

Table 6.1 – Mean epoch training and validation losses for all models in the experiment B series, trained for two epochs.

Model	train. loss ep. 1	val. loss ep. 1	train. loss ep. 2	val. loss ep. 2
ExB1	16.1	16.3	15.1	15.2
ExB2	38.2	37.4	36.6	37.0
ExB2.1	30.4	31.7	30.6	31.1
ExB3	16.3	15.6	14.7	15.7
ExB4	15.7	15.5	14.5	15.1

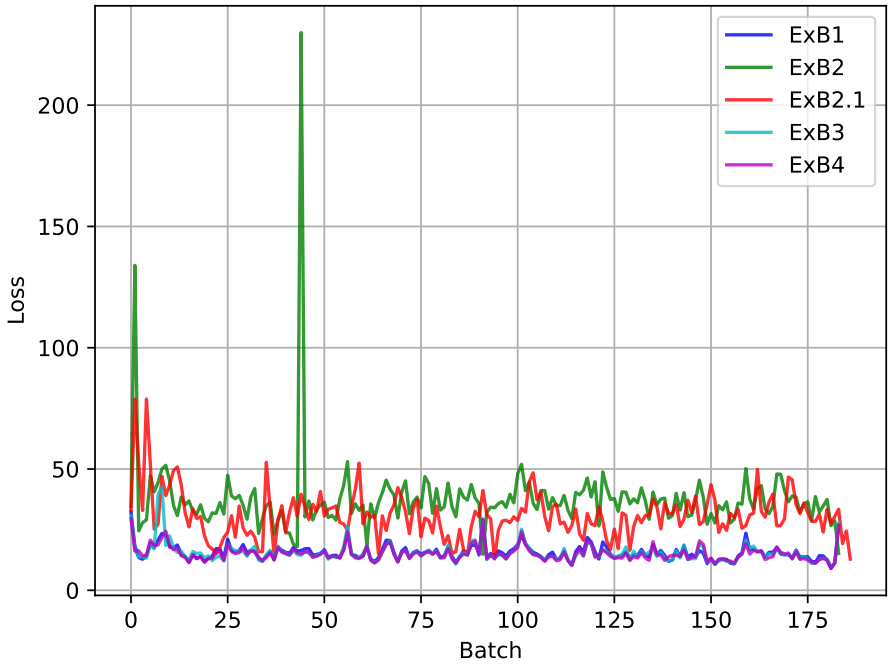


Figure 6.20 – Training batch losses for experiments B 1 to 4.

Figure 6.23 shows a section of the directly synthesized sources $\hat{s}_i[n]$ in comparison to the target sources $s_i[n]$ for all experiments of the B series. ExB0 shows many wrong onsets and high amplitude since it is not neurally controlled but has exclusively fixed parameters. All other models with neural controls produce synthesized sources with very small amplitude compared to the target sources.

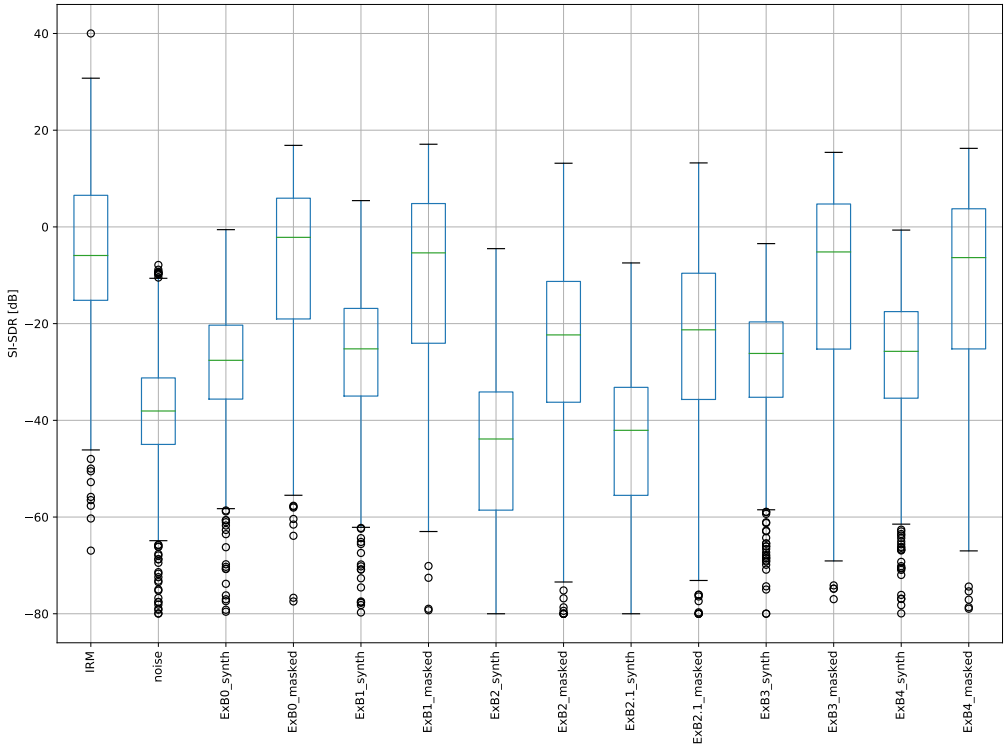


Figure 6.21 – SI-SDR box plot for the experiment B series.

The neurally predicted control signals are depicted in the appendix from Figure 8.1 to 8.10. The amplitudes of these control signals are overall very small. Furthermore it is visible that controls for different strings are very similar. Correlation coefficient matrices for the predicted control signals excluding the ones from ExB2 and ExB2.1 are depicted in Figure 6.24.

The *correlation coefficient matrix* (or Pearson product-moment correlation coefficients) R is calculated as described in [Num23] via

$$R_{ij} = \frac{C_{ij}}{\sqrt{C_{ii}C_{jj}}} \tag{6.1}$$

with covariance $C_{ij} = \mathrm{Cov}(c_i[m], c_j[m])$ and variance $C_{ii} = \mathrm{Var}(c_i[m])$, given control signals $c_i[m]$ and $c_j[m]$ with instrument indices i and j. $R_{ij} \in [-1, 1]$ quantifies the linear dependence of $c_i[m]$ and $c_j[m]$ modeled as realizations of random vectors. A high positive correlation with $R_{ij} \approx 1$ expresses a high linear statistical relashionship between the two random vectors and hence quantifies the similarity of the control signals $c_i[m]$ and $c_j[m]$ for different strings i and j.

Neurally predicted control signals for different strings are highly correlated with correlation coefficients above 0.9. This indicates that the DNN is not able to learn to control the source models of the individual strings effectively, but rather controls them in a very similar way.

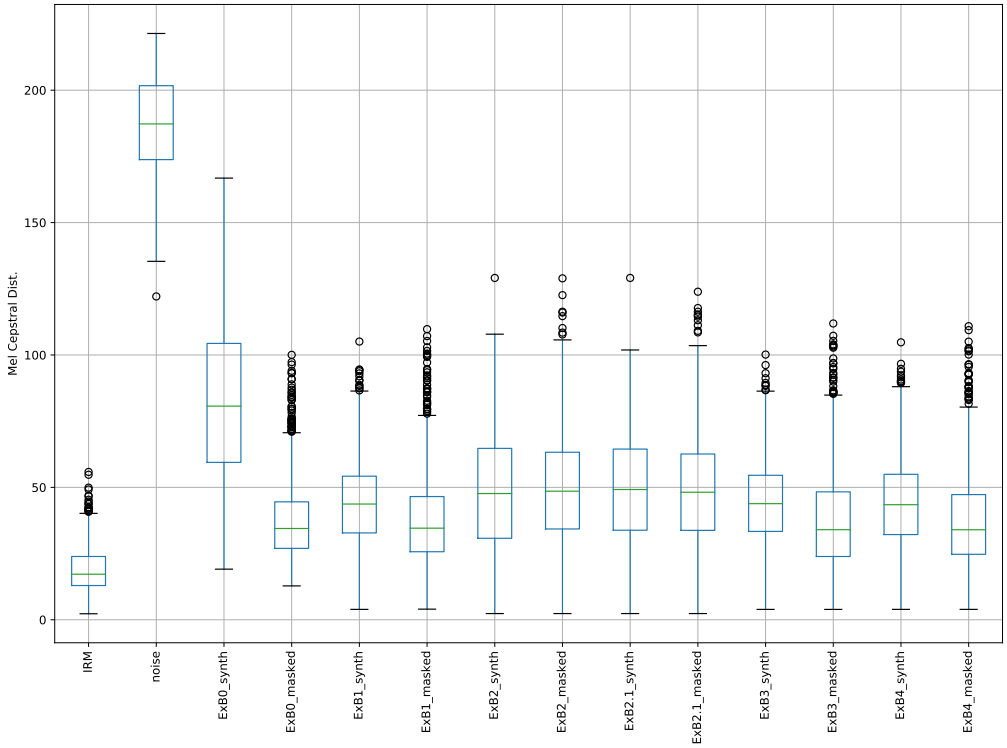


Figure 6.22 – Mel Cepstral Distance box plot for the experiment B series.

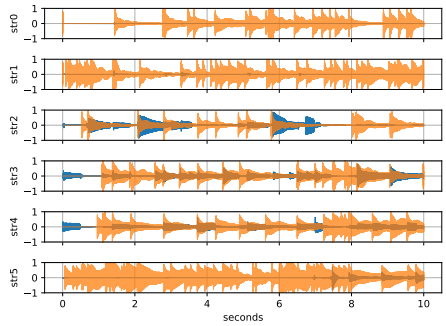
The learning curves of the models in the experiment B series show that the models containing a neurally controlled feedback factor ρ (ExB2 and ExB2.1) have stability problems at training and reach the worst performance. An explanation for this is that the time invariant predictions for ρ introduce energy into the feedback loop of the Karplus-Strong model leading to instability. Overall, neurally controlled models in the experiment B series produced synthesized sources with very low amplitude.

The SI-SDR and MCD metrics do not show an improvement of neurally controlled models over the uncontrolled model ExB0. However, the learning curves suggest that ExB4 is a candidate for further improvement, since it produces the lowest losses.

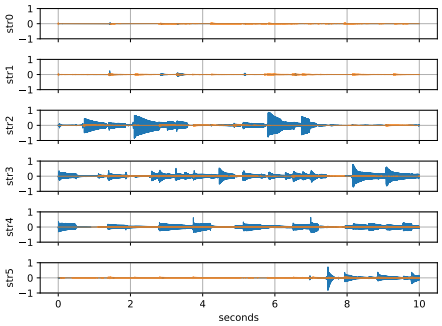
6.5 Experiment C Series

ExC1 training resulted in a learning curve with mean training losses of 16.1 (epoch 1) and 15.0 (epoch 2), with mean validation losses of 15.4 (epoch 1) and 15.9 (epoch 2). The ExC2 training produced an extremely high training loss at batch index 1 which raised the mean training loss of epoch 1 significantly. Hence, it was removed from mean loss calculation. ExC2 batch losses are depicted in Figure 6.25. The corrected mean training losses for ExC2 stayed at 82.0 for both epoch 1 and 2 and the mean validation loss stayed at 84.3 for both epochs.

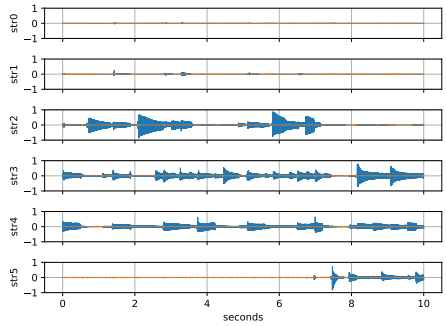
The training and validation losses of ExC1 are comparable to ExB1 and ExB3. Training



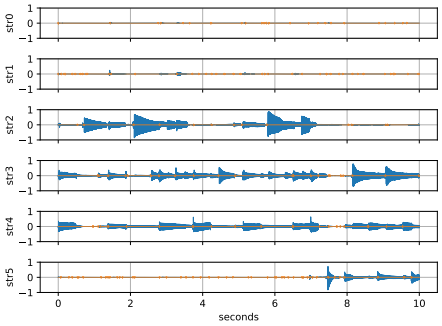
ExB0



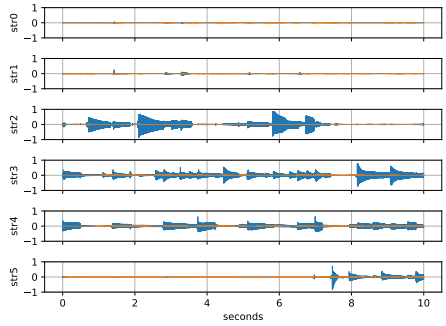
ExB1



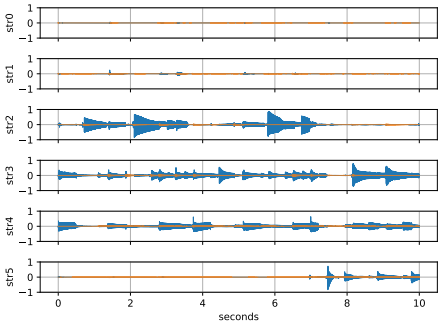
ExB2



ExB2.1

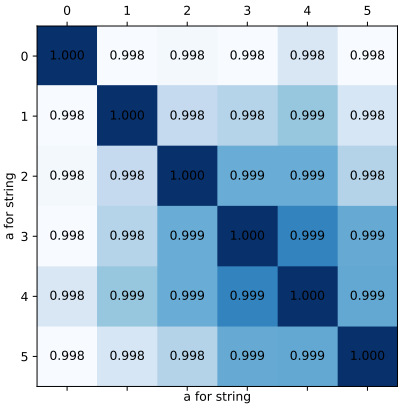


ExB3

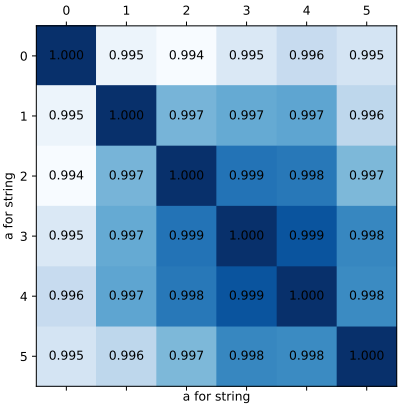


ExB4

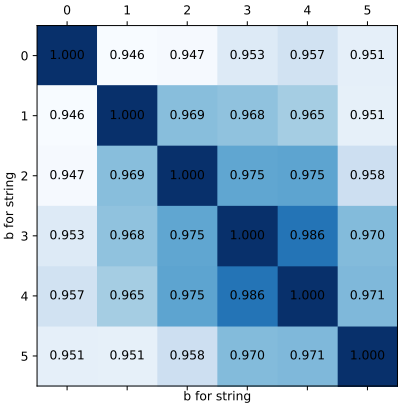
Figure 6.23 – Synthesized sources $\hat{s}_i[n]$ in orange with target sources $s_i[n]$ in blue for the experiment B series.



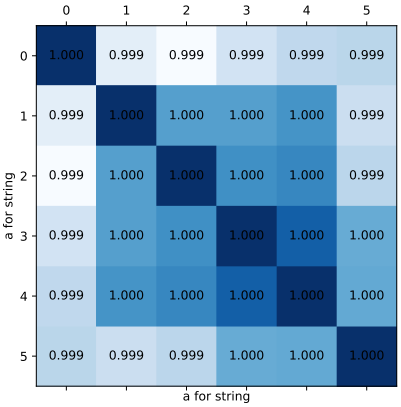
ExB1 a



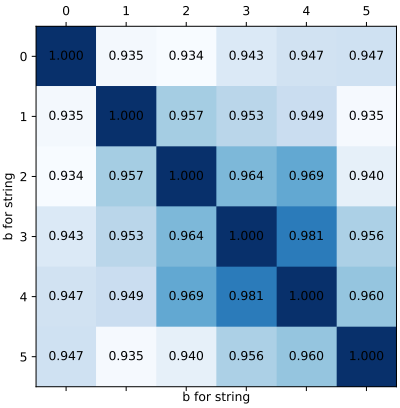
ExB3 a



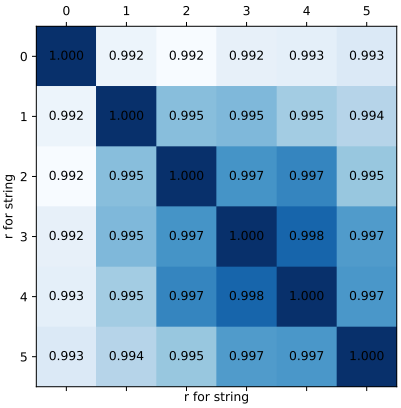
ExB3 b



ExB4 a



ExB4 b



ExB4 r

Figure 6.24 – Correlation coefficient matrices for neural control signals of stable models of experiment B series.

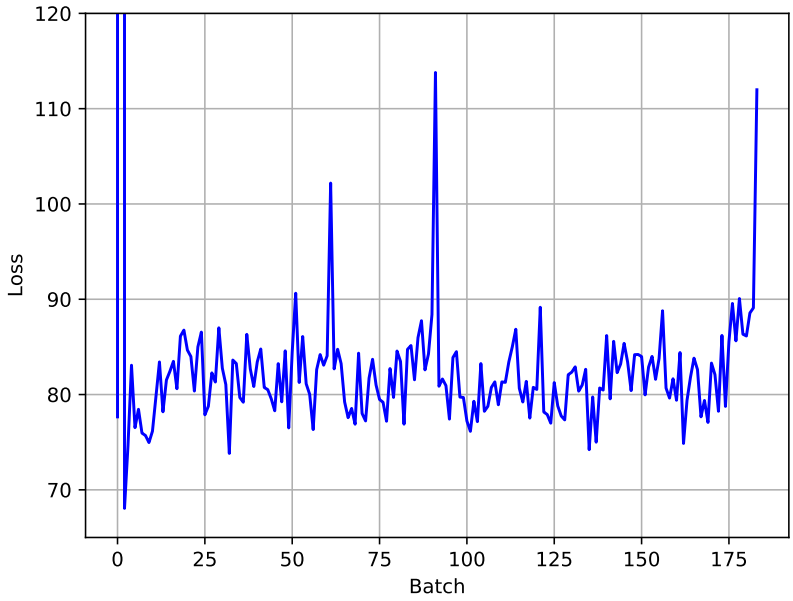


Figure 6.25 – ExC2 batch losses. The batch loss at index 1 has a value of 82340395089920.

of ExC2 was unstable with different seeds and had to be aborted repeatedly. This was recognizable through a training loss of *NaN* (not a number) which occurs at invalid operations like underflows or overflows. However it was possible to finish training with the depicted high training loss at batch index 1.

Figures 6.26 and 6.27 show the directly synthesized sources $\hat{s}_i[n]$ in comparison to the target sources $s_i[n]$ for ExC1 and ExC2 respectively. ExC1 produces sources with very small amplitude whereas ExC2 is constantly excited producing maximum signal level, constrained by the nonlinearity in the feedback path of the physical model. Correlation coefficient matrices for the predicted control signals are depicted in Figure 6.30. Again, the predicted controls are highly correlated with all correlation coefficients above 0.8 and most above 0.9. The neurally predicted control signals for ExC1 and ExC2 are depicted in the appendix from Figure 8.11 to 8.16. Box plots for the SI-SDR and the MCD are depicted in Figure 6.28 and 6.29.

The unsuccessful training passes of ExC2 were most likely produced due to an overflow since the training loss at batch index 1 has a unusually high value. Its learning curve stays on constant loss values indicating that the DNN is not able to learn from the data with this source model. Mean training and validation losses are about a factor 5 higher than ExC1.

Since ExC1 failed to increase the synthesized source amplitudes and ExC2 showed unstable training behavior with source estimates profoundly deviating from the target sources, the experiment C series did not improve on the experiment B series.

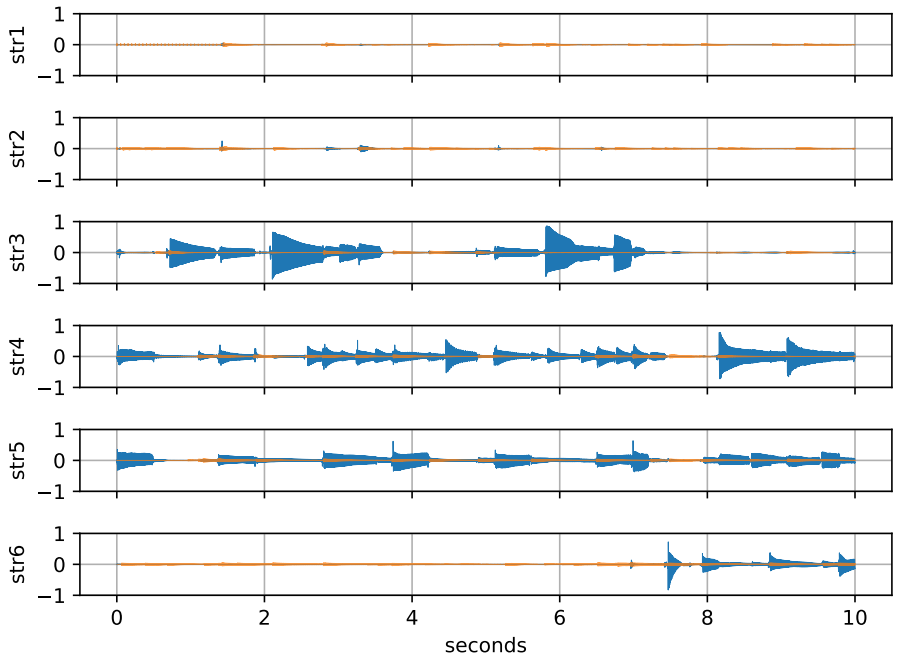


Figure 6.26 – Synthesized sources $\hat{s}_i[n]$ in orange with target sources $s_i[n]$ in blue for ExC1.

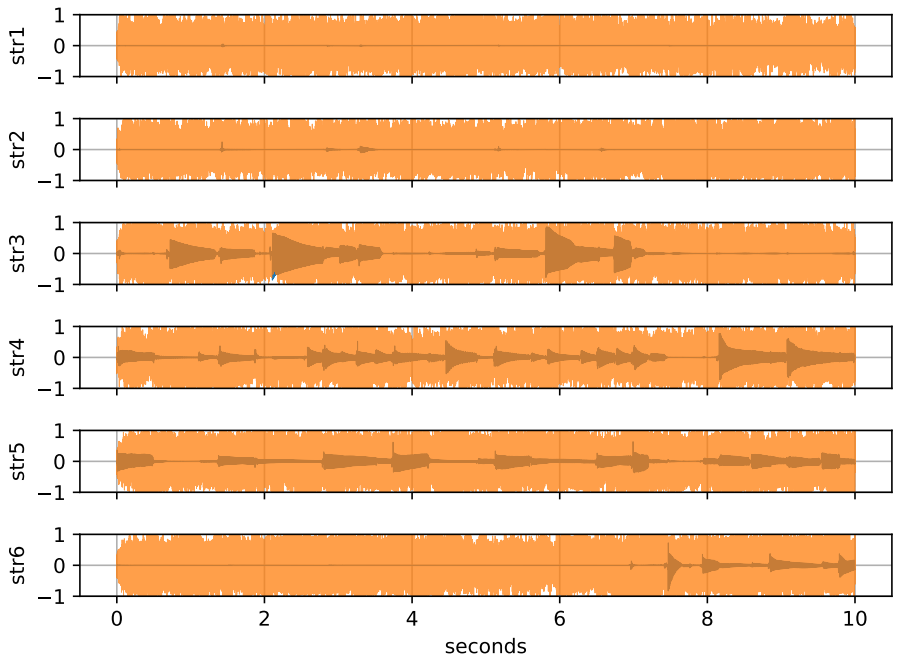


Figure 6.27 – Synthesized sources $\hat{s}_i[n]$ in orange with target sources $s_i[n]$ in blue for ExC2.

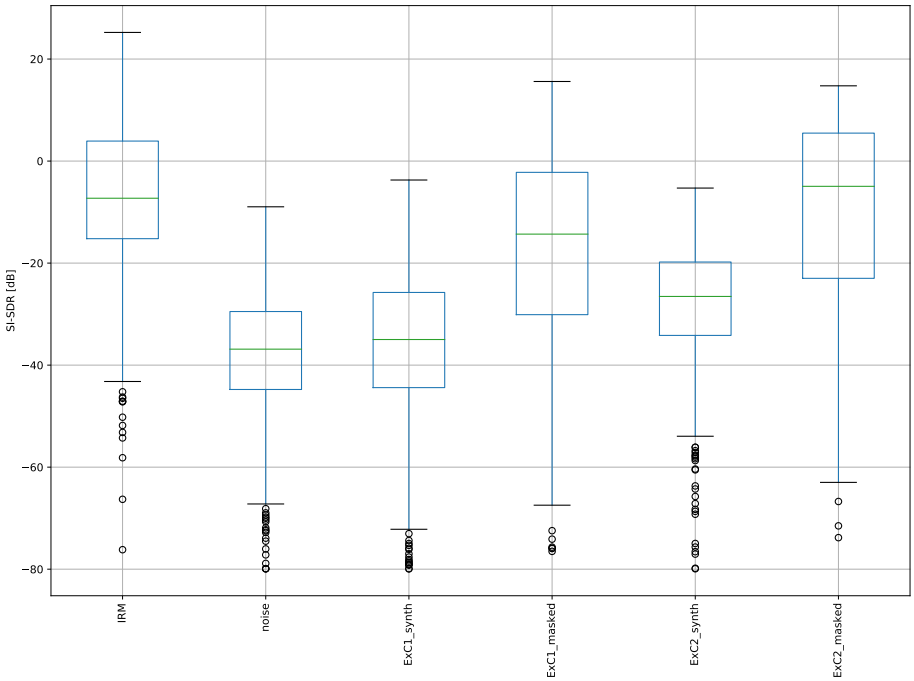


Figure 6.28 – SI-SDR box plot for experiment C series.

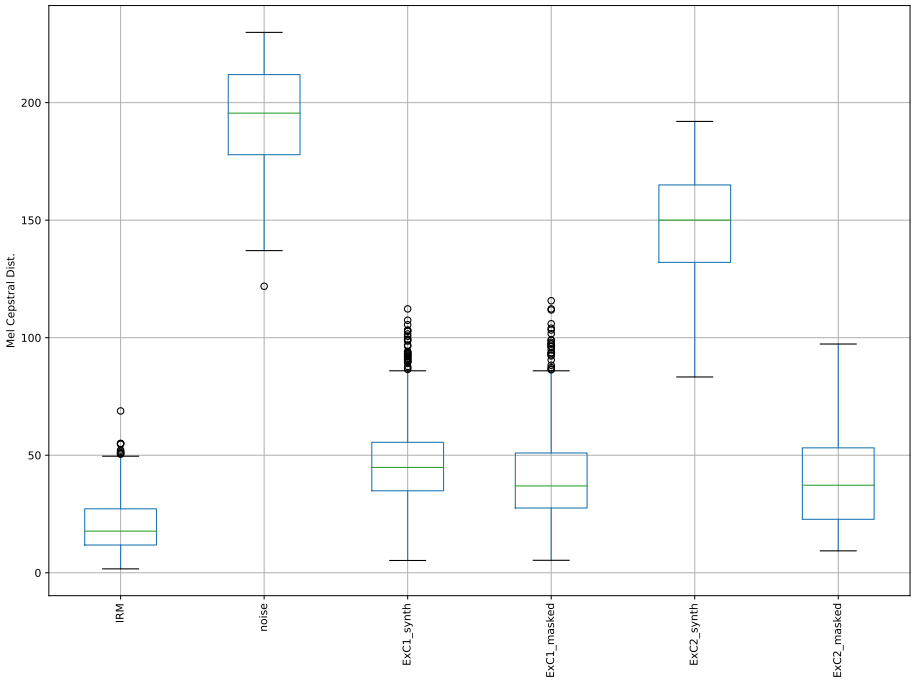
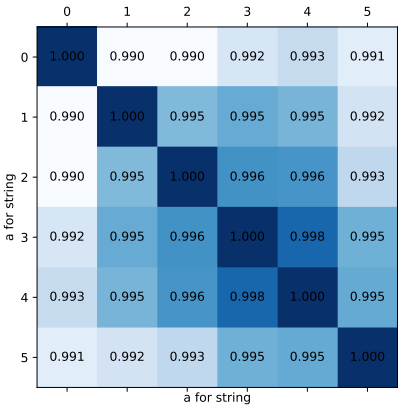
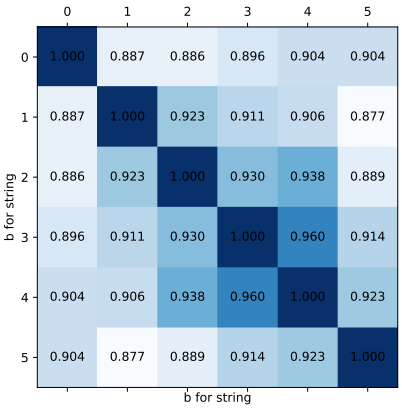


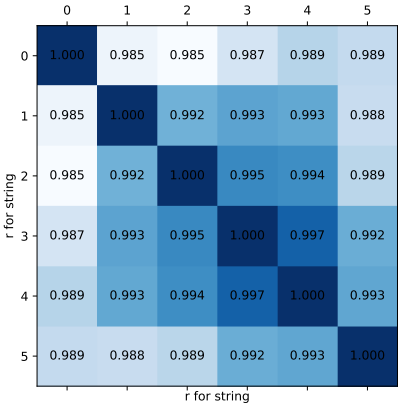
Figure 6.29 – MCD box plot for experiment C series.



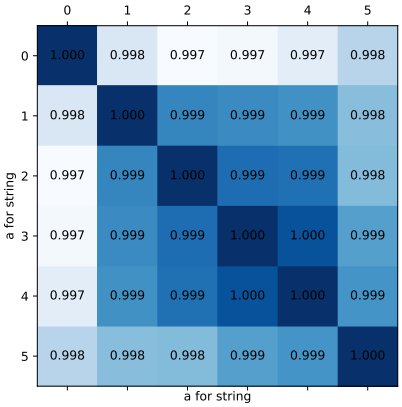
ExC1 a



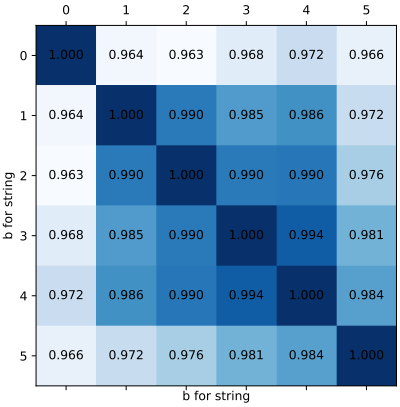
ExC1 b



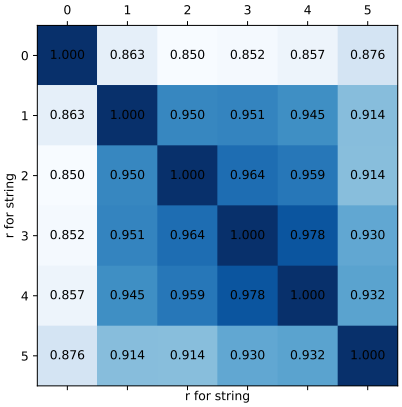
ExC1 r



ExC2 a



ExC2 b



ExC2 r

Figure 6.30 – Correlation coefficient matrices for neural control signals of experiment C series.

Chapter 7

Conclusion and Outlook

In this work instrument-specific music source separation was examined using interpretable and physics-inspired artificial intelligence. Unlike many state-of-the-art music source separation methods which separate different instruments, the proposed method achieved separation of guitar string signals. This was done by encoding knowledge about the instrument with a physical string model into the neural network, extending a method for singing voice separation.

A differentiable Karplus-Strong physical string model has successfully been integrated in a neural network architecture. It has been shown that it is in fact differentiable and that gradients can flow through the physical model. This enabled the neural network to effectively learn from data, predicting interpretable synthesis controls for the physical model to synthesize source signals from the mixture signal.

A preliminary stage of unsupervised guitar string separation has been achieved by informing the method with fundamental frequencies analyzed from the target signals. It was able to successfully separate guitar string signals from a mixture signal using a single neurally predicted control by synthesizing source signals with the physical model. The results were refined by masking the mixture signal with masks obtained from the synthesized sources.

The proposed method serves as a proof of concept for introducing differentiable physical modeling synthesis in neural music source separation. To the knowledge of the author this is the first time differentiable physical modeling synthesis was used in a neural network for music source separation.

Achieving high quality source separation with systems based on this work would enable a multitude of applications. In music production it enables to make numerous changes to monophonic sources in a polyphonic mixture, extending existing workflows. This enables improvements and new applications in postproduction, upmixing, remixing, new recording strategies, audio restoration and enhancement. Furthermore this concept can be generalized and applied in different fields. Examples are medical diagnostics and imaging procedures, image processing in the context of physical objects (mechanics, astronomy, etc.), navigation, weather forecasts, and seismic predictions.

Although the separation of guitar string signals was successful, it was not possible to improve the separation quality by using extended versions of the physical model. Experiments with extended models produced source estimates with very low amplitude or had stability issues. Therefor, the research question could not be completely answered. It has been shown that a differentiable physical string model can successfully be used for music source separation. However, how to achieve better separation quality by extending the physical model remains an open question.

Looking at the results from the conducted experiments, it poses the question why experiment series B and C did not improve music source separation quality over experiment series A, by extending the simple Karplus-Strong string model. There are multiple reasons why this is the case.

Firstly the crude note onset detection with the simple algorithm which uses f_0 and f_0 confidence is not an ideal solution. Since the f_0 data is analyzed frame-wise the time resolution is limited and produced deviations from the true note onsets of the target source signals. Having inaccurate onset timing and also wrong onsets leads to bad learning behavior. This might be the reason why the optimization of the DNN leads to trained models that produce very low amplitude synthesized source estimates.

Secondly the models in experiment series B and C have been trained for only two epochs to assess the performance tendency of different implementations in short periods of time. This was done due to timely constraints in the development of this work. Two epochs is the minimal amount of training for such an assessment and models trained for more epochs may perform better. Furthermore the amount of training data was also reduced to a minimum following the work of [EHGR20] and [SFDRB22] which suggest very high data efficiency of these systems.

Thirdly neurally predicted control signals have not been smoothed with the exception of a in ExC2. Time variant dynamic systems with feedback such as the employed Karplus-Strong physical string model are inherently prone to instability. Although the excitation of the system with the intended excitation signal may not reach the stability limit, instability may occur due to control signal changes. These instability inducing control signal changes can be caused by the form of the neural prediction or even due to the frame-wise predictions.

Fourthly the employed pitch tracking with the CREPE pitch tracker [KSLB18] may not have sufficient quality for this system. By using inaccurate f_0 data for synthesis, there is no way the DNN may improve MSS quality since the loss function penalizes detuning. Furthermore the onset detection was based on this f_0 data.

Fifthly the DNN architecture adopted from [SFDRB22] may not be able to control the extended physical string models. Although the DNN was able to achieve good results in [SFDRB22] it may not be ideal for this work. A fundamental difference of the systems is that in this work source models with a larger memory, representing the internal state, are used. The delay line in the Karplus-Strong model uses more memory then the 20th order IIR filter in [SFDRB22]. Therefor, in this work neurally predicted control signal changes cause effects that reach longer into the future than in [SFDRB22]. This poses higher requirements on the RNNs (GRUs) employed in the NN architecture.

There are multiple ways for future research based on this work. Note onset detection may be improved by employing better algorithms or jointly learning this task with a neural network. Smoothing neurally predicted control signals may improve separation quality. Since the source model does not pose any limitation on the neural network architecture, different architectures can be employed and may achieve better results. It is also desirable to achieve true unsupervised learning with the use of a polyphonic pitch tracker or jointly learning pitch tracking together with the music source separation task.

Differentiable digital signal processing proved to be a fruitful concept for future research. This work extended this concept with the use of physical modeling synthesis inside neural networks with the application of guitar string separation.

Bibliography

[Bis06] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer Science+Business Media, LLC, 2006.

[BMR+20] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[Bur19] A. Burkov, *Machine Learning kompakt: Alles, was Sie wissen müssen*. Frechen: mitp, 2019.

[CFCS17] K. Choi, G. Fazekas, K. Cho, and M. Sandler, "A tutorial on deep learning for music information retrieval," *arXiv preprint arXiv:1709.04396*, 2017.

[CGGMDL18] H. Cuesta, E. Gómez Gutiérrez, A. Martorell Domínguez, and F. Loáiciga, "Analysis of intonation in unison choir singing," *15th International Conference on Music Perception and Cognition*, 2018. [Online]. Available: http://hdl.handle.net/10230/35953

[CHL+22] B.-Y. Chen, W.-H. Hsu, W.-H. Liao, M. A. M. Ramírez, Y. Mitsufuji, and Y.-H. Yang, "Automatic dj transitions with differentiable audio effects and generative adversarial networks," in *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2022, pp. 466–470.

[CKC+20] W. Choi, M. Kim, J. Chung, D. Lee, and S. Jung, "Investigating u-nets with various intermediate blocks for spectrogram-based singing voice separation," 2020.

[CKCJ21] W. Choi, M. Kim, J. Chung, and S. Jung, "Lasaft: Latent source attentive frequency transformation for conditioned source separation," in *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2021, pp. 171–175.

[CMS22] F. Caspe, A. McPherson, and M. Sandler, "Ddx7: Differentiable fm synthesis of musical instrument sounds," *arXiv preprint arXiv:2208.06169*, 2022.

[CR21] J. T. Colonel and J. Reiss, "Reverse engineering of a recording mix with differentiable digital signal processing," *The Journal of the Acoustical Society of America*, vol. 150, no. 1, pp. 608–619, 2021.

[dee23a] "Deezer source separation library including pretrained models." deezer, 2023, accessed at 2023-01-17. [Online]. Available: https://github.com/deezer/spleeter

[dee23b] "Deezer streaming platform," deezer, 2023, accessed at 2023-01-17. [Online]. Available: https://www.deezer.com/

[Déf21] A. Défossez, "Hybrid spectrogram and waveform source separation," *arXiv preprint arXiv:2111.03600*, 2021.

[DFAG17] Y. N. Dauphin, A. Fan, M. Auli, and D. Grangier, "Language modeling with gated convolutional networks," in *International conference on machine learning*. PMLR, 2017, pp. 933–941.

[DM19] B. De Man, "Intelligent music production," 2019.

[DUBB19] A. Défossez, N. Usunier, L. Bottou, and F. R. Bach, "Music source separation in the waveform domain," *CoRR*, vol. abs/1911.13254, 2019. [Online]. Available: http://arxiv.org/abs/1911.13254

[DV16] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv preprint arXiv:1603.07285*, 2016.

[EHGR20] J. Engel, L. Hantrakul, C. Gu, and A. Roberts, "Ddsp: Differentiable digital signal processing," 2020.

[ESH+20] J. Engel, R. Swavely, L. H. Hantrakul, A. Roberts, and C. Hawthorne, "Self-supervised pitch detection by inverse audio synthesis," in *ICML 2020 Workshop on Self-supervision in Audio and Speech*, 2020. [Online]. Available: https://openreview.net/forum?id=RlVTYWhsky7

[FGKC20] G. Fabbro, V. Golkov, T. Kemp, and D. Cremers, "Speech synthesis and control using differentiable dsp," *arXiv preprint arXiv:2010.15084*, 2020.

[GBY+18] L. H. Gilpin, D. Bau, B. Z. Yuan, A. Bajwa, M. Specter, and L. Kagal, "Explaining explanations: An overview of interpretability of machine learning," in *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA)*. IEEE, 2018, pp. 80–89.

[GCC22] Z. Guo, C. Chen, and E. S. Chng, "Dent-ddsp: Data-efficient noisy speech generator using differentiable digital signal processors for explicit distortion modelling and noise-robust speech recognition," *arXiv preprint arXiv:2208.00987*, 2022.

[Goo23a] "Differentiable digital signal processing - online supplement," https://storage.googleapis.com/ddsp/index.html, Google, 2023, accessed: 2023-02-07.

[Goo23b] "Differentiable digital signal processing - repository," https://github.com/magenta/ddsp, Google, 2023, accessed: 2023-02-07.

[Goo23c] "Google magenta project," https://magenta.tensorflow.org/, Google, 2023, accessed: 2023-02-07.

[Goo23d] "Google research: Brain team," https://research.google/teams/brain/, Google, 2023, accessed: 2023-02-07.

[Gé19] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. Sebastopol: O'Reilly Media, Incorporated, 2019.

[HERG19] L. Hantrakul, J. H. Engel, A. Roberts, and C. Gu, "Fast and flexible neural audio synthesis." in *ISMIR*, 2019, pp. 524–530.

[HG16] D. Hendrycks and K. Gimpel, "Gaussian error linear units (gelus)," *arXiv preprint arXiv:1606.08415*, 2016.

[HJA20] J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models," *Advances in Neural Information Processing Systems*, vol. 33, pp. 6840–6851, 2020.

[HKVM20] R. Hennequin, A. Khlif, F. Voituret, and M. Moussallam, "Spleeter: a fast and efficient music source separation tool with pre-trained models," *Journal of Open Source Software*, vol. 5, no. 50, p. 2154, 2020. [Online]. Available: https://doi.org/10.21105/joss.02154

[HSF21] B. Hayes, C. Saitis, and G. Fazekas, "Neural waveshaping synthesis," *arXiv preprint arXiv:2107.05050*, 2021.

[JS83] D. A. Jaffe and J. O. Smith, "Extensions of the karplus-strong plucked-string algorithm," *Computer Music Journal*, vol. 7, no. 2, pp. 56–69, 1983.

[KB14] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014. [Online]. Available: https://arxiv.org/abs/1412.6980

[KCC+21] M. Kim, W. Choi, J. Chung, D. Lee, and S. Jung, "Kuielab-mdx-net: A two-stream neural network for music demixing," *arXiv preprint arXiv:2111.12203*, 2021.

[KLA+20] T. Karras, S. Laine, M. Aittala, J. Hellsten, J. Lehtinen, and T. Aila, "Analyzing and improving the image quality of stylegan," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 8110–8119.

[KNK+22] M. Kawamura, T. Nakamura, D. Kitamura, H. Saruwatari, Y. Takahashi, and K. Kondo, "Differentiable digital signal processing mixture model for synthesis parameter extraction from mixture of harmonic sounds," in *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2022, pp. 941–945.

[KPE20] B. Kuznetsov, J. D. Parker, and F. Esqueda, "Differentiable iir filters for machine learning applications," in *Proc. Int. Conf. Digital Audio Effects (eDAFx-20)*, 2020, pp. 297–303.

[KS83] K. Karplus and A. Strong, "Digital synthesis of plucked-string and drum timbres," *Computer Music Journal*, vol. 7, no. 2, pp. 43–55, 1983.

[KSLB18] J. W. Kim, J. Salamon, P. Li, and J. P. Bello, "Crepe: A convolutional representation for pitch estimation," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2018, pp. 161–165.

[Kub93] R. Kubichek, “Mel-cepstral distance measure for objective speech quality assessment,” in *Proceedings of IEEE pacific rim conference on communications computers and signal processing*, vol. 1. IEEE, 1993, pp. 125–128.

[KVT98] M. Karjalainen, V. Välimäki, and T. Tolonen, “Plucked-string models: From the karplus-strong algorithm to digital waveguides and beyond,” *Computer Music Journal*, vol. 22, no. 3, pp. 17–32, 1998.

[LCHL21] T. Li, J. Chen, H. Hou, and M. Li, “Sams-net: A sliced attention-based neural network for music source separation,” in *2021 12th International Symposium on Chinese Spoken Language Processing (ISCSLP)*, 2021, pp. 1–5.

[LCL22] S. Lee, H.-S. Choi, and K. Lee, “Differentiable artificial reverberation,” *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 30, pp. 2541–2556, 2022.

[LRWEH19] J. Le Roux, S. Wisdom, H. Erdogan, and J. R. Hershey, “Sdr–half-baked or well done?” in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019, pp. 626–630.

[LS19] A. Liutkus and F.-R. Stöter, “Eusipco 2019: Deep learning for music separation,” Colab notebook: https://colab.research.google.com/drive/1Zo6iSPIi6SjOAL7wg8yzVWkS9mjLgjI-, 2019, accessed at 2023-01-19. [Online]. Available: https://sigsep.github.io/tutorials/#eusipco-2019-deep-learning-for-music-separation

[LY22] Y. Luo and J. Yu, “Music source separation with band-split rnn,” *arXiv preprint arXiv:2209.15174*, 2022.

[Mas22] “Mit 6.s191: Recurrent neural networks and transformers,” Massachusetts Institute of Technology, 2022, accessed on 21.12.2022. [Online]. Available: https://www.youtube.com/watch?v=QvkQ1B3FBqA

[MBP19] G. Meseguer Brocal and G. Peeters, “Conditioned-u-net: Introducing a control mechanism in the u-net for multiple source separations,” in *Proceedings of the 20th International Society for Music Information Retrieval Conference*, Delft, Netherlands, Nov 2019. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02448917

[MDS21] S. I. Mimilakis, K. Drossos, and G. Schuller, “Unsupervised interpretable representation learning for singing voice separation,” in *2020 28th European Signal Processing Conference (EUSIPCO)*. IEEE, 2021, pp. 1412–1416.

[MFUS21] Y. Mitsufuji, G. Fabbro, S. Uhlich, and F.-R. Stöter, “Music demixing challenge 2021,” 2021.

[MKG+16] S. Mehri, K. Kumar, I. Gulrajani, R. Kumar, S. Jain, J. Sotelo, A. C. Courville, and Y. Bengio, “Samplernn: An unconditional end-to-end neural audio generation model,” *CoRR*, vol. abs/1612.07837, 2016. [Online]. Available: http://arxiv.org/abs/1612.07837

[MS19] D. Moffat and M. B. Sandler, “Approaches in intelligent music production,” in *Arts*, vol. 8, no. 4. MDPI, 2019, p. 125.

[MS21] N. Masuda and D. Saito, “Synthesizer sound matching with differentiable dsp.” in *ISMIR*, 2021, pp. 428–434.

[Nie15] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015, accessed: 2021-11-09. [Online]. Available: http://neuralnetworksanddeeplearning.com/index.html

[NSW21] S. Nercessian, A. Sarroff, and K. J. Werner, “Lightweight and interpretable neural modeling of an audio distortion effect using hyperconditioned differentiable biquads,” in *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2021, pp. 890–894.

[Num23] “Numpy documentation: corrcoef,” Numpy, 2023, accessed: 2023-02-23. [Online]. Available: https://numpy.org/doc/stable/reference/generated/numpy.corrcoef.html

[ODZ+16] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, “Wavenet: A generative model for raw audio,” *arXiv preprint arXiv:1609.03499*, 2016.

[Pap23] “Music source separation on musdb18,” Papers with Code, 2023, accessed at 2023-01-17. [Online]. Available: https://paperswithcode.com/sota/music-source-separation-on-musdb18

[PK20] F. Pernkopf and C. Knoll, “Computational intelligence skript 2020,” July 2020.

[PLV+19] H. Purwins, B. Li, T. Virtanen, J. Schlüter, S.-Y. Chang, and T. Sainath, “Deep learning for audio signal processing,” *IEEE Journal of Selected Topics in Signal Processing*, vol. 13, no. 2, pp. 206–219, 2019.

[PSdV+18] E. Perez, F. Strub, H. de Vries, V. Dumoulin, and A. Courville, “Film: Visual reasoning with a general conditioning layer,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, Apr. 2018. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/11671

[PyT22] “Pytorch documentation: Conv1d,” PyTorch Foundation, 2022, accessed on 20.12.2022. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Conv1d.html?highlight=conv1d#torch.nn.Conv1d

[PyT23] “Pytorch documentation: Hardtanh,” PyTorch, 2023, accessed: 2023-02-17. [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Hardtanh.html#torch.nn.Hardtanh

[RBL+22] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, “High-resolution image synthesis with latent diffusion models,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022. [Online]. Available: https://github.com/CompVis/latent-diffusionhttps://arxiv.org/abs/2112.10752

[RFB15] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, Eds. Cham: Springer International Publishing, 2015, pp. 234–241.

[RLS+17] Z. Rafii, A. Liutkus, F.-R. Stöter, S. I. Mimilakis, and R. Bittner, "The MUSDB18 corpus for music separation," Dec. 2017. [Online]. Available: https://doi.org/10.5281/zenodo.1117372

[RMD22] S. Rouard, F. Massa, and A. Défossez, "Hybrid transformers for music source separation," *arXiv preprint arXiv:2211.08553*, 2022.

[RMR22] L. Renault, R. Mignot, and A. Roebel, "Differentiable piano model for midi-to-audio performance synthesis," in *25th International Conference on Digital Audio Effects (DAFx20in22)*, 2022.

[RWSB21] M. A. M. Ramírez, O. Wang, P. Smaragdis, and N. J. Bryan, "Differentiable signal processing with black-box audio effects," in *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2021, pp. 66–70.

[SBR22] C. J. Steinmetz, N. J. Bryan, and J. D. Reiss, "Style transfer of audio effects with differentiable signal processing," *arXiv preprint arXiv:2207.08759*, 2022.

[SFDRB22] K. Schulze-Forster, C. S. Doire, G. Richard, and R. Badeau, "Unsupervised audio source separation using differentiable parametric source models," *arXiv preprint arXiv:2201.09592*, 2022.

[SHC+22] S. Shan, L. Hantrakul, J. Chen, M. Avent, and D. Trevelyan, "Differentiable wavetable synthesis," in *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2022, pp. 4598–4602.

[SHG21] O. Slizovskaia, G. Haro, and E. Gómez, "Conditioned source separation for musical instrument performances," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 29, pp. 2083–2095, 2021.

[SL23] F.-R. Stöter and A. Liutkus, "Open-unmix for pytorch (github repository)," 2023, accessed at 26.02.2023. [Online]. Available: https://github.com/sigsep/open-unmix-pytorch

[Smi13] J. O. Smith, *Physical Audio Signal Processing*. CCRMA Stanford, accessed 2023-02-13, online book, 2010 edition. [Online]. Available: \url{http://ccrma.stanford.edu/~jos/pasp/}

[Son23] "Sound demixing challenge 2023," https://www.aicrowd.com/challenges/sound-demixing-challenge-2023/problems/music-demixing-track-mdx-23, Sony, Moises, Mitsubishi, 2023, accessed: 2023-01-12.

[SP97] M. Schuster and K. Paliwal, "Bidirectional recurrent neural networks," *Signal Processing, IEEE Transactions on*, vol. 45, pp. 2673 – 2681, 12 1997.

[SPPS21] C. J. Steinmetz, J. Pons, S. Pascual, and J. Serrà, "Automatic multitrack mixing with a differentiable mixing console of neural audio effects," in *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2021, pp. 71–75.

[SU20] F.-R. Stöter and S. Uhlich, "Aes virtual symposium 2020: Current trends in audio source separation," https://sigsep.github.io/tutorials/#aes-virtual-symposium-2020-current-trends-in-audio-source-separation, INRIA, Sony, 2020, accessed: 2021-10-15.

[SULM19] F.-R. Stöter, S. Uhlich, A. Liutkus, and Y. Mitsufuji, "Open-unmix - a reference implementation for music source separation," *Journal of Open Source Software*, vol. 4, no. 41, p. 1667, Sep 2019. [Online]. Available: https://hal.inria.fr/hal-02293689

[SUTM21] R. Sawata, S. Uhlich, S. Takahashi, and Y. Mitsufuji, "All for one and one for all: Improving music separation by bridging networks," in *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2021, pp. 51–55.

[TDFH+22] R. Thoppilan, D. De Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H.-T. Cheng, A. Jin, T. Bos, L. Baker, Y. Du *et al.*, "Lamda: Language models for dialog applications," *arXiv preprint arXiv:2201.08239*, 2022.

[TM21] N. Takahashi and Y. Mitsufuji, "D3net: Densely connected multidilated densenet for music source separation," 2021.

[UPG+17] S. Uhlich, M. Porcu, F. Giron, M. Enenkl, T. Kemp, N. Takahashi, and Y. Mitsufuji, "Improving music source separation based on deep neural networks through data augmentation and network blending," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2017, pp. 261–265.

[VGF06] E. Vincent, R. Gribonval, and C. Févotte, "Performance measurement in blind audio source separation," *IEEE transactions on audio, speech, and language processing*, vol. 14, no. 4, pp. 1462–1469, 2006.

[VSP+17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[VVG18] E. Vincent, T. Virtanen, and S. Gannot, *Audio source separation and speech enhancement*, 1st ed. John Wiley & Sons, 2018.

[Wan22] P. Wang, "This person does not exist," 2022, accessed on 09.01.2023. [Online]. Available: https://thispersondoesnotexist.com/

[Wik23a] "Discrete-time high-pass filter," Wikipedia, 2023, accessed: 2023-02-16. [Online]. Available: https://en.wikipedia.org/wiki/High-pass_filter#Discrete-time_realization

[Wik23b] "Discrete-time low-pass filter," Wikipedia, 2023, accessed: 2023-02-13. [Online]. Available: https://en.wikipedia.org/wiki/Low-pass_filter#Discrete-time_realization

[WMD+21] Y. Wu, E. Manilow, Y. Deng, R. Swavely, K. Kastner, T. Cooijmans, A. Courville, C.-Z. A. Huang, and J. Engel, "Midi-ddsp: Detailed control of musical performance via hierarchical modeling," *arXiv preprint arXiv:2112.09312*, 2021.

[WVBW+22] J. J. Webber, C. Valentini-Botinhao, E. Williams, G. E. Henter, and S. King, "Autovocoder: Fast waveform generation from a learned speech representation using differentiable digital signal processing," *arXiv preprint arXiv:2211.06989*, 2022.

[XBP+18] Q. Xi, R. M. Bittner, J. Pauwels, X. Ye, and J. P. Bello, "Guitarset: A dataset for guitar transcription." in *ISMIR*, 2018, pp. 453–460.

[Zen19] "Choral singing dataset," Zenodo, 2019, accessed: 2023-02-12. [Online]. Available: https://zenodo.org/record/2649950#.Y-jjaRzMKi0

[ZXT19] Y. Zhao, X. Xia, and R. Togneri, "Applications of deep learning to audio generation," *IEEE Circuits and Systems Magazine*, vol. 19, no. 4, pp. 19–38, 2019.

Chapter 8

Appendix

In the appendix predicted control signals from experiment series B and C are depicted.

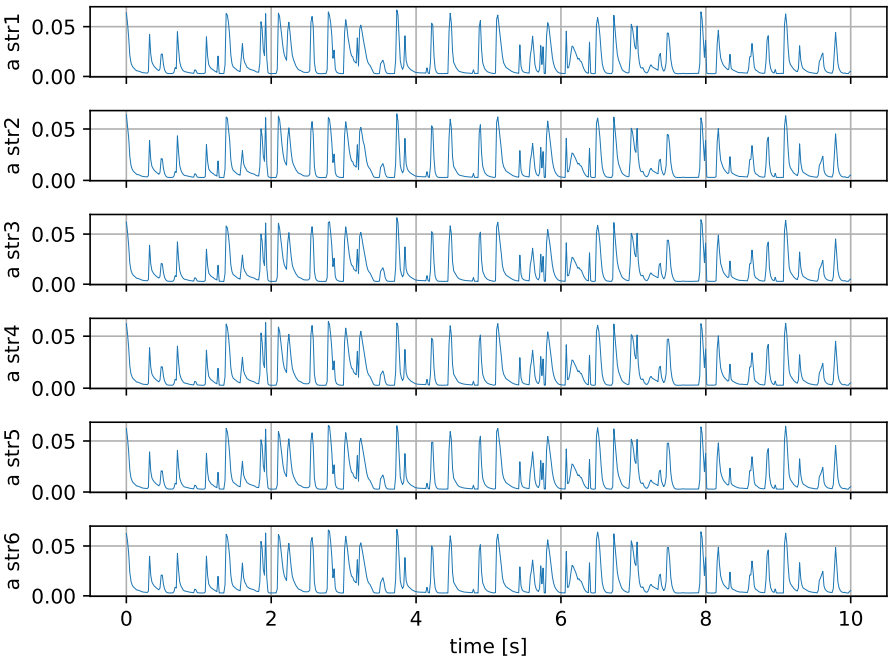


Figure 8.1 – ExB1: parameter a.

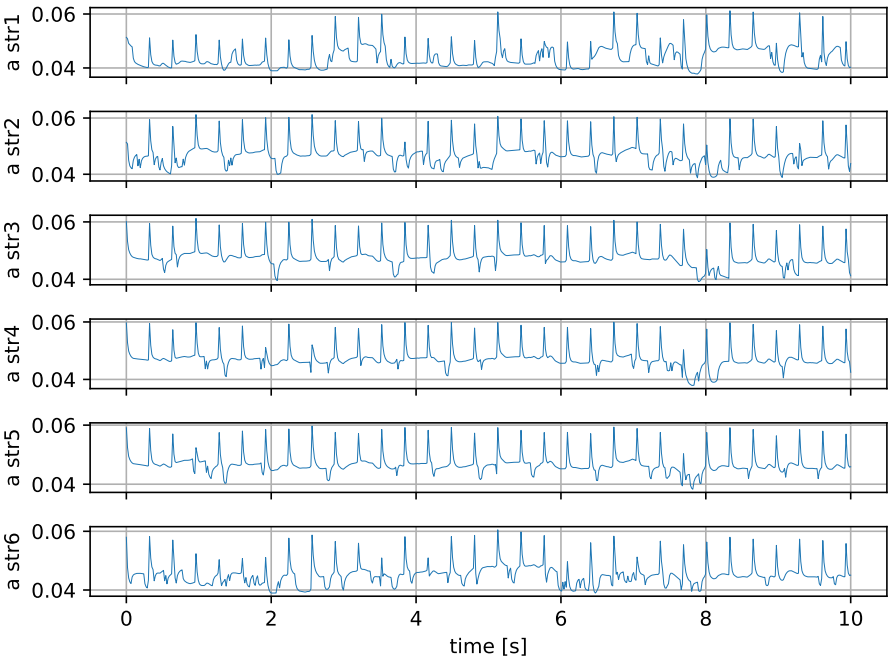


Figure 8.2 – ExB2: parameter a.

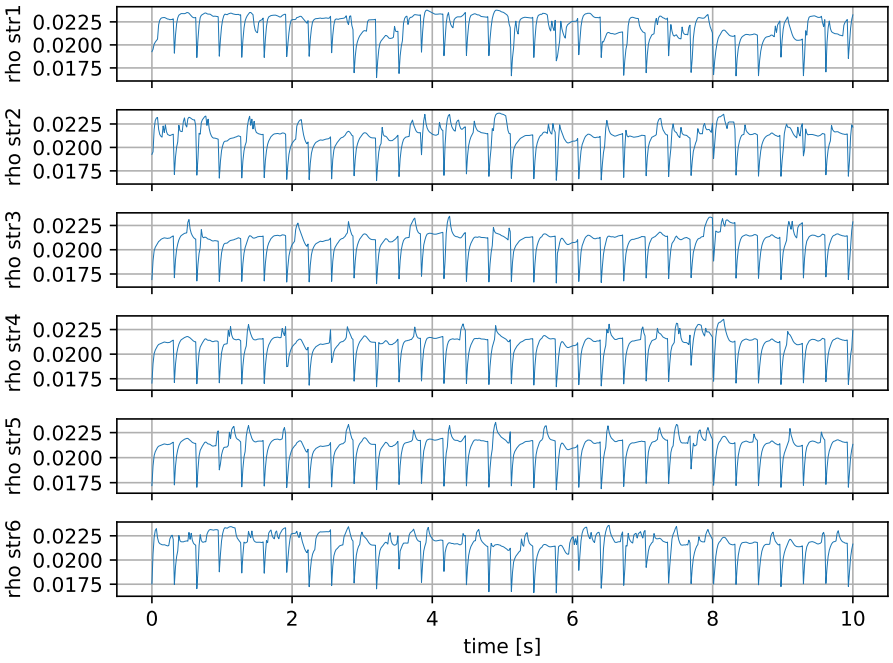


Figure 8.3 – ExB2: parameter ρ.

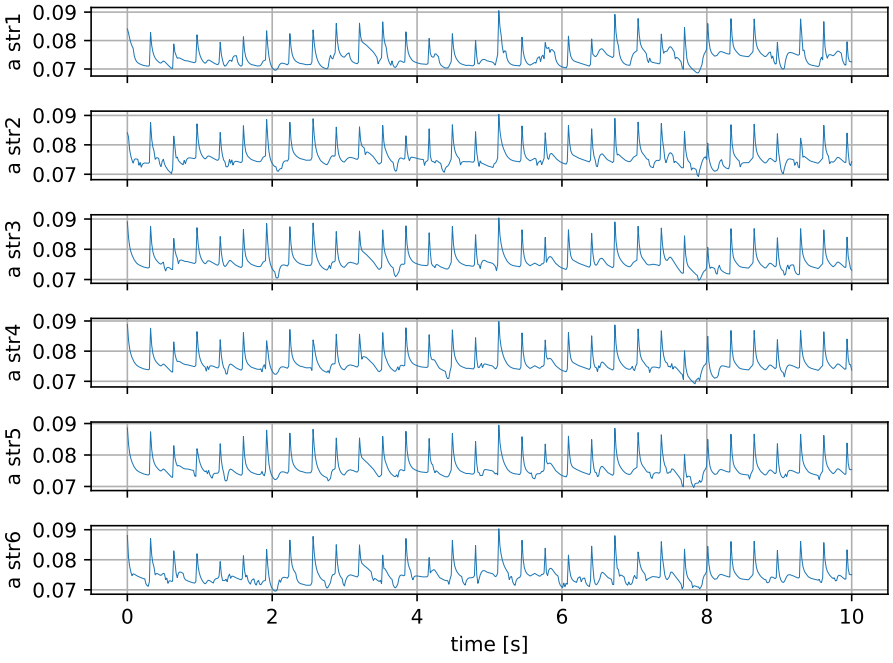


Figure 8.4 – ExB2.1: parameter a.

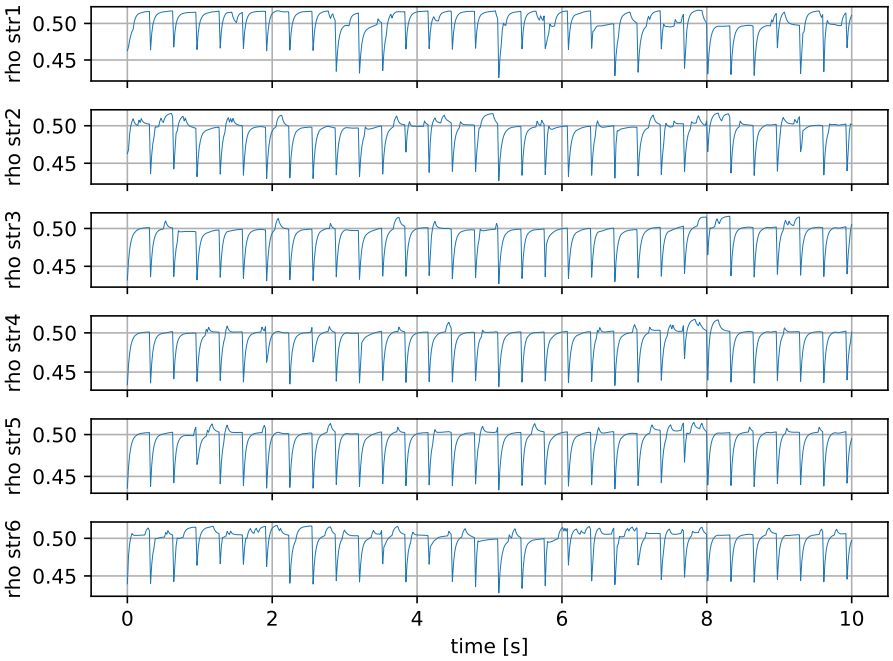


Figure 8.5 – ExB2.1: parameter ρ.

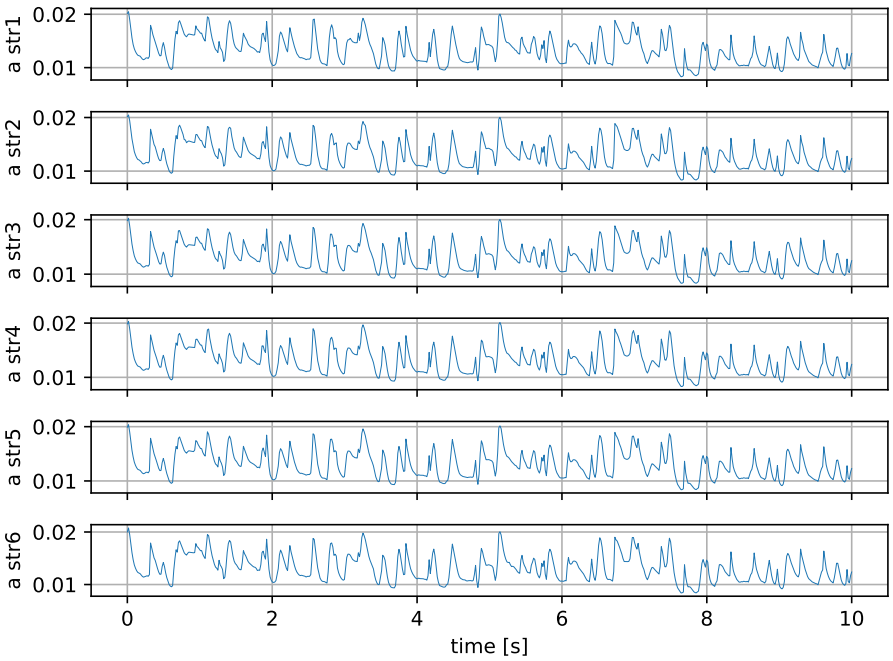


Figure 8.6 – ExB3: parameter a.

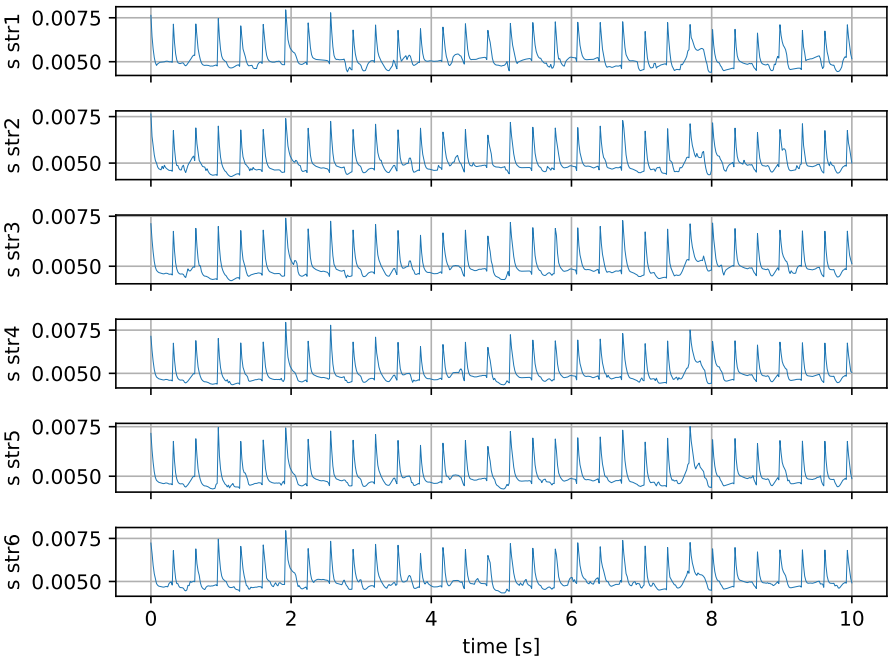


Figure 8.7 – ExB3: parameter b.

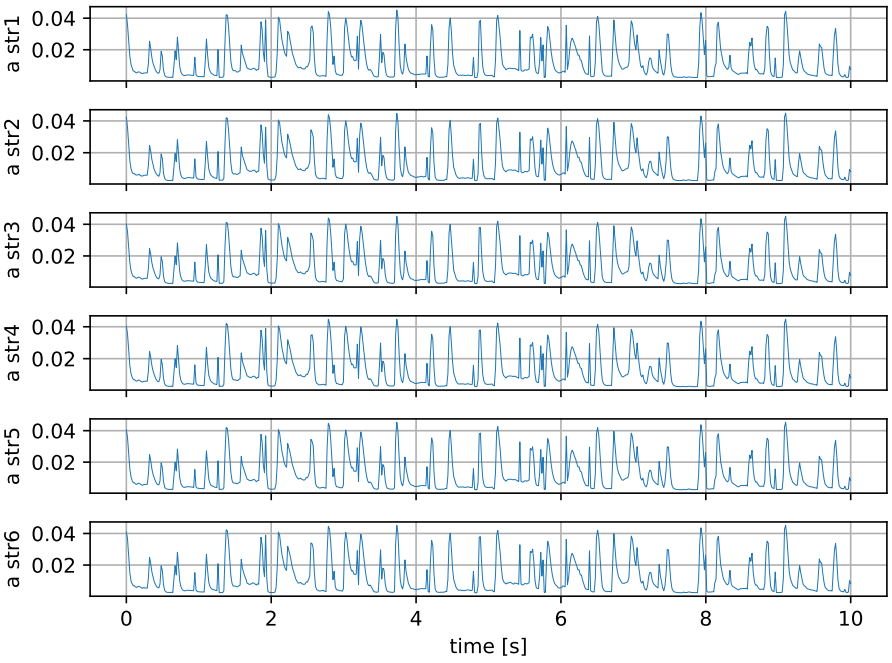


Figure 8.8 – ExB4: parameter a.

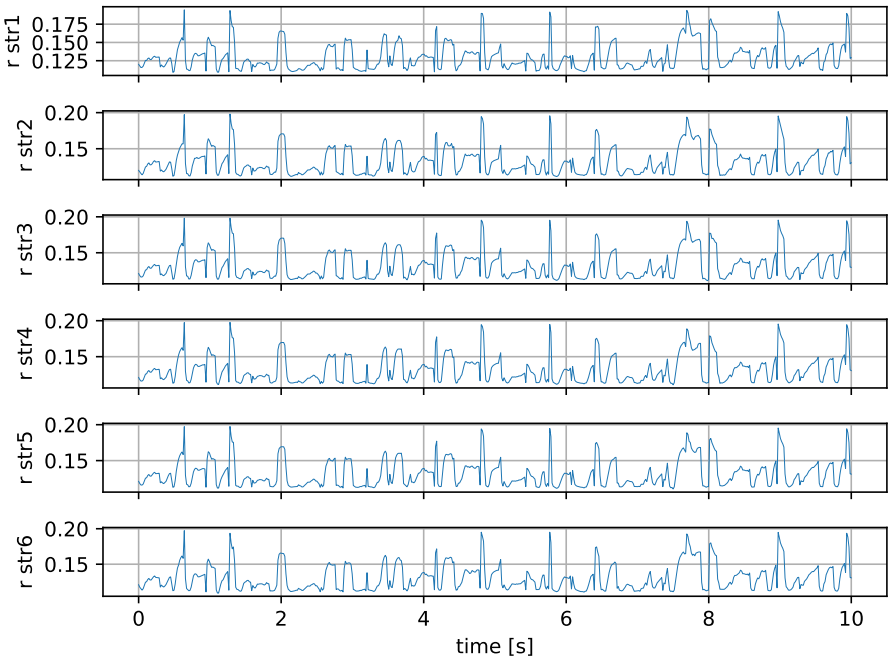


Figure 8.9 – ExB4: parameter r.

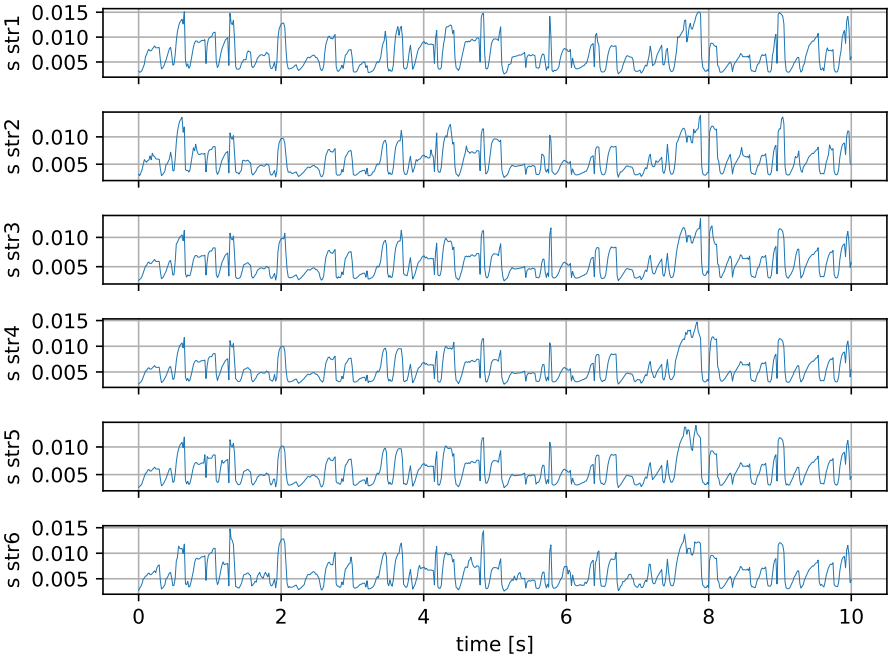


Figure 8.10 – ExB4: parameter b.

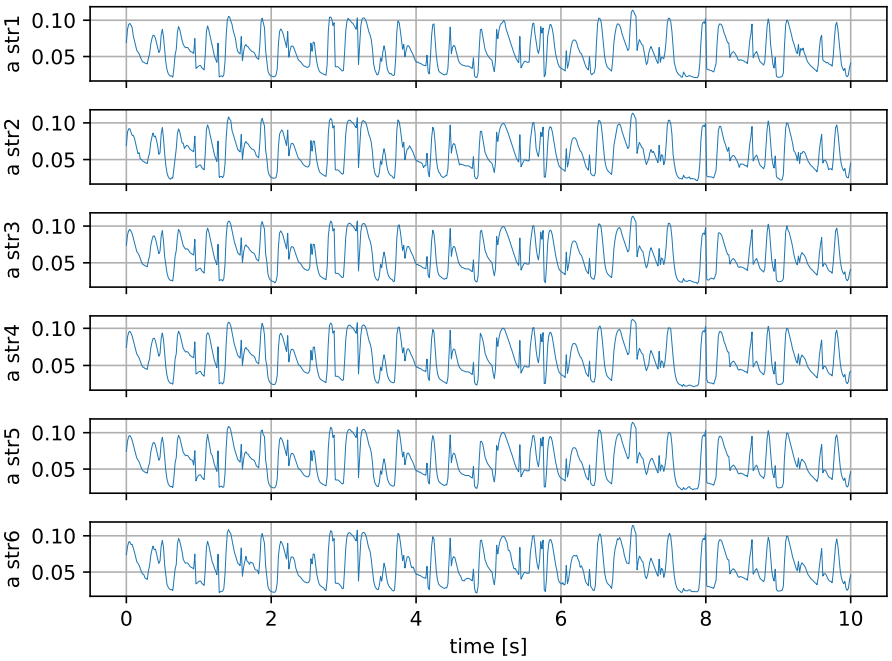


Figure 8.11 – ExC1 parameter a.

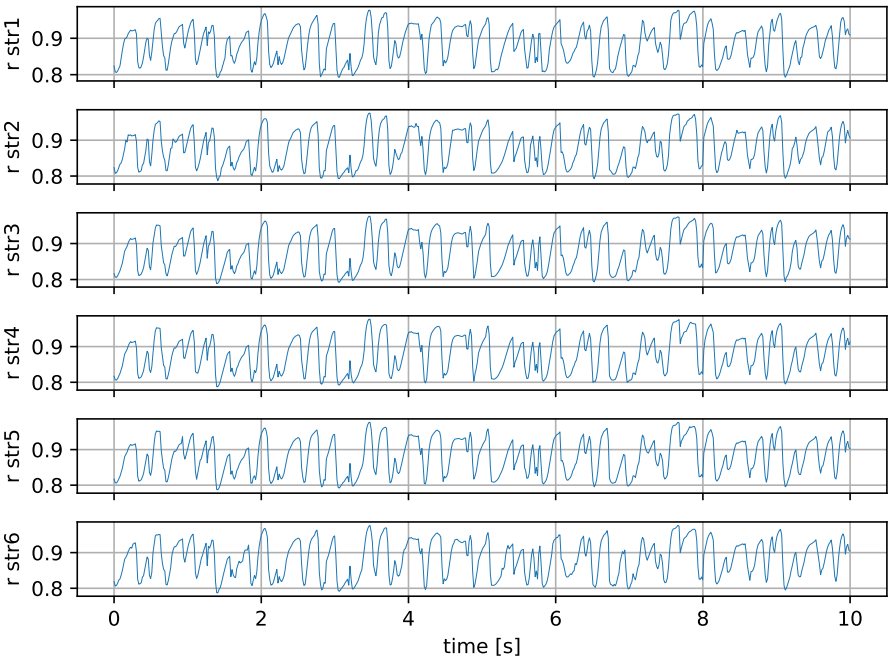


Figure 8.12 – ExC1 parameter r.

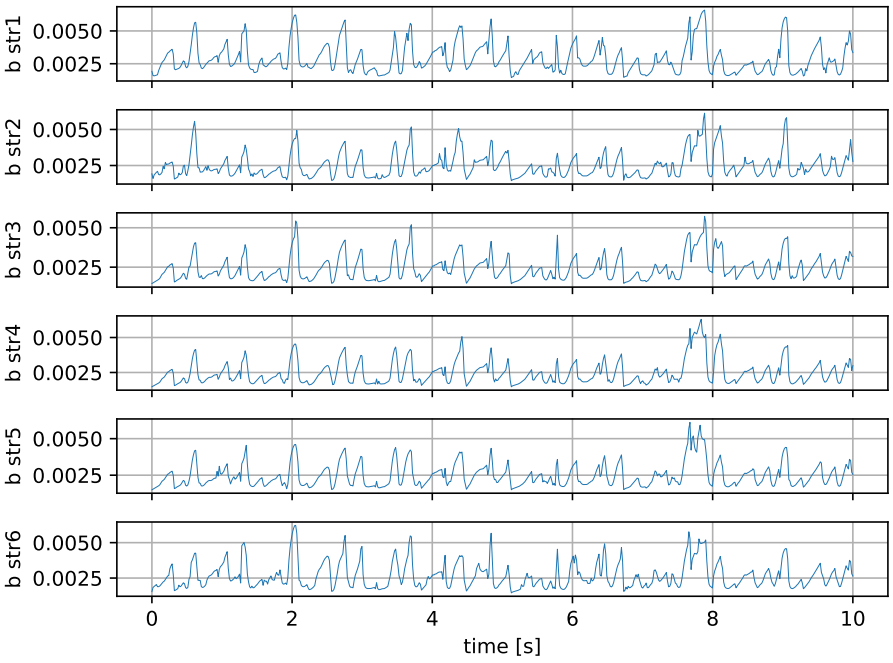


Figure 8.13 – ExC1 parameter b.

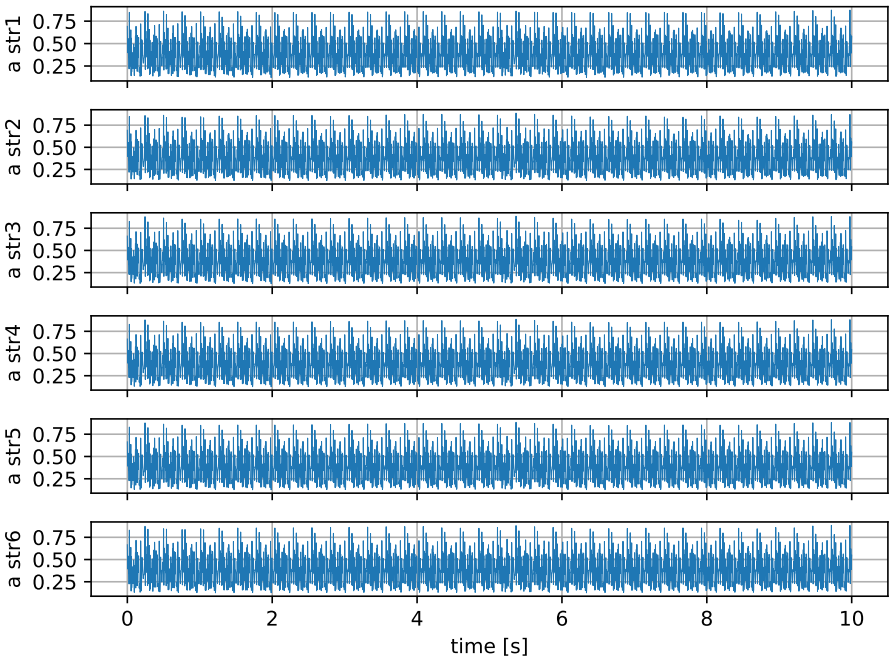


Figure 8.14 – ExC2 parameter a.

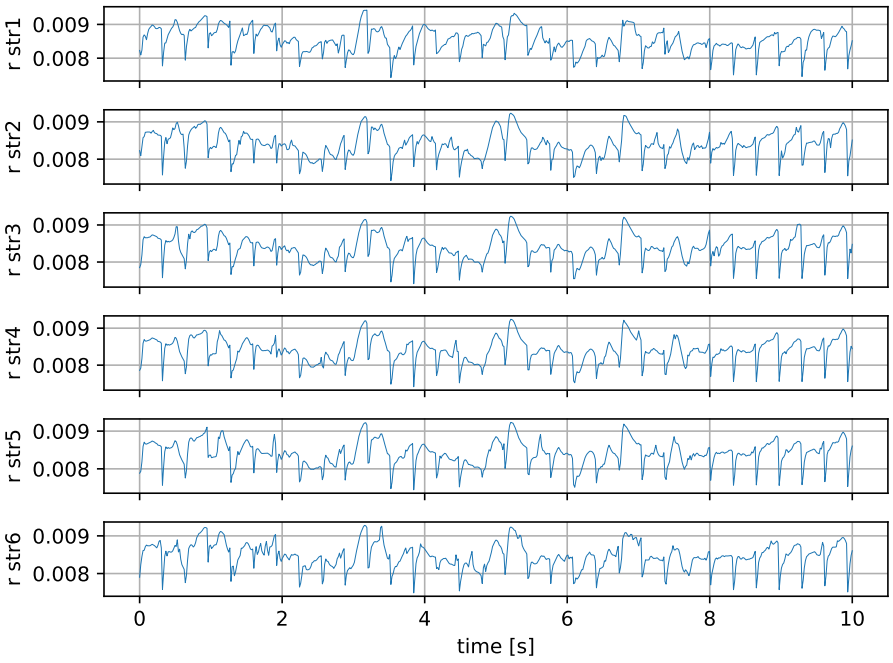


Figure 8.15 – ExC2 parameter r.

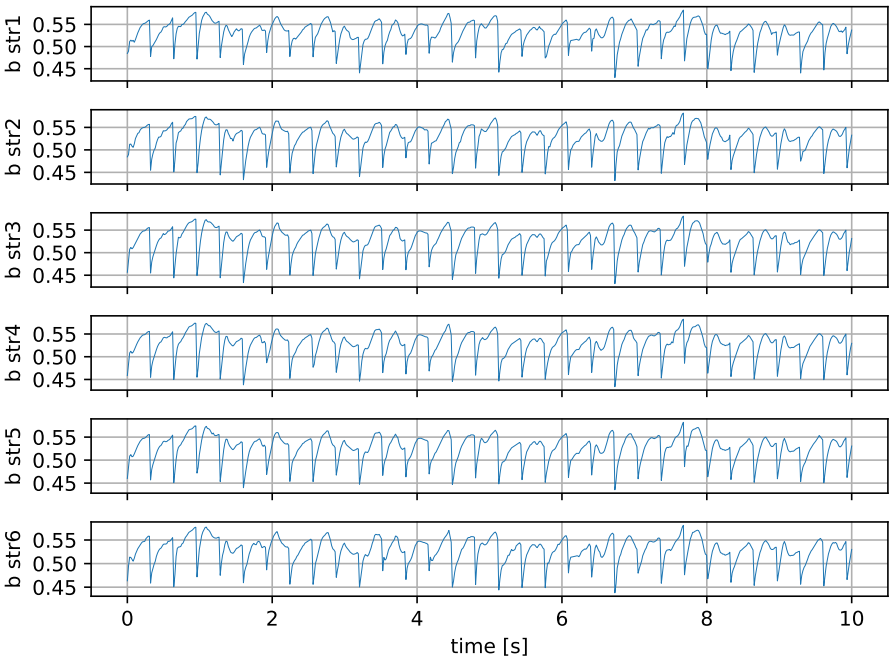


Figure 8.16 – ExC2 parameter b.