BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

APPLICATION FOR GUITAR SOUND SEPARATION FROM MUSIC RECORDING

APLIKACE PRO SEPARACI KYTAROVÉHO ZVUKU Z HUDEBNÍ NAHRÁVKY

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. NATÁLIA HOLKOVÁ**
AUTOR PRÁCE

SUPERVISOR **Ing. LADISLAV MOŠNER,**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

148426

Institut:	Department of Computer Graphics and Multimedia (UPGM)
Student:	**Holková Natália, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Machine Learning
Title:	**Application for Guitar Sound Separation from Music Recording**
Category:	Signal Processing
Academic year:	2022/23

Assignment:

1. Get acquainted with state-of-the-art models for source separation.
2. Obtain a suitable training and test music corpus containing annotations of guitar stems.
3. Select and train source separation model to perform guitar sound extraction.
4. Design an application that integrates the trained model and provides a user interface.
5. Implement the application using selected libraries and development tools.
6. Evaluate the guitar-extracting model in terms of objective metrics. Perform subjective evaluation of the user interface.

Literature:

- Defossez, A., Usunier, N., Bottou, L., Bach, F. (2019). Music Source Separation in the Waveform Domain. *arXiv preprint arXiv:1911.13254*.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Mošner Ladislav, Ing.**
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	17.5.2023
Approval date:	31.10.2022

Abstract

This thesis aims to implement a model capable of separating guitar sounds from a recording and use it in a practical application. It was necessary to manually create our dataset from remixes of songs and modify the existing MedleyDB dataset for our purposes. We have chosen Demucs architecture as a basis for our neural network. We trained it from scratch to separate audio files into five distinct recordings containing drums, bass, vocals, guitars, and other accompaniment. We trained five models on MetaCentrum, which we evaluated objectively and subjectively. The implemented application serves as both a music player and an educational tool. The main feature is to allow users to listen to isolated instruments, for example, a guitar, and therefore more easily learn songs by ear. The application was subjected to user testing, and the knowledge learned will be used in future development.

Abstrakt

Cieľom tejto práce bolo implementovať model na separáciu gitarového zvuku z nahrávky a použiť ho v praktickej aplikácii. Bolo nutné manuálne vytvoriť vlastný trénovací dataset z remixov piesní a upraviť existujúci MedleyDB dataset pre naše účely. Ako základ neurónovej siete sme si vybrali Demucs architektúru, ktorú sme od základu učili rozdeľovať audio súbory na celkovo päť samostatných nahrávok obsahujúcich bicie, basgitaru, vokály, gitaru a zvyšné nástroje. Celkovo sme na MetaCentre natrénovali päť rôznych modelov, ktoré boli objektívne aj subjektívne vyhodnotené. Implementovaná aplikácia slúži ako hudobný prehrávač a zároveň výučbový nástroj. Hlavnou funkcionalitou je, že umožňuje používateľovi počúvať izolovaný nástroj, napríklad gitaru, a vďaka tomu sa ľahšie učia piesne podľa sluchu. Aplikácia bola podrobená užívateľskému testovaniu a zistené poznatky budú využité pri ďalšom vývoji.

Keywords

music source separation, neural networks, Demucs, Python, PyTorch, PyQt, guitar sound separation

Kľúčové slová

separácia hudobných zdrojov, neurónové siete, Demucs, Python, PyTorch, PyQt, separácia gitarového zvuku

Reference

HOLKOVÁ, Natália. *Application for Guitar Sound Separation from Music Recording.* Brno, 2023. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ladislav Mošner,

Rozšírený abstrakt

Táto práca sa zaoberá problematikou separácie gitarového zvuku z hudobnej nahrávky pomocou neurónových sietí a následne tvorbou praktickej aplikácie, ktorá využíva natrénovaný model. Existujúce metódy na separáciu hudobných nástrojov sa nezameriavajú na gitarový zvuk a nahrávky rozdeľujú iba na vokály, bicie, basgitaru a zvyšok. Nami navrhnutá sieť stavia na jednej z takýchto metód a upravuje jej architektúru, aby poskytovala jeden výstup navyše pre gitaru.

V práci popisujeme dve existujúce metódy, ktoré riešia túto problematiku, a to Demucs a Hybrid Demucs. Demucs je neurónová sieť typu autoenkodér založená na U-Net architektúre. Hybrid Demucs priamo rozvíja túto sieť tým, že pridáva konvolučnú rekurentnú časť, vďaka čomu dosahuje lepšie výsledky. Našu prácu staváme na Demucs architektúre.

Existujúce modely pre separáciu zvuku sú trénované a testované na dátových sadách MUSB18 a MUSB18-HQ, ktoré ale nie sú použiteľné pre naše účely, pretože neobsahujú samostatné nahrávky gitary. Z tohto dôvodu sme manuálne vytvorili vlastný dataset z remixov piesní. Pre zachovanie dobrej variability sme sa obmedzili na jednu skladbu od daného interpreta. Z tohto datasetu sme vyčlenili časť nahrávok pre validačné účely a časť na finálne testovanie modelov. Ďalej sme využili existujúci dataset MedleyDB, ktorý ale samostatne nepostačoval pre úspešné trénovanie neurónových sietí.

Implementácia neurónovej siete prebehla v jazyku Python s využitím knižníc PyTorch a Torchaudio. Na trénovanie našej metódy založenej na architektúre Demucs sme využili MetaCentrum. Najprv sme uskutočnili niekoľko menších experimentov na nájdenie najlepších hyperparametrov a následne sme celkovo natrénovali päť rôznych modelov, ktoré sa líšili veľkosťou trénovacej sady a aj komplexnosťou architektúry. Modely sme potom podrobili evaluácii na testovacej sade, pri ktorej sme uvádzali metriky používané v oblasti separácie zvuku, ako je SDR, SIR, ISR a SAR. Na základe výsledkov dotazníka sme subjektívne porovnali jednotlivé modely. Výsledkom objektívneho aj subjektívneho testovania je, že model natrénovaný na najväčšej datovej sade s najkomplexnejšou architektúrou dosahuje najlepšie výsledky z nich.

Natrénované modely sú využívané v počítačovej aplikácii, ktorá bola implementovaná v jazyku Python a s využitím knižnice PyQt. Pri dizajne sme sa zameriavali na jednoduchosť a ľahkosť navigácie. Aplikácia slúži ako hudobný prehrávač a zároveň výučbový nástroj. Používateľovi umožňuje počúvať izolovaný hudobný nástroj, ako napríklad gitaru, a vďaka tomu sa ľahšie učiť piesne podľa sluchu. Aplikácia bola podrobená užívateľskému testovaniu, ktorého cieľom bolo vyhodnotiť celkový dizajn aplikácie, mieru responzívnosti a prehľadnosť navigácie. Vďaka tomuto testovaniu sme získali návrhy na ďalšiu možnú funkcionalitu aplikácie.

Pri našich natrénovaných modeloch je stále priestor na zlepšenie. Napriek tomu, že nedosahujú výsledky porovnateľné s najmodernejšími metódami, slúžia dostatočne na poukázanie možností a výhod ich zakomponovania do praktickej aplikácie. V budúcnosti sa budeme sústrediť iba na separáciu gitary a využijeme už predtrénované modely na zvyšné nástroje. Implementácia aplikácie by mohla profitovať z použitia iného jazyka a knižnice na tvorbu grafického rozhrania, ktoré by bolo lepšie prispôsobené prehrávaniu audio médií.

Application for Guitar Sound Separation from Music Recording

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Ladislav Mošner. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Natália Holková
May 14, 2023

Acknowledgements

I would like to thank my supervisor for his guidance and help with the thesis. Computational resources were provided by the e-INFRA CZ project (ID:90140), supported by the Ministry of Education, Youth and Sports of the Czech Republic.

Contents

Chapter 1

Introduction

Music source separation is a field of study that isolates individual sounds or instruments from a recorded mixture of sounds. The objective is to separate different sources of a song, such as vocals, drums, and bass, so that they can be manipulated independently. This can be useful for various applications, such as music production, audio analysis, and learning musical instruments.

This thesis aims to create a music source separation model capable of separating the guitar sound from the mix as, to our knowledge, the state-of-the-art methods in the field have yet to focus on it. Once such a model is created and trained, it will be used in a practical educational application aimed at amateur musicians and people learning to play musical instruments. Its purpose is to make transcribing music by ear easier for beginners.

The thesis is organized as follows. Chapter 2 introduces state-of-the-art methods in music source separation and explains their details. Also, it covers the necessary theory behind neural networks and creating GUI. Next, Chapter 3 covers the evaluation metrics and existing datasets used to train models mentioned in the previous chapter. Furthermore, it also discusses the process and results of creating own dataset. Chapter 4 considers the process of designing and implementing a waveform-to-waveform neural network capable of separating instrument tracks in songs. Additionally, it describes the training process.

Next, Chapter 5 focuses solely on creating a practical application that uses the previously trained model. It discussed the initial design creation and implementation of the application. Chapter 6 evaluates both the neural network and its application. Concerning the trained models, it details the various conducted experiments, shows results on benchmark datasets, and also presents subjective evaluation. Furthermore, it details the user testing of the application. Lastly, we discuss the limitations and plans for the future.

Chapter 2

Theory

This chapter introduces the field of music source separation. First, we briefly touch on the two main approaches to this problem. Afterward, we briefly discuss the theory behind neural networks and their building blocks before describing some of the existing state-of-the-art methods in this field. Lastly, we focus on one of the libraries used for developing the GUIs, as that is the framework we chose for our application.

Music source separation is the process of separating individual instruments or vocals from a mixed audio signal. This task has been the subject of extensive research in signal processing and machine learning, aiming to improve the quality of audio recordings and perform tasks such as remixing and music transcription.

The methods used for music source separation have evolved significantly, reflecting advances in signal processing techniques and machine learning. More recently, deep learning methods have gained popularity for music source separation, using the power of neural networks to learn complex mappings between mixed audio signals and their sources. These methods can be divided into spectrogram-based methods, waveform-based, and their combination [5].

Spectrogram-based methods have been a popular approach for music source separation due to their effectiveness in representing the time-frequency content of audio signals. These models typically take the spectrogram representation of a mixed audio signal as input and output the corresponding spectrogram representations of each source. One advantage of spectrogram-based models is their ability to handle multi-pitch and polyphonic sources.

Waveform-based methods for music source separation are an alternative approach that operates directly on the time-domain waveform of a mixed audio signal. These methods typically involve training a model to predict the waveform of each source given the waveform of the mixed signal. Waveform-based methods have the advantage of preserving the temporal information of the audio signal, which can be essential for maintaining the natural timing and phrasing of individual sources.

This chapter introduces state-of-the-art models for music source separation—namely, the Demucs [5] and Hybrid Demucs [6]. We will discuss underlying principles of the approaches, their strengths, and weaknesses.

2.1 Neural networks

Neural networks are a machine learning model inspired by the structure and function of the human brain. They consist of multiple layers of nodes called neurons that process the

information. The power of artificial neural networks arises from the interactions between a large set of neurons [15].

Neural networks are trained using backpropagation, an optimization algorithm that modifies the weights and biases to minimize a loss function. Initially, the network weights and biases are initialized with small random values. Afterward follows the forward pass, where the input is fed into the network, and the output is calculated. Next, the loss is calculated as the difference between the predicted and true outputs. During the backward pass, the error is propagated backward through the network to compute the gradients of the loss function with respect to the weights and biases of the network. Finally, the network weights and biases are updated using an optimization algorithm like Adam. These steps are repeated for several epochs until the network performs well on the training data [10].

Next, we introduce some of the building blocks of the neural networks.

2.1.1 Convolutional layer

The convolutional layer is based on learnable filters or kernels, which are small in size. The filter is convolved across the input volume and computes the dot product between the filter entries and the input. This produces an activation map of a filter. This way, the network learns kernels that activate when they see a specific feature at a given input position [22].

The following parameters mainly define convolutional layers:

- the filter/kernel - represented as m×n matrix
- the stride - defines how we slide the filter across
- zero-padding - adding zeroes on the input image border

2.1.2 Loss Functions

The loss or cost function is used to evaluate a candidate solution (the set of weights). During training, the network iteratively adjusts its weights and biases to minimize the loss. The choice of loss function depends on the type of problem being solved. Loss functions include the L1 loss or MSE loss [16].

L1 loss, also called the mean absolute error, is calculated by taking the absolute difference between the predicted values and the true values. This is expressed in the following equation:

$$L1 = \sum_i |y_i - f(x_i)|, \tag{2.1}$$

where y_i is the true value and $f(x_i)$ is the estimated value.

Unlike other loss functions, for instance, the mean squared error, L1 loss is more robust to outliers in the data. This is because it penalizes outliers more heavily than small errors. It is also computationally less expensive.

On the other hand, in *Mean Squared Error* loss, also known as L2 loss, the difference between the predicted value and the actual value is squared. MSE loss is more stable compared to L1 due to its continuous nature. Unfortunately, it is also more computationally expensive. The formula for MSE loss is as follows:

$$MSE = \sum_i (y_i - f(x_i))^2, \tag{2.2}$$

where y_i is the true value and $f(x_i)$ is the estimated value.

2.2 Existing methods for music source separation

2.2.1 Demucs

Demucs is a waveform-to-waveform model for music source separation [5], which separates audio into 4 distinct categories: *vocals*, *bass*, *drums*, and *other*. At the time of its creation, there were models already using the waveform approach, such as Wave-U-net [31], but all of them performed significantly worse than their spectrogram-based counterparts. However, with proper data augmentation, Demucs was able to surpass all of the state-of-the-art architectures available at the time.

2.2.1.1 Architecture

When designing the model, the authors were inspired by *Conv-Tasnet*, a model initially developed for speech source separation [17]. Speech source separation is about separating multiple speakers' utterances from a mixture thereof. Compared to the music source separation, speech separation only deals with monophonic audio, usually sampled at 8 kHz or 16 kHz. While the adapted Conv-Tasnet architecture in [5] achieved high accuracy, when listening to the generated audio, listeners observed significant artifacts, mainly in drums and bass sounds.

Therefore the Demucs' authors had to make modifications. They chose a new model with **U-Net** architecture as the basis. U-Net is a type of convolutional neural network initially developed for biomedical image processing [26]. It modifies the fully convolutional network so that it has fewer parameters and better segmentation. They do this by adding new layers to the normal convolutional network, but the usual pooling operations are replaced by upsampling. This way the resolution of output increases. The following convolutional layer can even assemble a precise output.

Demucs makes several changes to the described U-Net architecture. First of all, the U-Net architecture was initially developed for biomedical image processing, it has to be adapted to work with sound waveforms.

One such model that uses the U-Net architecture is the *Wave-U-Net* [31]. This model was developed for the task of music source separation, too. Moreover, it takes raw audio as input, much like Demucs. Unfortunately, when it was submitted to the *SiSEC* campaign 2018 [1], it performed poorly compared to its spectrogram counterparts, achieving an average SDR (explained in 3.1.1.4) of *3.17 dB* as opposed to a score of *5.97 dB* achieved by *MMDenseLSTM* [32], the best-performing model of said campaign.

From the structure of Wave-U-Net Demucs retains its encoder/decoder architecture. The complete structure can be seen in Figure 2.1. It consists of several main components:

- convolutional encoder
- bidirectional LSTM
- convolutional decoder

The encoder and decoder layers are linked with *skip U-Net connections*. Skip connections help deep neural models with degradation in performance as the depth of the model increases [30]. In essence, they are shortcuts between some layers, feeding the output of one layer into the input of another. In the U-Net architecture, they enable the network to use the fine-grained details learned in the encoder to construct an image in the decoder.

[1] SiSEC campaign official website: https://sisec18.unmix.app/

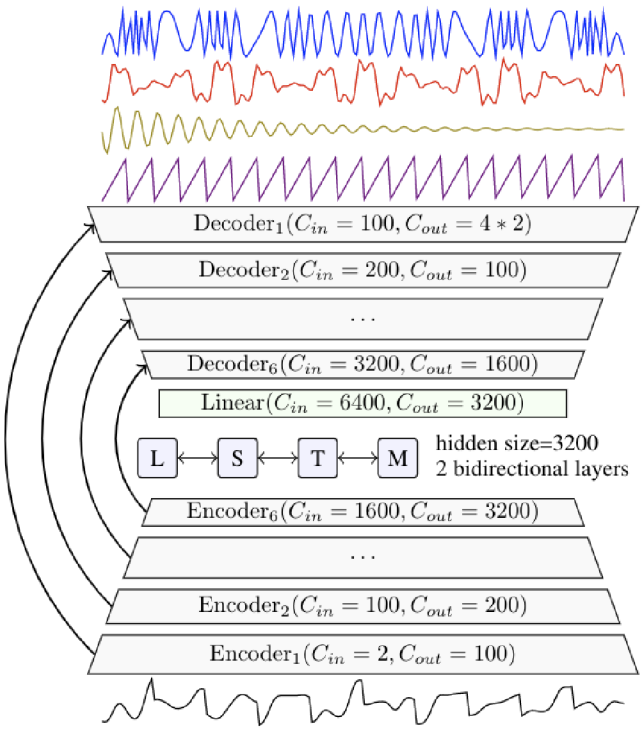


Figure 2.1: Demucs architecture [5]

In addition, the authors of Demucs drew inspiration from other fields, namely music note synthesis [7]. In particular, the use of transposed convolution was motivated by advances in this field. Moreover, experiments in this area of study have shown that using batch normalization hurt performance.

Next, we take a closer look at the individual components of the architecture.

Encoder

A single encoder consists of 6 stacked convolutional blocks. As shown in Figure 2.2, each block is made up of a one-dimensional convolution with a kernel of size 8 and stride 4, connected to a ReLU activation function. Following that is a 1x1 convolution to make the network deeper and more expressive at a low cost. Finally, the authors added a GLU (gated linear units) [4] as an activation which resulted in a performance improvement. The number of input channels is doubled with each block, starting at 2.

Bidirectional LSTM

Between the encoder and decoder layers lies the bidirectional LSTM (Long short-term memory).

LSTM is a type of recurrent neural network first proposed in [13]. The main feature of the network is its ability to hold information for future processing.

Bidirectional LSTM is an extension of the basic LSTM. One can imagine it as two models, where one learns the normal input sequence and the second learns its reverse. The

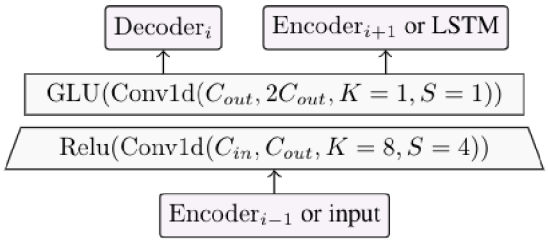


Figure 2.2: Demucs encoder in detail [5]

main advantage of this approach is that this way each component has information about both the past and future. [40]

An example of the bidirectional LSTM is displayed in Figure 2.3.

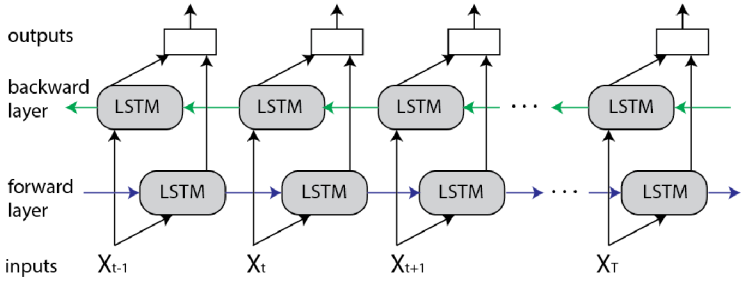


Figure 2.3: Structure of bidirectional LSTM [40]

Decoder

The decoder, depicted in Figure 2.4, is for the most part a reverse of the encoder. It, too, is comprised of 6 stacked layers. Each starts with a convolution with a kernel of size 3 and stride 1 intending to provide context about adjacent time steps. Next follows a ReLU function. After that, a *transposed convolution* of size 8 and stride 4 is used.

Transposed convolution is different from deconvolution which simply reverses the operation. Generally, it is used for upsampling, meaning it generates output with more spatial dimensions than its input [1]. Alternatively, to achieve upsampling of the signal, linear interpolation could be used instead, which is what Wave-U-Net uses. However, it has the disadvantage of not being able to generate high frequencies. Furthermore, for the same upsampling factor, transposed convolutions require fewer operations and less memory.

Lastly, a ReLU activation function follows the transposed convolution, except for the final decoder layer. That specific layer is linear as it directly produces all the output channels, in the case of Demucs, 4 stereo waveforms representing separated vocals, bass, drum, and other accompaniments.

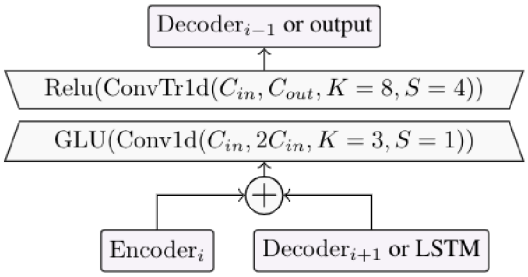


Figure 2.4: Demucs decoder in detail [5]

2.2.1.2 Training

Loss

As the reconstruction loss, the authors were deciding between the *average absolute error* (denoted here as L_1) and *average mean squared error* (denoted as L_2) between waveforms. The losses can be calculated with these equations:

$$L_1(\hat{x}_s, x_s) = \frac{1}{T}\sum_{t=1}^{T}|\hat{x}_{s,t} - x_{s,t}|, \tag{2.3}$$

$$L_2(\hat{x}_s, x_s) = \frac{1}{T}\sum_{t=1}^{T}(\hat{x}_{s,t} - x_{s,t})^2, \tag{2.4}$$

where:

x_s = waveform containing T samples representing source s
$\hat{x}_s$ = predicted waveform for x_s
$x_{s,t}/\hat{x}_{s,t}$ = t-th sample of waveform

Both losses are viable for music source separation. For most of the experiments, the Demucs' authors used the L_1 loss. Nevertheless, there were no noteworthy differences between the L_1 and L_2 loss.

Weights initialization

The initialization of weights can be critical to the model's performance. With the right initialization, models can be trained even without batch normalization.

The conventional initialization technique for U-Net-style architectures is the *Kaiming (He) initialization* introduced in [11]. This method takes into consideration the non-linearities of the activation functions and should prevent the exponential growth of input signal magnitudes.

However, the authors chose to instead employ a trick where they use specific learning rates for each layer. Focusing on a single convolutional layer, we denote its weight as they were first initialized as w. We compute $\alpha = std(w)/a$, where a is the reference scale empirically set to 0.1. The weights w are then replaced by w', which is defined as $w' = w/\sqrt{\alpha}$. This approach leads to better convergence and faster decay of loss. [5]

Training setup

Demucs was trained on the MusDB dataset(characterized in 3.2.1) as well as on an additional dataset, where they manually prepared 150 new songs. To provide better variety, during training sources within one batch were shuffled, channels were randomly switched and waveforms were randomly scaled.

The model was trained for a duration of 360 epochs or 240 epochs when using extra data. Other used hyper-parameters were the batch size of 64 and the use of Adam optimizer with a learning rate set to 3e-4.

2.2.1.3 Evaluation results

Demucs trained only on MusDB achieved an average SDR(described in 3.1.1.4) of *6.28 dB* across all sources and an SDR of *6.79 dB* with the use of extra data. In both cases, Demucs outperforms the best spectrogram-based method at the time, the D3Net [33]. Needless to say, it surpasses the previous waveform-base method, the Wav-U-Net.

2.2.2 Hybrid Demucs

Hybrid Demucs, introduced in [6], is a hybrid source separation model, meaning it works in waveform and spectrogram domains. At the time, it was the first model to use this strategy. We describe it here because it is one of the popular state-of-the-art models. It was the winner of the Music Demixing Challenge (MDX) 2021. Similarly to all the other models, it also focuses on separating music tracks into four categories - vocals, bass, drums, and the rest.

As could be derived from its name, it builds upon the *Demucs* [5] model from the same author. It utilizes its strengths and fixes some of its shortcomings. Namely, while it surpassed state-of-the-art spectrogram methods at the time in terms of average SDR(3.1.1.4), its performance when separating vocals and other accompaniments was slightly inferior.

2.2.2.1 Architecture

The model splits into two parallel branches. They are not entirely separate, as they share some layers. Both employ the U-Net architecture, with one working in the time domain and the second in the frequency domain.

The time domain branch works directly with sound waveforms and is virtually the replica of the original Demucs (detailed in 2.2.1.1). The only significant change is the replacement of all ReLU activation functions with *Gaussian Error Linear Unit* (GELU). GELU, presented in [12], serves as regularization and activation. Experiments in natural language processing showed it improves accuracy when tested against ReLU-only models.

The frequency, also called the spectral branch, first creates the spectrogram using the *Short-time Fourier transform* (STFT). The spectrogram is a visual representation of the spectrum of frequencies in time, typically portrayed as a heat map. It is calculated utilizing the *Discrete Fourier transform* (DFT) of a small sliding window on the input waveform. The window and hop length are chosen to match the temporal branch.

Convolutions are applied to reduce dimensions. Since the branch follows U-Net architecture, there as several spectral encoders. After that, a shared layer sums up both inputs from both branches. Following that is a decoder part with skip connections from corresponding encoders.

Next, the output of the frequency branch is transformed into a waveform using the *Inverse Short-time Fourier transform* (ISTFT). Summing the outputs of both branches yields the final prediction. The complete architecture can be seen in figure 2.5.

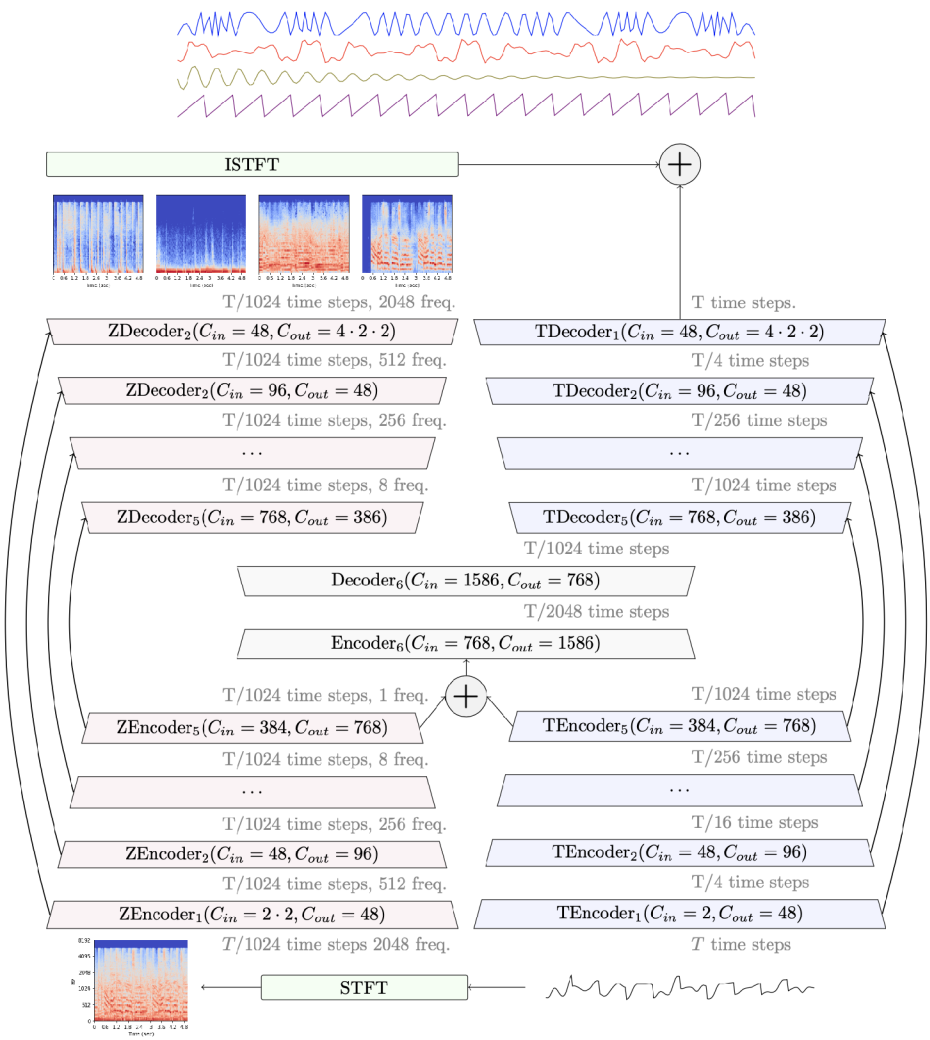


Figure 2.5: Hybrid Demucs architecture [6]

The benefit of this structure is that the model itself can choose which sound representation to use to separate each instrument. For instance, since Demucs performed best on separating drums, Hybrid Demucs might prefer to use the time branch output as the final prediction. On the other hand, separating other instruments was less successful for the waveform model, therefore a spectral approach will be used here.

2.2.2.2 Training

As the model took part in the MDX challenge, it was trained primarily on the MusDB18-HQ (described more in 3.2.1). A separate version of the model using extra training data was also prepared and evaluated.

The model was implemented using mostly the PyTorch library for Python.

To evaluate the model's performance, a new version of the standard SDR metric was used. This metric is described more in 3.1.1.5.

2.2.2.3 Evaluation

The version of the Hybrid Demucs trained only on MusDB18-HQ achieved an average SDR of *7.33 dB*, making it the winner of the MDX challenge. The model performed best on drums and bass separation, with a score of *8.04 dB* and *8.86 dB* respectively, similar to the original Demucs. For vocals and other instruments separation, it was inferior to the *KUIELAB-MDX-Net*.

2.3 PyQt

PyQt[2] is a set of bindings in Python for the Qt framework – a popular cross-platform GUI toolkit written in C++. The Qt Group is currently developing Qt. It can run on numerous software and hardware systems, including Windows, MacOS, Linux, or Android. PyQt is capable of much more than GUI development. It also provides tools for, for instance, SQL databases, network communication, graphics, and much more [37].

PyQt also comes with **Qt Designer**, designed to make the GUI layout much quicker using the drag-and-drop mechanism. However, we did not use it as our application did not have such a complex design that we could not program it ourselves manually.

GUI written in PyQt is *event-driven*. It either responds to events that the user generates, such as a click of a mouse or the press of a key on a keyboard, or an event the system generates. The act of responding to these events is known as event handling. The application begins listening for events when `exec_()` is called and does not stop until it closes.

PyQt employs a *signals and slots* system [9] for communication between objects as an alternative to the callback technique. A signal is an event that occurs, whereas slots are the methods that are executed as a response to the signal. In other words, widgets send out signals, and we collect and use them with slots to force the application to perform actions. Widgets have predefined signals and slots, but we can also create custom signals.

For instance, let's inspect the following code:

```
splitButton.clicked.connect(split_song)
```

When we push the button, the `clicked()` signal is emitted. We *connect* it to a callable function `split_song()` that is executed after clicking the button.

In GUI applications attempting to perform long-running background tasks might cause the application to freeze due to the event-based nature of PyQt. Events are placed into an event queue and processed sequentially as they arrive. Calling `exec_()` by the `QApplication` object starts the event loop on the so-called GUI thread. Anytime the application executes code, the communication is frozen. With simple tasks, this freezing is imperceivable. However, the work must be done outside the GUI thread for more challenging tasks. Otherwise, the application would appear unresponsive. [8]

One of the approaches for executing independent tasks is to use threads. This is because threads share the memory space, unlike processes, which do not.

Running jobs in separate threads from the main GUI thread in PyQt can be done through `QRunnable` and `QThreadPool` classes. `QRunnable` serves as a container for the code

[2] https://pypi.org/project/PyQt5/

to perform. `QThreadPool` handles the execution of the `QRunnable` workers. Below is an example of a worker:

```
class Worker(QRunnable):
    def __init__(self):
        super(Worker, self).__init__()

    @pyqtslot()
    def run(self):
        # code to perform
```

To start the worker, we first acquire an instance of `QThreadPool` and call its `start()` method with the worker as an argument:

```
threadpool = QThreadPool()
worker = Worker()
threadpool.start(worker)
```

The worker can be further improved to run the desired function with arguments or emit a signal when it finishes.

Chapter 3

Evaluation Metrics and Datasets

This chapter introduces several popular metrics used to evaluate music source separation models. We divide these metrics into objective and subjective. Objective metrics include the Source-to-Artifacts Ratio, Source-to-Interference Ratio, and Source-to-Distortion Ratio.

Next, we describe the existing datasets for this field of study and discuss their usefulness for this thesis. Lastly, we explain the process of creating our custom dataset.

3.1 Evaluation Metrics

When it comes to evaluating the performance of music source separation models, there are two main approaches [18]:

1. objective metrics
2. subjective metrics

While the objective approach relies on calculations comparing original stems (the *ground truth*) to the model's outputs, the subjective approach depends on a listening test by human participants. Each method has its advantages and disadvantages. For example, while the objective metrics are fast to calculate, the subjective ones take longer to prepare and are more expensive. On the other hand, human listeners can compare models more accurately.

3.1.1 Objective metrics

The goal of music source separation is to separate the mixture s into K sources $s_1, ..., s_K$. The estimate of source s_i, denoted as $\hat{s}_i$, is made up of multiple components [18]:

$$\mathbf{\hat{s}_i} = \mathbf{s_{target}} + \mathbf{e_{interf}} + \mathbf{e_{noise}} + \mathbf{e_{artif}} \tag{3.1}$$

where:

$\mathbf{s_{target}}$ = true source
$\mathbf{e_{interf}}$ = interference error
$\mathbf{e_{noise}}$ = noise error
$\mathbf{e_{artif}}$ = artifacts error

The most widely used objective metrics include the **Source-to-Artifacts Ratio** (SAR), **Source-to-Interference Ration** (SIR), and **Source-to-Distortion Ratio** (SDR). Python

package `museval` [1] provides a reference implementation of these metrics. For the Music Demixing (MDX) Challenge of 2021 [2], the original SDR metric was altered.

3.1.1.1 Source-to-Artifacts Ratio

Source-to-Artifacts Ratio (SAR), described in [35], is:

$$\text{SAR} := 10 \log_{10} \frac{||\mathbf{s_{target}} + \mathbf{e_{interf}} + \mathbf{e_{noise}}||^2}{||\mathbf{e_{artif}}||^2}. \tag{3.2}$$

It stands for the amount of unwanted artifacts in the source estimate.

3.1.1.2 Source-to-Interference Ratio

Source-to-Interference Ration (SIR), introduced in [35], is computed as:

$$\text{SIR} := 10 \log_{10} \frac{||\mathbf{s_{target}}||^2}{||\mathbf{e_{interf}}||^2}. \tag{3.3}$$

It represents the „bleed“, or the extent to which the other sources are audible.

3.1.1.3 Source Image to Spatial distortion Ratio

While the Image to Spatial Distortion Ratio (ISR) metric was initially introduced in image restoration and super-resolution, it has also been used in music source separation to evaluate the quality of separated audio signals. It is typically applied to the magnitude spectrograms of the original and separated audio signals. The use of the ISR metric in music source separation allows for a quantitative evaluation of the quality of the separated audio signals in both the frequency and time domains [36].

It is calculated as:

$$\text{ISR} := 10 \log_{10} \frac{||\mathbf{s_{target}}||^2}{||\mathbf{e_{noise}}||^2}. \tag{3.4}$$

3.1.1.4 Source-to-Distortion Ratio

Source-to-Distortion Ratio (SDR), which was defined in [35], is calculated as:

$$\text{SDR} := 10 \log_{10} \frac{||\mathbf{s_{target}}||^2}{||\mathbf{e_{interf}} + \mathbf{e_{noise}} + \mathbf{e_{artif}}||^2}. \tag{3.5}$$

It is the most widely used metric and is considered to be overall the best measure of how good the separation is. The higher the SDR score, the better the model performs.

3.1.1.5 New Source-to-Distortion Ratio

The MDX introduced a new definition of SDR, denoted here as nSDR to distinguish it from the original SDR metric, which is simpler and faster to evaluate [21]:

$$\text{nSDR} := 10 \log_{10} \frac{||\mathbf{s_{target}}||^2 + \epsilon}{||\mathbf{s_{target}} - \hat{\mathbf{s}}||^2 + \epsilon} \tag{3.6}$$

[1] Package website: https://github.com/sigsep/sigsep-mus-eval

[2] Official challenge website: https://www.aicrowd.com/challenges/music-demixing-challenge-ismir-2021

where ϵ is a small constant to avoid zero division.

In the MDX Challenge, the final value reported was the average nSDR across all test songs.

3.1.2 Subjective metrics

For human evaluations, the **Mean opinion score** (MOS) is used. This numerical measure is useful anywhere where human subjective experience is valuable [14]. Common domains where it is used include video or audio quality evaluation.

As done for the Hybrid Demucs' assessment [6], subjects were asked to assess several samples on two main criteria:

- absence of artifacts,
- absence of bleeding.

Subjects evaluated each criterion on a scale from 1 to 5, with 1 meaning „bad“ and 5 meaning „excellent“. As individuals tend to avoid the lowest or best ratings, the range of 4.3 to 4.5 is considered an outstanding quality. The final score is the average.

3.2 Datasets

3.2.1 Existing datasets

There are several datasets available for music source separation systems. For this task, apart from the original mixture, the dataset must contain isolated stems of different instruments that make up the songs.

The most popular is **MUSDB18** [24], which is also one of the benchmark datasets.[3] This dataset provides 150 songs from different genres, which total to approximately 10 hours of music. These songs are already divided into train and test subsets. All tracks have a sampling frequency of 44.1 kHz and are stereo. The entire dataset has only approximately 5.7GB because recordings are encoded in the *Native Instruments stem format* with `.mp4` extension. These audio files are created with *Stem Creator* [4], which is a free tool used by DJs and music producers for creating multi-track recordings.

Unfortunately, MUSDB18 only contains stems for vocals, drums, and bass. All the other instruments, such as the guitar, are grouped, which makes this dataset itself unusable for our purposes. However, since this dataset was used to train the original **Demucs**, we took inspiration from its structure and tools used when creating our dataset which is further explained in Section 3.2.3.

Another dataset rising in popularity is the **MUSDB18-HQ** [25]. It was created as an alternative to the original MUSDB18 for models that work with broader ranges of frequencies as it uses uncompressed audio format `.wav`. Other than a different file format, it is virtually the same as MUSDB18. This dataset was used as the training dataset in the *Music Demixing (MDX) Challenge* of 2021 organized by Sony. The state-of-the-art model *Hybrid Demucs* [6], one of the models created as a response to this challenge, was trained using this dataset.

[3] Leaderboard of models using this dataset can be found at https://paperswithcode.com/sota/music-source-separation-on-musdb18

[4] Stem Creator official website: https://www.stems-music.com/stem-creator-tool/

[6] Source: https://sigsep.github.io/datasets/musdb.html

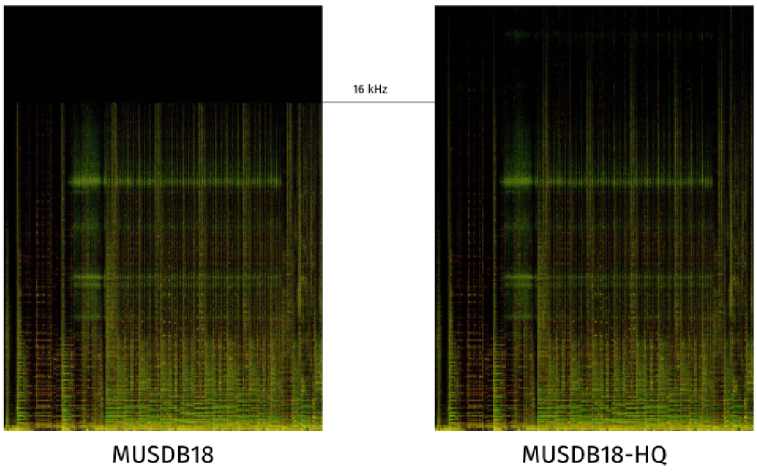


Figure 3.1: Comparison of the audio frequency range of the MUSDB18 and MUSDB18-HQ datasets.[6]

The **Slakh2100** [19] dataset offers a much greater selection of songs, totaling over 145 hours of recordings spread over 2100 tracks. This is possible due to the MIDI nature of instruments present in the dataset. While Slakh2100 contains individual guitar stems, it instead lacks vocals. Again, this dataset is unsuitable for our purposes.

The **MedleyDB** [3] and its newer version **MedleyDB 2.0** [2] provides exactly what we need. They feature a wider selection of instruments, including several types of guitars, such as clean electric sound, distorted, or acoustic. These datasets are available after submitting a request. Together, they total 196 multi-tracks. Apart from the final mixtures of songs, they contain both unedited raw recordings of instruments and edited stem tracks, where multiple recordings of the same instruments are grouped.

Unlike other datasets, MedleyDB contains a much greater amount of annotations, for instance, pitch and melody, and therefore can be used for a wider range of applications than just music source separation.

One downside of the dataset is its size. The compressed archive of version 1 is 43,1 GB and newer version 2 is 44,3 GB. The reason behind these enormous sizes is the lossless audio format that is used (`.wav`) and the fact that every song has generally more than 10 stem tracks and even more raw recordings. For models that require only the best audio quality can be perceived as an advantage, but as we will not have such a requirement, it made handling the dataset slightly problematic.

3.2.2 Creating dataset using existing models to separate songs

Music source separation models such as Wav-U-Net, Demucs, or Hybrid Demucs all focus on separating audio into 4 categories: *vocals*, *bass*, *drums*, and *other* accompaniment. This mirrors the structure of datasets they were trained on. Since this thesis aims to create a model capable of separating into 5 categories, with the new category being the *guitar* category, we cannot use the already established datasets like MUSDB18 or MUSD18-HQ.

Despite **MedleyDB** technically provides the necessary data to train a neural network, the amount is not great enough. Deep neural networks require massive amounts of data to

train properly. Only a small subset of songs in the dataset contains some type of guitar. Other times, a song has a guitar part, but lacks, for instance, a vocal recording. Although data augmentation such as multiplying source signals [31] might help with the issue, it would still not be enough.

The vast majority of state-of-the-art models for music separation apart from the MUSDB18 dataset use extra data for training. The original **Demucs** model creates its private dataset from around 2000 songs from a variety of genres [5] which it then uses for unsupervised learning.

At first, we intended to go a similar route, meaning the first step was to accumulate a large number of recordings. Afterward, we would artificially create isolated stems by using an existing model. At the time of writing, the model with the highest average SDR score was **Hybrid Demucs** [6]. The main problem was that to create good guitar stems, no other instruments apart from drums and bass could be present in the recording. To counter this, we explored discographies of famous musical trios[7], where members vast majority of times only play those three instruments we seek. Even this does not completely eliminate the previous issue, as in the studio bands tend to insert occasional flourishes on songs such as small keyboard parts or similar.

After downloading the discographies of potential artists, it was necessary to manually at least briefly listen to each of their songs and remove songs that contained unwanted instruments. This proved a time-consuming activity that also required concentration. A large portion of tracks was eliminated in the process.

The authors of Hybrid Demucs provided a Google Colab notebook[8], where one can use the model to separate their audio files. We took advantage of the possibility to mount your own Google Drive to the Colab notebook by uploading the songs there. Separating even a single album took a long time, but fortunately, it could be run in the background.

While to the human ear the drums and bass stems created by Hybrid Demucs seemed almost real, the guitars seemed less so. Therefore we chose to abandon this approach as there was still an unsolved issue of getting samples of other instruments of the network to learn on.

Moreover, compared to the dataset we acquired using the next method (described in Section 3.2.3), this dataset would have significantly less variance as a result of including multiple songs by the same artist.

3.2.3 Creating dataset using remixes to build up a dataset

The other approach to creating a training dataset was to acquire a large amount of multi-track recordings and then for each mixture reduce the number of tracks down to 5, namely tracks containing only vocals, bass, drums, guitars, and lastly other accompaniments.

Collecting original mixtures of popular songs with individual stems for each instrument present is hard to do without connections in the producing industry as these are not publicly available. Fortunately, there exist sites where users post their recordings so that others may try their hand at remixing.

The process of creating a single entry for the dataset included the following steps:

1. download an archive containing recordings

[7] Online article used as a source of inspiration for finding such bands at https://spinditty.com/genres/Best-Rock-Trios-of-All-Time

[8] Available at https://colab.research.google.com/drive/1dC9nVxk3V_VPjUADsnFu8EiT-xnU1tGH?usp=sharing

2. determine which instrument is played in each stem
3. join together files that contain vocals/bass/drums/guitars/other into a single file for each category
4. export all files to create the final mixture

At first, we downloaded all remix archives that listed that they contained guitar parts. However, they often contained many individual recordings, generally more than 15, which proved problematic. Furthermore, there was no way to automate determining which recording contained which instruments as the filenames did not follow any naming system, and no metadata were included.

Thankfully, we were able to later find remixes with a smaller number of stems and the same file structure as follows:

- `Bass (Mono).flac` – bass recording
- `Vocals.flac` – recording of vocals
- `Guitar.flac` – guitar recording
- `Cymbals.flac`, `Snare.flac`, `Kick (Mono).flac` – recording of drums
- `Keys.flac` – keyboard or piano recording
- `Backing.flac` – rest of the song not covered in the above

The clear naming system makes it easy to determine which audio file represents which instrument. If we were to disregard the `Backing.flac`, it would be possible to automate the process of creating only a single file for each of our selected categories. In this case only the separate recordings of individual drum components would have to be merged together. Note that this way, the only instruments that would appear in the *other* category in our dataset would be the keyboard and piano.

Having said that, we have decided to dismiss the option of automation and endure manual labor to acquire a dataset better representing real music. The manual part is listening to the `Backing.flac` track and determining what instruments it contains. Oftentimes guitars as well as other instruments play in it. Based on which one prevails and whether they overlap we remove one or the other. In addition, there is always a drum count-in in this track that has to be removed. We use the Audacity®[9] audio software for editing and exporting the final audio.

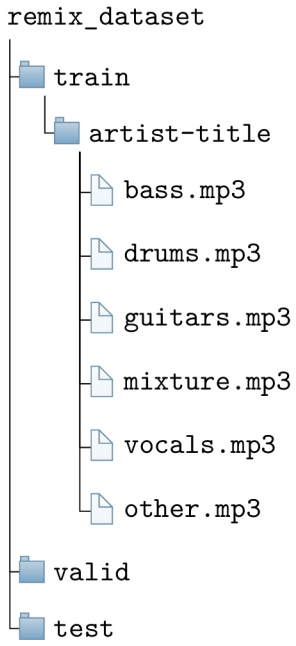
An important decision before proceeding further was to choose the encoding format. Different datasets use different audio formats. Both **MedleyDB** and **MUSDB18-HQ** use lossless `.wav`. As we first attempt to adjust the original Demucs architecture to allow the separating of the guitar, seemingly another option would be the `.mp4` format used in MusDB18. The advantage would be having one file for each song instead of 5 and possibly using the authors' data pipeline with minimal editing. However, unfortunately, the tool authors' of the mentioned dataset, the Stem Creator, allow importing of only five tracks in total – *drums*, *bass*, *synth*, *vox*, and *master file*. This means we would be short one track for the guitar. Therefore, we had to dismiss this encoding option completely.

Nevertheless, apart from creating the neural network model, this thesis aims to use it in a practical application where users will upload files to separate them. In this scenario, regular users are likelier to use the `.mp3`, a popular lossy compression format [28]. Therefore,

[9] Audacity is available at: https://www.audacityteam.org/

converting the `.mp3` back to `.wav` that existing models use would add additional delay for users.

For that reason, we decided to use `.mp3` with the highest bitrate of 320 kbps and a sampling rate of 44100 Hz, forcing all tracks to stereo. Also, we tried to adapt the naming structure of MUSDB18-HQ, the dataset Hybrid Demucs uses for training to be able to use parts of its data pipeline. Thus the final dataset has the following folder structure:



To increase the variety we chose to include only a single song from one artist in the dataset. This way we would get a wider representation of possible sounds since interprets often have similar distinct sound across multiple songs.

The final training dataset created from remixes has a size of 6.2 GB, contains 101 songs, and a total of 7h 13m 5s of recordings.

3.2.4 Additional datasets used

Despite the fact that **MedleyDB** and **MedleyDB 2.0** by themselves do not provide enough data to train a complicated neural network, we used them as additional datasets to enhance the training and provide variety. Both datasets had had to be first filtered by removing all entries not containing guitars. After some contemplation, we decided to keep even those mixtures that did not have the *other* components.

After the cleanup, out of the original 122 songs from MedleyDB remained 36 met our needs, and from MedleyDB 2.0 we kept 17. Altogether, this additional dataset provided 3h 18m 32s of data, with 2h 6m 14s being from the original MedleyDB and 1h 12m 18s being from MedleyDB 2.0.

Chapter 4

Design and implementation of neural network model

This chapter describes our choice of architecture for the neural network and the necessary changes needed to adapt it to our specific task. We also focused on the training process, which was done in MetaCentrum.

4.1 Design

We chose to take an existing model for music source separation as a basis and modify it to fit our purpose and data. Such a model had to fulfill several criteria. First of all, the model must have publicly available code with a license permitting us to use it in our project. Secondly, it needed to have satisfactory results on benchmark datasets. Additionally, it had to be relatively new, but not so brand new that it would not be used much. Lastly, we had to consider the effort it takes to train and apply in an application.

After much consideration, we chose two models: Demucs and Hybrid Demucs. Demucs, at the time of its creation, presented a significant breakthrough in the field. Although it is from 2019 and there are now better-performing models, Demucs is still used to this day. It is relatively straightforward and well-documented, making it an ideal starting point.

We initially chose Hybrid Demucs as the next model mainly because it was built upon Demucs and was designed by the same author. Additionally, we considered experimenting with Hybrid Transformer Demucs [27], a new iteration of Demucs that introduces a transformer [34] encoder block into the architecture. However, we decided against using both due to the time constraints and computational cost in the case of the Hybrid Transformer Demucs.

The first thing that needs addressing when modifying these existing models is the data pipeline. Both models expect specific file structures of their training data. When creating the dataset of remixes we opted for a file structure resembling that of Hybrid Demucs. Thanks to this we will only have to slightly adjust its pipeline.

Besides that, the structure of some layers of the neural network will have to be altered. Namely, the output layer will have to reflect the addition of another track to separate. Dimensions and parameters of other layers, for instance, the convolutional layers of encoder and decoder blocks, will be changed if experiments reveal improvement in accuracy.

4.2 Implementation

This section discusses the implementation details of the neural network model for music source separation.

4.2.1 Used technologies

4.2.1.1 PyTorch

PyTorch is a machine learning framework developed by Meta AI [23]. It is based on Python programming language and Torch library which is written in Lua. Presently, it is ranked among the most popular machine learning frameworks due to its ease of use and flexibility.

The main features of PyTorch are its tensor computations, GPU acceleration support, and automatic differentiation when creating neural networks.

4.2.1.2 Torchaudio

Another crucial tool when developing the music separation model was **Torchaudio**. Torchaudio is a library for audio and signal processing [39]. It is designed to work with the PyTorch framework. Apart from simply supporting the use of audio, it also provides tools for transformation, augmentation, or feature extraction from audio data.

4.2.1.3 Hydra

Hydra [38] is an open-source Python framework developed by Meta. It allows users to create a hierarchical configuration and even override it with additional configuration files or command line arguments [29].

Traditionally, Python parses arguments using the `argparse` library. When used in deep learning applications where all hyperparameters are passed this way, the code can soon become difficult to change. The advantage of Hydra is that one can edit parameters and constants without touching the code, leading to increased reproducibility.

The configuration file has the `.yaml` extension. Its location and name are defined in code with annotation typically above the main function. It is important to note that using Hydra changes the current working directory.

4.2.2 Neural network

As a basis for implementation, we took the code for Hybrid Demucs[1], which is written in Python and uses the PyTorch framework. The original Demucs code[2] used the classic Python library `argparse` to deal with command line arguments setting various model properties for different runs. The Hybrid Demucs changed this as it instead uses configuration files processed by the Hydra library. It also allows us to switch between the Demucs architecture and Hybrid Demucs easily.

This makes it easy to switch between the Demucs or Hybrid Demucs, as the model is chosen based on the variable model in the configuration file `config.yaml`. Furthermore, all model hyperparameters, such as depth or kernel size, are defined therein. Before proceeding

[1]Code for Hybrid Demucs is available at https://github.com/facebookresearch/demucs/tree/v3
[2]Code for Demucs is available at https://github.com/facebookresearch/demucs/tree/v2

further, we changed the dataset location and names of output sources to include the guitar track.

The first major hurdle when adapting the model appeared when we tried loading the remix dataset using `torchaudio.load()`. The newer versions removed the support for loading `mp3` files. So the solution was to limit the version of torchaudio to *0.11.0* and the PyTorch framework to *1.11.0*. Thanks to this, we could use the code to create custom dataloader in `mp3.py`.

Original Hybrid Demucs evaluates the model on the test dataset at the end of each training session. We removed this feature because we intended only to evaluate the final models once. To do this, we removed a portion of the code in function `train()` in `solver.py`.

The code for the evaluation on the test dataset is inside the `evaluate.py` file. Initially, the dataset was loaded using the `musdb` library instead of fetching the files from storage. We iterate through the test directory and manually load the audio files for the mixture and individual sources.

Because we execute the evaluation with only a single worker thread, testing on songs longer the approximately 30 seconds caused the program to crash. We solve this by splitting the estimates acquired from `apply_model()` and ground truth references into smaller segments. We can do this since models in music separation report the average metric value across all songs. However, this might negatively impact the score as the model has less context with shorter recordings.

We calculate the number of segments to split each track as follows:

$$\text{number of segments} = \text{total length}/\text{framerate}/\text{desired segment length} + 1 \tag{4.1}$$

Specific segments that contain only silence are removed from the evaluation as this would negatively affect the score. We use the *try-except* structure to catch those segments.

4.2.3 Training environment

MetaCentrum VO is a virtual organization available to researchers and students of academic and scientific institutions in the Czech Republic [20]. This organization allows using its computational and storage resources for computations that would be otherwise too demanding when performed locally. To become a member, it is first necessary to submit the application form and await the results.

MetaCentrum offers resources for grid computing, with a grid representing a network of interconnected computers with different properties such as RAM, CPU, or GPU. Batch jobs system PBS [3] tracks the grid's resources and organizes the computational jobs into queues until there are enough resources for execution. First, users prepare and submit their jobs on frontends. Then, the computations themselves are performed on computational nodes.

To access the grid, the user has to log into one of the frontends[4]. Users prepare and submit their jobs there. However, frontends are not meant for the execution of resources-intensive computations. Instead, those should be handled on one of the computational nodes.

Submitted jobs fall into two categories: batch jobs and interactive jobs. Batch jobs execute commands from a batch script prepared in advance. Alternatively, the interactive

[3]Scheduling system used is more described at: https://wiki.metacentrum.cz/wiki/About_scheduling_system

[4]List of all available frontends is at https://wiki.metacentrum.cz/wiki/Frontend

job allows users to run commands manually in real time instead of reading them from a script. This allows greater flexibility in handling errors. Interactive jobs are better suited for small tests or environment setups. It ends when the user logs out of the assigned computational node or the requested resources, such as the wall time, have been used up.

We need specific versions of certain libraries to train our version of Demucs for guitar sound separation. Most notably, the PyTorch framework requires versions older than 1.11.0 due to removed support for loading `.mp3` files. Torchaudio library needs version 0.11.0 at the newest for the same reason. Usually, MetaCentrum users can load PyTorch in their jobs through the command `module load pytorch`. However, this version is not compatible with our neural network.

Therefore, we use the Anaconda [5] instead to install our preferred versions of libraries. After loading the Anaconda module, we create a new „conda“ environment. Due to space quotas on MetaCentrum, we save this environment into our home directory. With the „conda“ environment activated, we first install PyTorch, torchaudio, and the appropriate cuda toolkit. The rest of the libraries needed for training are installed through pip, as some of them, such as Dora-search, are unavailable through Anaconda. Thus, the whole setup is best done using an interactive job to verify the installation's success. Listing 4.1 contains the exact setup script.

```
cd $HOME_DIR
module load anaconda3-2019.10
conda create --prefix=$CONDA_ENV_NAME
source activate $HOME_DIR/$CONDA_ENV_NAME
conda install pytorch==1.10.0 torchaudio==0.10.0 cudatoolkit=11.3 -c pytorch
pip install -r requirements.txt --no-cache-dir
```

Listing 4.1: Metacentrum environment setup script

The training was done on GPU nodes to speed up the training. Users can reserve such nodes using the `-q` GPU option and specify the required GPUs with `ngpus=X`. However, jobs on GPU nodes can run for a maximum of 24 hours. This means the model cannot be trained during a single run and must be trained across multiple sessions.

Some GPU nodes, such as the Konos, could not train our models due to their GPU memory size. The final models trained on either the full remix dataset, Medley dataset, or their combination were trained almost exclusively on Adan and Galdor machines. These machines have the following properties:

- Adan – 32 × CPU, 2 × GPU nVidia Tesla T4 16GB, 192 GB RAM
- Galdor – 64 × CPU, 4 × nVidia A40 48GB, 512 GB RAM

When initially training on other nodes, frequently, the program would end with a run-time error reporting that CUDA ran out of memory while trying to load the model. Specific steps could be taken to solve this problem, such as reducing the batch size, clearing the PyTorch cache, or lowering the number of channels produced by the convolutional layers and the depth of the model representing the number of recurrent layers.

However, lowering the batch size increases the training time, and we risk failing to complete even a single epoch before the allocated time is up. Also, while lowering model complexity reduces memory requirements, the model, at some point it is not able to learn correctly. Therefore it is necessary to reach some compromise.

[5] Anaconda's official website: https://www.anaconda.com/

Most of the time, we trained only using a single GPU. While using multiple GPUs can speed up training when appropriately utilized, jobs with multiple GPUs take longer to get assigned computational nodes. If we did not care which hardware executes the job, it would not be problematic to wait a few hours to begin execution. However, limited to only two viable machines in the cluster, we would have to manually search through all nodes and select only one at a time to execute the job. Otherwise, we could be assigned a node with insufficient GPU memory. We still do this when training the final models on a single GPU, but the probability of finding a viable node with free resources is much higher.

To train the model, we prepared the bash script 4.2:

```
#PBS -l select=1:ncpus=4:mem=32gb:scratch_local=20gb:ngpus=1
#PBS -l vnode=$NODE_NAME
#PBS -q gpu
#PBS -l walltime=24:00:00
#PBS -N $JOB_NAME
#PBS -m abe
module load anaconda3-2019.10
module load ffmpeg
cd $HOME_DIR
source activate $HOME_DIR/$CONDA_ENV_NAME
dora -P demucs run [config overrides]
```

Listing 4.2: Metacentrum training script

We requested 4 CPUs, 32GB of CPU RAM, and a single GPU in the script. Variable `$NODE_NAME` represents the exact name of the chosen node on which to run the job. Option `-m` allows users to receive updates about job status through email. After activating the prepared Anaconda environment, the training is initiated with the command `dora run`. We specify the location of the neural network's code with the switch `-P`. We can also run a model with different hyperparameters without changing the configuration file. When training a new model, we must specify this by adding `-clear` to the command. We can override the configuration property `continue_from` to load and prepare a specific model on any subsequent run.

4.2.4 Exporting model and track separation

To use a trained model to separate a song into individual instruments, it needs to be first exported. For this, we use the existing script `exportModel.py`. By default, after training, we save the models in `$HOME_DIR/demucs/outputs/xps`, each in its folder named after their signature. The model's signature is printed out at the start of each training session. The saved trained model can be exported by running the command:

```
python exportModel.py $SIGNATURE [-o path_where_to_store]
```

This saves the models in a single `.th` file, which can be more easily loaded. To finally use the model for splitting music, we modified the script `separatedTracks.py`, namely removing unnecessary assertions. Then, we execute it in the following manner:

```
python separateTracks.py $MP3_LOCATION -n $SIGNATURE --mp3 -d cpu/cuda
```

By default, results from separation will appear in folder `separated/$SIGNATURE/track_name`.

Chapter 5

Design and implementation of a practical application

Many people pick up musical instruments, for example, the guitar, to learn their favorite songs. Plenty of instructional videos or tablature exist for popular songs – a musical notation used by guitarists. Unfortunately, there is no such option for the less popular music. Instead, the aspiring musicians have to figure out how the song is played on their own.

This process is called transcribing, and it is an essential skill for a musician to have. It trains ears to listen and analyze recordings [1]. However, especially at the beginning, it can be challenging.

A beginner's ears might still need to be more accustomed to active listening and analyzing music. In addition to that, they have to filter out other instruments in the song to focus only on a single one they are trying to transcribe.

Part of this thesis was to create an application that would separate guitar sounds from recordings. The application integrates the neural networks described in Chapter 4. It is an educational tool for musicians, especially beginner to intermediate guitarists. By splitting the recording into individual tracks, the user can listen to isolated instruments. Listening to solo instruments should make it easier to transcribe the desired songs.

5.1 Initial design

When considering the potential design of the application, the main criterion was the ease of use. The application should be intuitive and without distractions. It would function as a music player with the additional option to separate songs. For inspiration, we explored existing music players and editors. Ultimately we chose to base our program on the Audacity audio software [2].

Audacity is a multi-platform open-source audio editor and recording software. Initially released in 2000, it remains among the most popular free audio programs. It supports a variety of post-processing effects for audio tracks. A notable visual feature we would like to imitate is the vertically organized tracks with a waveform graphs of the audio signals. Figure 5.1 shows the user interface of the Audacity audio software.

[1] More about transcribing music: https://www.leanmusician.com/post/transcribing-music
[2] Official website of Audacity: https://www.audacityteam.org/

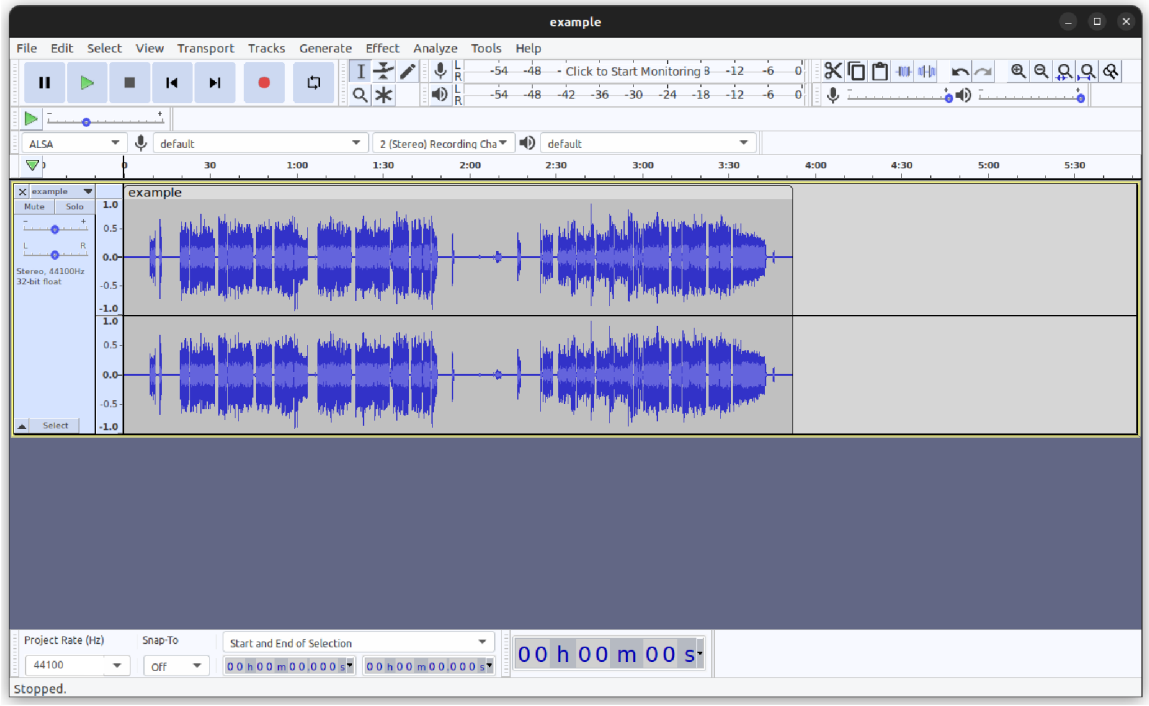


Figure 5.1: Audacity audio software

We created the initial design using **Figma**. Figma[3] is a tool for graphical design established in 2016. With more than 4 million users, it is well-known tool designers use for tasks such as brainstorming, creating prototypes, or even project management. It has a free plan or several paid options. For this thesis, the free tier was more than enough.

Figures 5.2 and 5.3 show the application interface design before and after the neural network splits the song into components. The program should provide functionality expected from a music player, including opening and playing audio files, adjusting the overall volume, or jumping to a specific position in a song. Functionality related to an audio separator includes an easily visible option to split songs into individual instruments.

In this design prototype, the intention was to always split the mixture into all five tracks – the bass, drums, guitars, vocals, and accompaniment. However, user tests showed that users would welcome the ability to choose which separated instruments to display. For example, a guitarist attempting to transcribe a song only cares about the guitar sounds. We have taken into account this suggestion during the implementation.

Each track is labeled according to the instrument played, and the final product should display the waveform graph of its soundwave. There is a button to mute each track to allow users to listen to isolated tracks. In the prototype, we also intended to allow adjusting the volume of individual tracks but did not implement this feature for reasons described in the next section. As shown in Figure 5.3, we can visually distinguish the currently muted tracks from the active ones.

[3]Official website of Figma: https://www.figma.com/

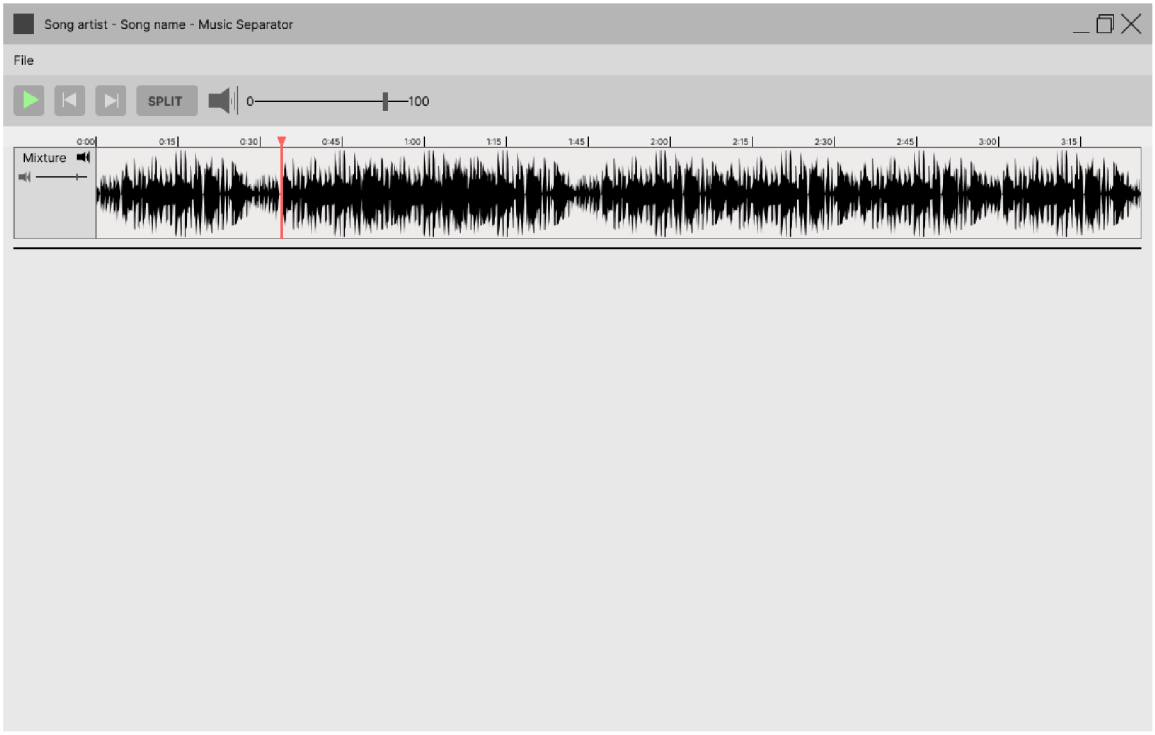


Figure 5.2: Design of application – before the song is split into individual instruments

5.2 Implementation

Upon launching the application, users are welcomed by the mostly empty screen, as showed in Figure 5.4. The toolbar is disabled at first. Therefore, the only possible actions are to load an audio file, adjust settings, or quit the application.

We organized the code for the application into separate `.py` files based on different aspects that make up the graphical interface. For instance, the class `Toolbar`, responsible for the display and functionality of the bar containing standard options related to playing music, is located in the file `toolbar.py`. Similarly, classes for the main window, or timeline displaying a song's current and total time, are placed in their separate files.

The application uses the neural network described in Chapter 4 to separate audio files into individual instruments. All source code files must be present in the directory with application implementation for the trained model to work. In addition, of course, all exported models we want to use must also be present. We can change their location in the settings dialog.

Settings

We implemented the settings menu as a custom dialog class `SettingsDialog` that inherits from `QDialog` class. `QDialog` is a widget appearing in front of the rest of the windows, and its job is to collect responses from the user. During initialization, the settings dialog loads the `separation_config.json` file and sets the default setting values accordingly. When settings are modified, current setting values related to audio separation are converted into a dictionary saved to the same `json` file. These settings include:

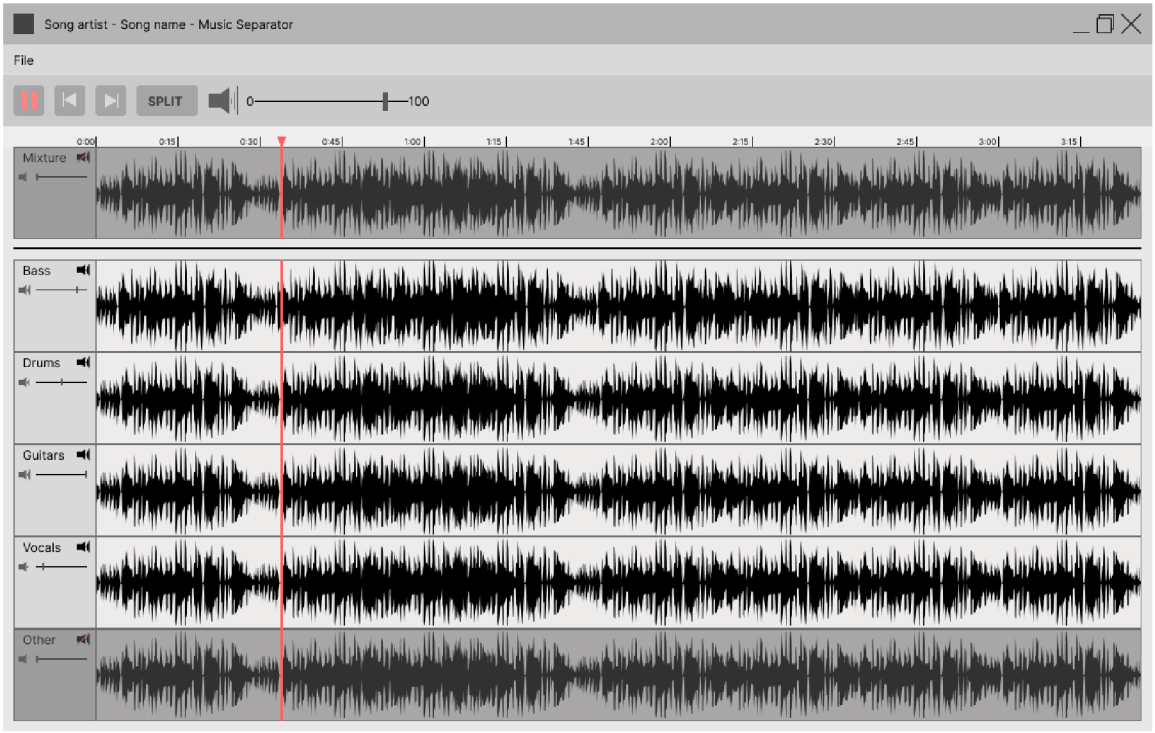


Figure 5.3: Design of application – after the song is split into individual instruments

1. Users can, as previously mentioned, choose a directory for the exported models.
2. We can edit the specific model signature to use in the application.
3. Users can switch between running the neural network on the CPU or GPU.

For this, we employ the `QComboBox` object, which shows the items added to it in a dropdown form. After clicking the OK button, the new settings are gathered from their respective forms and written back into the same `.json` configuration file. At the start of the separation process, we reread this file.

Opening file and loading song

Before the user can use the application as a music player or a learning tool, they must open an audio file by selecting the desired song in a standard file dialog. The option to choose a file is in the traditional menu bar. The application's main window, an instance of the `QMainWindow` class, provides an empty `QMenuBar` object that can be filled by whatever we choose and is accessible through `.menuBar()`. The action to open the file dialog connects to the menu bar. The advantage of using `QAction` class is that the command performs the same way regardless of whether the menu or a keyboard invokes it.

Triggering the open action runs the `choose_file()` method inside the main window. First, a `QFileDialog`, a file selection widget, is created. The application only supports loading `.mp3` audio files, the network's input. Selecting a file from the file dialog returns its absolute path, which must be first convert to a `QUrl` object before passing it to the media player. Figure 5.5 shows the application after a file is opened.

The core functionality the application needs to satisfy for practical use is playing audio. PyQt5 achieves this using the `QMediaPlayer` class that allows playing various media types.

Figure 5.4: Application after launch

Unfortunately, the media player can only play a single file at a time, which poses a significant obstacle after artificial intelligence separates a song. In addition, we want to mute or unmute certain tracks.

We employ the **PyDub** library which focuses on audio manipulation to solve this issue. When the neural network splits the song, we save the individual tracks produced by the *Demucs* model into a temporary directory initialized at program start-up using the `tempfile` module. Then, all separated tracks are overlayed, and the resulting mix is saved back into the temporary folder and fed into the media player. The user no longer listens to the original song, even when no tracks are muted. Instead, they listen to a sum of all sources. We do this because we assume that when the user splits a song, they do this to focus on only some aspects of the song, for instance, the bass.

After the player loads a song, a few more tasks must be completed before the user can finally listen. Firstly, we generate a waveform plot of the audio signal using `matplotlib`. After that, we only show the signal by turning off all axes and reducing the margins and paddings. Unfortunately, due to using a tight layout while saving the figure, we cannot directly save the graph with exactly specified dimensions. Because of this, we need to resize the image to the desired size using the `resize()` function from the PIL library.

The background image on the progress bar widget can be set using `setStylesheet()` to display a specific image, in our case, the generated waveform plot. The background image location is specified in the CSS format. Unfortunately, the image does not automatically resize to the widget, as could be done in CSS. For this reason, we need to resize the graphs manually.

The final graph is set as a progress bar's background inside our custom `Track` widget. The `Track` class implemented in `track.py` is responsible mainly for visualizing the position in played audio. It does this by showing the above graph and filling the progress bar with a semi-transparent color as the song keeps playing. The `Track` widget contains an info panel

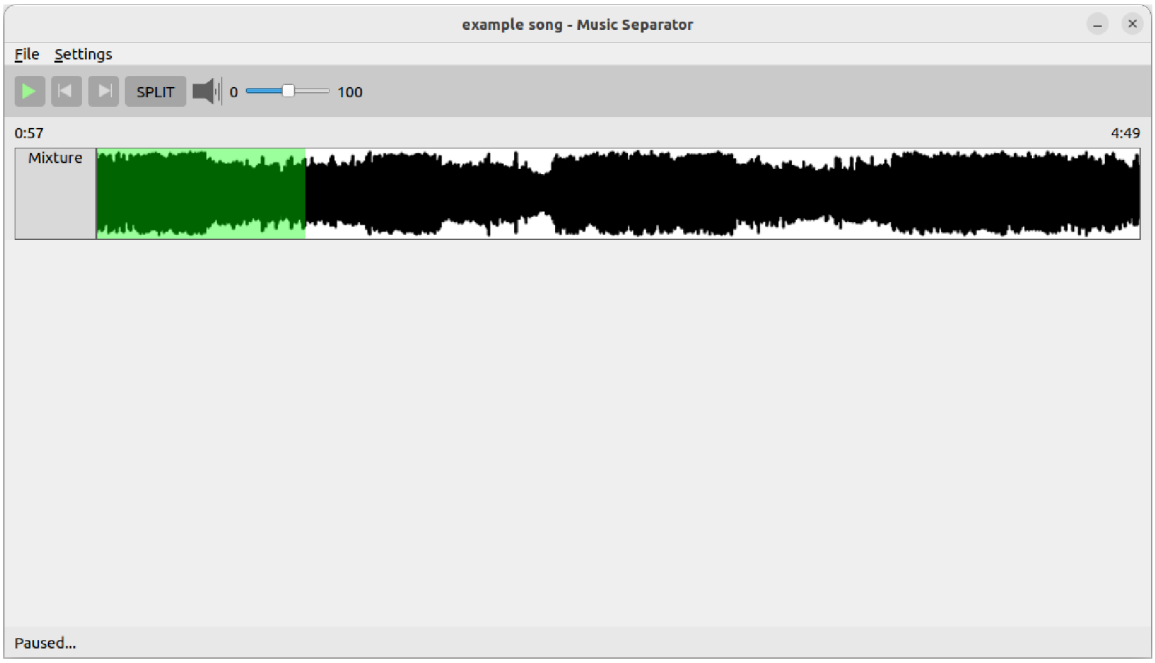


Figure 5.5: Application after opening an audio file

with the track name and mute button, and next to it, a progress bar. It uses a horizontal layout to separate the info panel from the progress bar. The info panel is a frame with a vertical layout because we wanted to include a mute track button at the bottom. This button is not present in the initial mixture the user loads, only in the separated tracks.

Playing song

The `Timeline` widget below the toolbar must update the total duration after the media player loads a song. It does this by accessing the duration property, which returns the time in milliseconds.

Users can toggle between playing a song and stopping it by clicking a button on the toolbar or pressing the space key. Both trigger the function `play_pause_song()`. The function changes the button's icon to reflect the current state and either pause the song using `pause()` or starts it up again with the `play()` method implemented in `QMediaPlayer`.

When the song is playing, it periodically triggers the signal `positionChanged`. This signal is connected to changing the timeline widget's current time and moving the progress bar in the `Track` widgets. We set the bar's range to the song's duration in milliseconds. Every time player's `positionChanged` signal is triggered, the progress bar's value property changes to reflect the player's position. This change has the effect that the overlay on the waveform graph accurately shows for how long the song has been playing.

Pressing the left or right arrows will rewind or forward the song by 5 seconds. We override the main window's `keyReleaseEvent` triggered on any key release to implement this feature. First, we acquire the pressed key from the `event.key()` and compare it to an array of keys. Then, we simply deduct an appropriate amount from the player's current position to rewind the song. Similarly, we implement the forwarding, except we add the amount.

Another standard features in the toolbar of a music player are the buttons to jump to the beginning or end of a song. The player position is set to 0 to move the song to start. Setting the player's position to duration will move a song to the end. Additionally, the user can choose the exact moment in the song where to jump to by clicking on the graph.

To achieve this, we created a custom `ProgressBar` class inheriting from the `QProgressBar` and overriding the `mousePressEvent`. Next, we acquire the x coordinate relative to the clicked widget from `event.x()` and convert it to a percentage by dividing it by the widget's width. Lastly, the main window moves the song to position according to the calculated percentage.

Separating song into components

The feature to split audio into individual components sets this program apart from regular music players. The split button is connected to the function `split_song()` by the signal `clicked`. Based on the feedback from initial testing, we added a special dialog to choose which instruments to keep as individual tracks while the rest would be combined into a single 'other' track. The `SplitInputDialog` indicates this with checkboxes, as is shown in Figure 5.6. It returns the selection as a dictionary of track names with the corresponding checkbox's state. Later we use this dictionary after the separation process is complete.

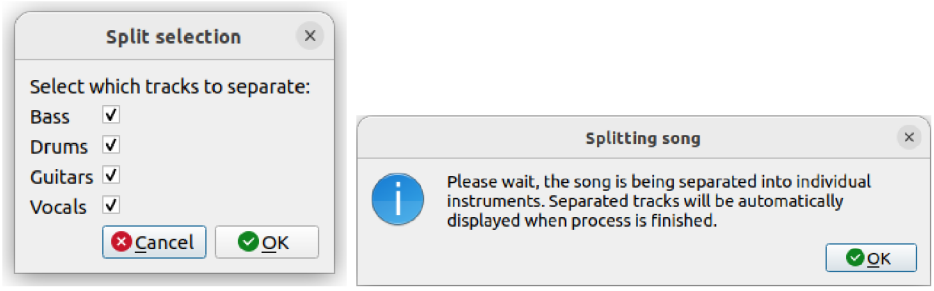


Figure 5.6: Dialog for the split selection and a warning dialog

We use multithreading, as described in Section 2.3, to prevent the application from becoming unresponsive and crashing during separation because it is a long-running task that blocks other events. A popup warning dialog warns the user that they might wait a while, depending on the song's length. However, they may still use the application to play music until the process finishes, which would not be possible without multithreading.

We task a worker with executing the function `split_song_thread()`. First, we call a slightly modified function `separate_track()` from the original Demucs source code inside. Results from separation are saved to the temporary folder. Settings used during the process are from the previously mentioned json configuration file. The thread then generates graphs for newly created tracks. Finally, the tracks not chosen during split selection are overlayed together along with the default 'other' output representing the rest of the components apart from the four main ones – the bass, drums, vocals, and guitars.

After the thread finishes the separation, it signals the worker to execute the method where we stop the audio, and the player's media content changes. This action triggers the timeline to change. The `Track` widget for the original loaded mixture is replaced with multiple new widgets for chosen separated tracks, as shown in Figure 5.7.

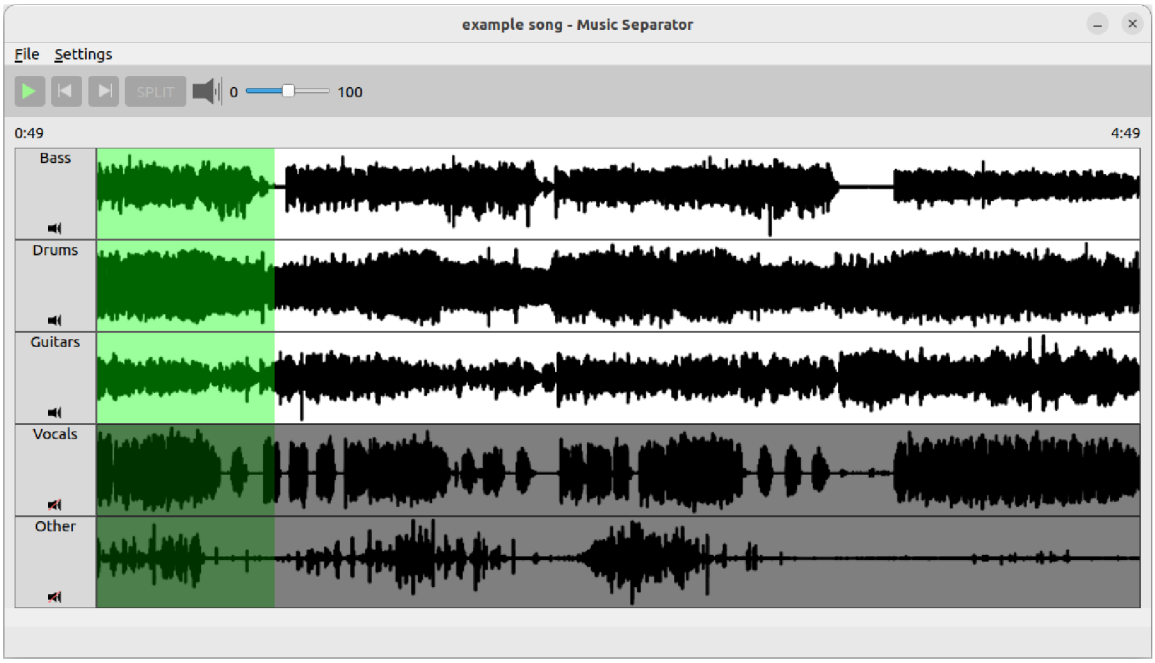


Figure 5.7: Application after separation and with muted tracks

Status bar

We added a simple status bar to the bottom of the screen that was not present in the original Figma design. We added it after the feedback from a tester who mentioned he would appreciate something to inform him of progress while waiting for separation to complete. It is implemented as a singular label that changes text during certain events. For example, the user is informed while the separation is in progress or when the waveform plots are generated, as those are the two most time-consuming tasks. The status bar is shown in Figure 5.5 at the bottom, informing the user that the song is paused.

Muting tracks

Another important feature is the ability to mute individual tracks until only one plays. Clicking the mute button on the selected track triggers the `toggle_track()` function. The status bar informs the user of what is happening. We temporarily block signals to the media player to prevent the sudden disappearance and reappearance of the progress bar that would otherwise occur due to incoming signals notifying the Track widget of a change in audio position when the media is swapped. The selected track is either added to or removed from the array containing all active tracks. Newly active tracks are overlayed together using the PyDub library, as was already described.

We visually darken a muted track to further indicate whether a track is active. For this purpose, we created a custom `Overlay` widget filled with semi-transparent color. Each `Track` is assigned its `Overlay` with the same dimensions and hides until the track is muted.

Exporting audio

The user can export the mix of active tracks using the *Export* action inside the main menu. We already saved the mixture in the temporary folder as `mixed.mp3`. So we simply copy it to a new location chosen through the file dialog.

Chapter 6

Evaluation

This chapter presents experiments testing various neural network structures on a partial dataset. Next, we introduce several trained models using complete datasets and evaluate them using objective and subjective methods. Then, we present results obtained through a questionnaire filled in by human respondents that tested our implemented application. Lastly, we discuss the limitations of both the trained neural networks and the application and propose possible plans for future development.

6.1 Neural network

6.1.1 Experimental setup

During the training, we used the Adam optimizer with a learning rate of 0.0003, a decay rate for the momentum term of 0.9, and a decay rate for the velocity term of 0.999. In addition, weight decay is set to 0. Batch size varied from 8 to 32 max, depending on the computational node used. The number of epochs done in a single training session also varied based on the number of CPUs requested. We do not modify the original network structure apart from changing the number of channels output by the convolutional layers, the number of recurrent layers, and the length of the input segment.

The models were trained on one of the three datasets: Remix dataset only, MedleyDB dataset only, or a combination of the two datasets. The Remix dataset consisted of 101 songs which totaled 7h 13m 5s of audio. MedleyDB comprised 53 examples with a total length of 3h 18m 32s. The combined dataset had 154 training songs and a length of 10h 31m 37s.

All models shared the same validation dataset, created by randomly selecting songs from the original Remix dataset. It contains 14 samples, up to 1h 3m 31s of audio. The test dataset used for the final objective evaluation was likewise created from the Remix dataset. Similarly, it had 14 samples and a length of 1h 9m 56s. We ensured that samples from the validation and test dataset did not appear in the training dataset.

6.1.2 Experiments with the neural network on a smaller dataset

We performed several different experiments with neural networks to determine the best hyperparameters. In these experiments, we examined the effects of, among others, batch size, loss function, or model depth. We used a smaller dataset to acquire results quickly. This smaller dataset was a subset of the regular remixes dataset from which we randomly

Table 6.1: Comparison of parameters and results of experiments on a partial dataset.

Experiment #	Input segment [s]	Channels	Depth	Loss	Valid loss	nSDR [dB]
1	6	24	6	L1	0.1647	**-1.157**
2	6	24	6	MSE	0.5873	-3.308
3	4	24	6	L1	0.1684	-1.664
4	4	4	2	L1	0.2040	-3.765

selected approximately 20% of training examples. The partial dataset for experiments had 20 training tracks and four validation songs. For all experiments, we only used the original Demucs architecture.

Experiments 1 and **2** studied the effects of changing a loss function, specifically L1 loss and the MSE loss, respectively. In both experiments, we reduced the number of channels the convolutional layers output and the number of recurrent layers, signifying the model's depth, to even load the model. We ran experiment 1 for 32 epochs and experiment 2 for 23 epochs as we reached a clear conclusion based on the achieved nSDR score.

Since we used various loss functions, we cannot look at the loss to determine which experiment performed better. However, a quick look at the SDR metric during validation shows that the neural network configuration used in experiment 1 performs better than the one in experiment 2. Experiment 2 achieved the best nSDR of **-3.308 dB**, while experiment 1 scored an nSDR of **-1.157 dB** (which can be seen in Table 6.1), possibly even better if we continued training. As a result, we trained all future models using L1 loss exclusively.

Experiment 3 investigated the effect of changing the input segment length. The rest of the model parameters remained the same as in experiment 1. Herefore, we could only experiment with reducing the segment length, as increasing the segment length above 6 seconds always resulted in a memory error. We trained this model for 28 epochs before we concluded that it performed worse than the model from experiment 1.

As we trained this model for approximately the same number of epochs as in experiment 1, we can compare the achieved nSDR scores. The best score from this experiment with an input length of 4 seconds was **-1.664 dB**, as is shown in Table 6.1. We conclude that shortening the input segment length adversely affects model performance.

Experiment 4 studied a very shallow version of the Demucs architecture and whether it could learn anything. We took the model hyperparameters from an example Makefile from the official repository[1]. This model was trained for 36 epochs before we determined it could only learn a little. The best nSDR score it attained was **-3.765 dB** (as is shown in Table 6.1).

Table 6.1 summarizes the model parameters for each experiment. Apart from the parameters, we also include the best-achieved loss scores and nSDR scores.

6.1.3 Models trained with complete datasets

After initial experiments with model architecture and hyperparameters, we trained models on more extensive datasets. Finally, we will use the best-performing model in a practical desktop application.

[1] Available at https://github.com/facebookresearch/demucs/blob/main/Makefile

Firstly, we trained three models with the same architecture using different training datasets. The first model utilized the full remix dataset instead of the small subset used in the previous experiments. The next model relied only on the MedleyDB dataset. Lastly, the final model used a combination of those two datasets. However, all the models share the validation dataset.

Compared to the previous experiments, these models were shallower. Therefore, the models' channels and depth were reduced to half to speed up training - reducing the channels to 12 from the initial 24 and depth to 3 instead of 6. Also, we wanted to see if an increased training dataset would even out the loss in model complexity. Furthermore, when we initially trained at Metacentrum using only a single GPU and CPU, more complex models could not finish even a single epoch. However, we could later train two deeper models thanks to increasing the number of CPUs used to help with mainly data loading.

We trained all three shallow models for a total of 70 epochs to be able to compare results more accurately. First, we present figures showing training and validation loss evolution and the nSDR metric computed during validation for each model. We especially highlight the boundary of 0 dB as the ratio between noise and actual signal in produced result shifts to contain more signal than the noise.

Figure 6.1 corresponds to the first model trained only on the remix dataset. Next, Figure 6.2 shows the model using only MedleyDB. Lastly, Figure 6.3 displays the model using a combination of datasets.

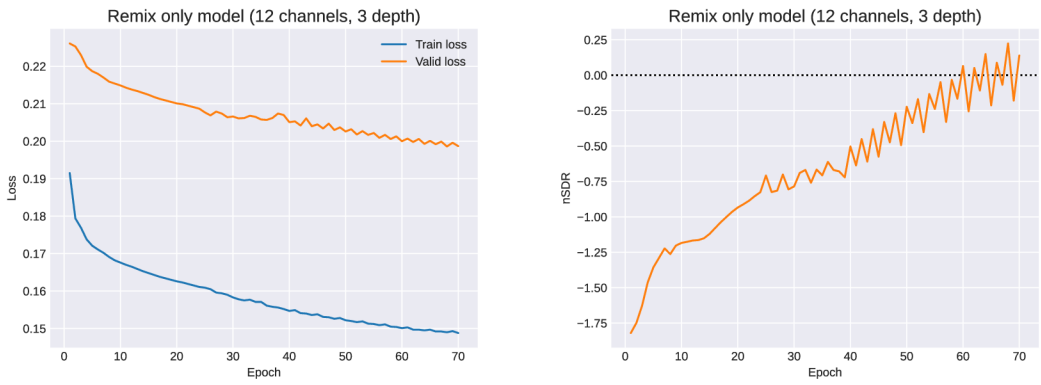


Figure 6.1: Loss and nSDR evolution for the Remix (12 channels, 3 depth) model.

Without any surprise, the model trained on the most extensive dataset achieved the best results out of the three, earning the best nSDR score on the validation dataset of **0.566 dB**. On the other hand, the MedleyDB model could not surpass the model using our remix dataset with the best nSDR of **0.035 dB**, most probably due to this dataset being shorter in total duration. The model trained on the remix dataset could, at best, achieve an nSDR of **0.224 dB**. The results are shown in Table 6.2. We selected the checkpoint during the last training session for the evaluation that yielded the best nSDR.

We must point out that none of these models achieve satisfactory performance judging from the SDR metric and are unsuitable for our desktop application. Because of that, we decided to train additional models.

Those deeper models using the combined dataset yield the best results. The first additional model doubled in size compared to previously trained ones. With 24 channels and a depth of 6, it copied the structure of experiment 1, the only difference being the size of the

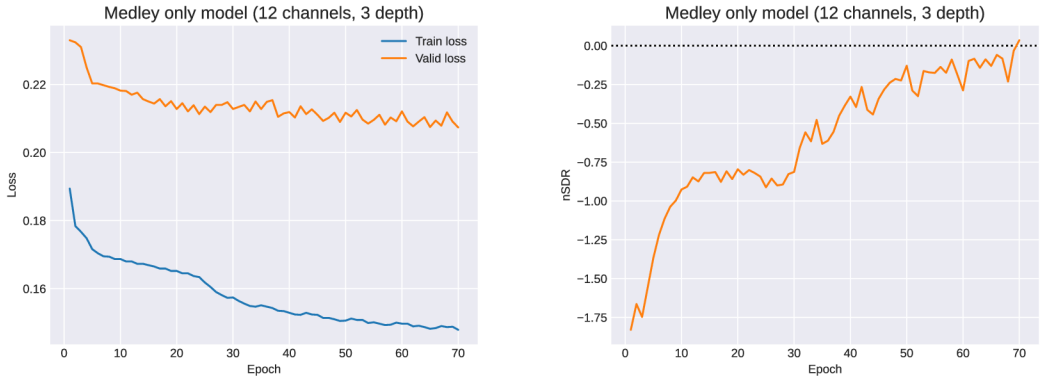


Figure 6.2: Loss and nSDR evolution for the Medley (12 channels, 3 depth) model.

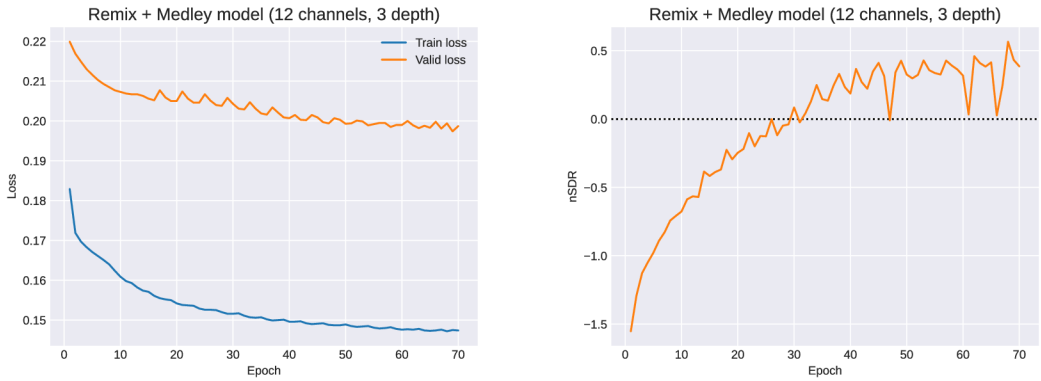


Figure 6.3: Loss and nSDR evolution for the Remix + Medley (12 channels, 3 depth) model.

training dataset. The second model had even more channels, a total of 28. Unfortunately, we could not further increase complexity, as this caused training models to run out of GPU memory after each epoch.

Figures 6.4 and 6.5 present the evolution of loss and nSDR during the training these two models, respectively. We did not limit ourselves to a specific number of epochs; instead, we trained models for as long as possible – both models were trained for 81 epochs.

From the very beginning, both models surpassed the previously trained models and quickly rose above the negative nSDR, something the shallower models could do only after tens of epochs. Figure 6.6 compares the validation loss across all models trained on the full dataset and Figure 6.7 the nSDR metric. Without any doubt, both 24-channel and 28-channel models performed significantly better. Examining these two more closely, we could argue that the 24-channel model slightly exceeds the 28-channel. It may be because this model requires even more data to learn correctly.

Table 6.2 describes the model parameters and the best-achieved validation loss and SDR.

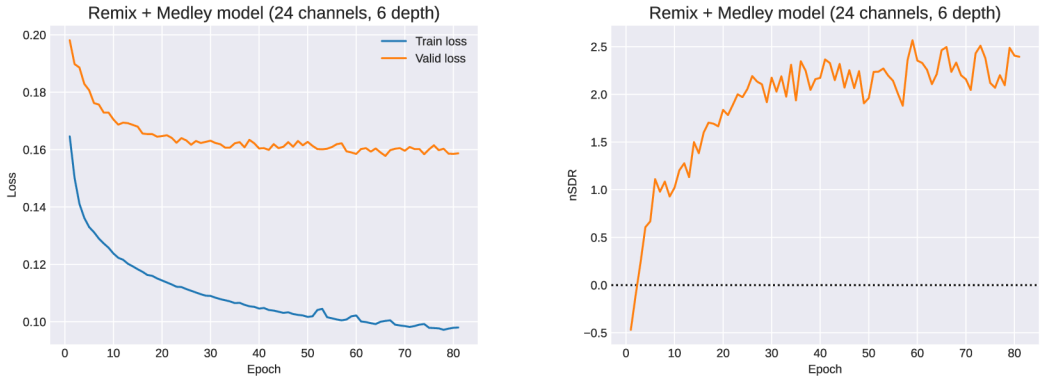


Figure 6.4: Loss and nSDR evolution for the Remix + Medley (24 channels, 6 depth) model.

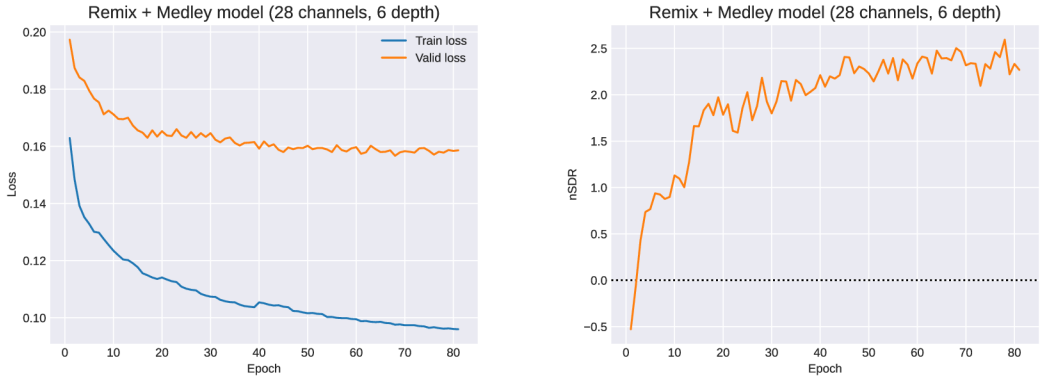


Figure 6.5: Loss and nSDR evolution for the Remix + Medley (28 channels, 6 depth) model.

6.1.4 Objective evaluation of models on a test dataset

We evaluated the five main models on a prepared test dataset. This test dataset was created by randomly selecting songs from initially downloaded remixes, making sure that these songs or their interprets do not appear in the train and validation datasets. We present four distinct metrics for better comparison - the nSDR, SIR, ISR, and SAR. These metrics are described in Chapter 3. First, table 6.3 shows the nSDR metric on the benchmark dataset for all models, table 6.4 shows the SIR, table 6.5 the ISR metric, and Table 6.6 shows the SAR results.

Unsurprisingly, the shallower models performed poorly, with the model trained only on MedleyDB being the worst overall. The two more complex models, the Remix + Medley (24/6) and (28/6), performed very similarly. While the 28-channel version outperformed the 24-channel one on nSDR and SIR metrics, it performed slightly worse on the ISR. In the SAR, there is no better model.

All models achieved better nSDR on the test dataset than on validation. We are unsure as to what is the cause for this. It is possible that overall the randomly selected songs for the test dataset were more similar to the training dataset.

Table 6.2: Comparison of parameters and results of training on a full dataset.

Dataset	Input segment [s]	Channels	Depth	Loss	Valid loss	nSDR [dB]
Remix	6	12	3	L1	0.1986	0.224
Medley	6	12	3	L1	0.2074	0.035
Remix + Medley	6	12	3	L1	0.1994	0.566
Remix + Medley	6	24	6	L1	0.1586	2.488
Remix + Medley	6	28	6	L1	**0.1578**	**2.593**

Table 6.3: nSDR results on test dataset.

Model	**Test nSDR [dB]** ↑					
	All	Bass	Drums	Guitars	Vocals	Other
Remix (12/3)	2.204	3.728	3.897	0.778	2.567	0.052
Medley (12/3)	1.811	3.112	3.498	0.927	1.504	0.013
Remix + Medley (12/3)	2.138	3.603	3.958	0.886	2.181	0.060
Remix + Medley (24/6)	4.035	6.610	7.230	1.732	4.554	0.051
Remix + Medley (28/6)	**4.137**	**6.669**	**7.481**	**1.791**	**4.662**	**0.082**

Table 6.4: SIR results on test dataset.

Model	**Test SIR [dB]** ↑					
	All	Bass	Drums	Guitars	Vocals	Other
Remix (12/3)	-14.563	-8.459	-10.851	-15.747	-12.815	-24.943
Medley (12/3)	-31.010	-7.737	-12.767	-17.693	-16.383	-31.010
Remix + Medley (12/3)	-14.903	-7.777	-14.089	-16.427	-12.703	-23.552
Remix + Medley (24/6)	-11.199	**-4.340**	-9.083	-12.599	-10.880	-19.094
Remix + Medley (28/6)	**-10.670**	-4.512	**-9.018**	**-12.382**	**-9.895**	**-17.545**

Table 6.5: ISR results on test dataset.

Model	**Test ISR [dB]** ↑					
	All	Bass	Drums	Guitars	Vocals	Other
Remix (12/3)	3.852	7.424	6.895	1.198	3.687	0.055
Medley (12/3)	2.902	4.541	6.391	1.901	1.666	0.012
Remix + Medley (12/3)	3.565	5.943	7.648	1.533	2.633	0.070
Remix + Medley (24/6)	**7.042**	**11.429**	**12.325**	**4.682**	6.518	-0.647
Remix + Medley (28/6)	7.036	11.186	11.754	4.643	**7.079**	**0.518**

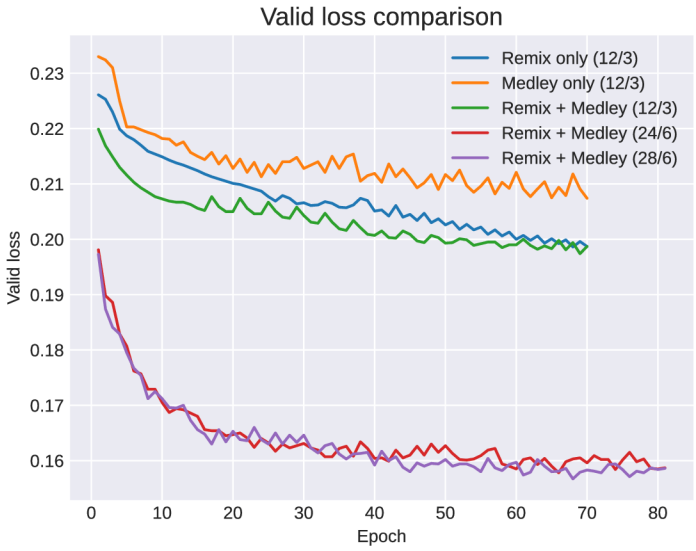


Figure 6.6: Comparison of validation loss across all trained models.

Table 6.6: SAR results on testing dataset.

Model	**Test SAR [dB] ↑**					
	All	Bass	Drums	Guitars	Vocals	Other
Remix (12/3)	0.268	0.598	0.179	0.210	**0.315**	**0.039**
Medley (12/3)	0.210	0.610	0.154	0.149	0.050	-0.002
Remix + Medley (12/3)	0.212	0.572	0.089	0.212	0.122	0.062
Remix + Medley (24/6)	0.372	**1.070**	**0.395**	0.245	0.147	0.003
Remix + Medley (28/6)	**0.379**	0.977	0.376	**0.310**	0.226	0.006

6.1.5 Subjective evaluation of models using human respondents

At first, we attempt to evaluate the trained models by listening to separated tracks. We randomly selected 3 songs from the test dataset for this purpose. The three smaller models fare worse as there is much noise in the records, especially in the guitars, vocals, and the other track. The bass track sounds the best, followed by the drum track. However, even there is still a bleed from the rest.

Examining the separated tracks, the difference between the two deeper models takes much work to notice. Again, both perform best at separating bass and drums. While the vocals and guitars are separated better than with the smaller models, noise and unwanted artifacts remain. Vocals tend to bleed into the guitar track and vice versa, although slightly less.

We created a questionnaire to determine which model performs the best according to independent evaluators:

Figure 6.7: Comparison of validation nSDR across all trained models.

	Rating				
Model	Bass	Drums	Guitars	Vocals	All
Remix (12/3)	4	4	3	3	3
Medley (12/3)	5	5	5	5	5
Remix + Medley (12/3)	3	3	4	4	4
Remix + Medley (24/6)	2	2	2	2	2
Remix + Medley (28/6)	1	1	1	1	1

Table 6.7: Results of users evaluating the models.

1. We select and separate 20-second long excerpts from three tracks and ask respondents to listen to all of them. The respondents do not know which model separated the track.
2. For each instrument category, namely the bass, drums, guitars, and vocals, we ask them to order tracks from what they believe is the worst separated to the best.
3. We average out their responses to get the final results.

We present the results in Table 6.7. In total, 11 people filled out the questionnaire. The models are ordered in each category, from 5 being the worst to 1 representing the best.

The previous objective testing showed that the most complex 28-channel model achieved the best result in several metrics. First, the respondents considered this model the best across all categories. Next, they felt that the 24-channel model was the second best.

The worst of all models was the Medley (12/3) model, which we anticipated based on previous tests. The 12-channel model trained on only the Remix dataset overall received better scores than the similarly complex model trained on a larger dataset. It only received worse scores on bass and drums separation.

We are aware that three short song segments are not enough for a thorough evaluation. However, we had to consider the time necessary to listen to all separated excepts by five

Rating	# of votes 1	2	3	4	5	Average
Design	0	0	1	6	6	4.38
Navigation clarity	0	0	0	3	10	4.77
Responsivity	0	0	6	6	1	3.62
Likelihood of using the application again	0	1	2	6	4	4.00

Table 6.8: Responses of users testing the application.

different models across multiple songs. If the respondent can order the models with repeat listening, the listening portion of the questionnaire would take 20 minutes. Unfortunately, we could not find many respondents that would commit to a questionnaire even longer than that.

6.2 Application testing

We subjected the created application to testing by human volunteers. Afterward, they would answer the questionnaire about their experience. The full transcript of the questionnaire is in Appendix C. Due to the application's straightforward nature, there was no specific use case for the user to perform. Instead, they were encouraged to use the application as much as possible and try the separation process on at least one provided song.

In the questionnaire, the respondents had to rate the application on a scale from 1 being the worst to 5 being the best in several categories:

- overall design
- responsivity
- navigation clarity
- possibility of further usage

We also inquired about their preferred choice of operating system to determine if we should port the application in the future. Apart from their likes and dislikes about the application, we also asked for suggestions about potential new functionality.

In total, **13** users tested the application. Table 6.8 shows the results from the part of the questionnaire where the respondents graded various aspects of the application. Overall, the application design received a positive rating. In addition, the testers deemed the application to be easy to navigate, which was to be expected. On the other hand, the responsivity received the worst ratings, with an average of *3.56*, making it the most crucial aspect of the application to improve in the future.

Most of the users indicated they would likely use the application again in the future. With most users indicating that they use the Windows operating system, attempting to port to other systems would be beneficial.

Users praised the simplicity of the application, its ease of use, and the feature to split songs into separate instruments. However, during testing, they disliked the occasional minor bugs and long waiting time for the separation process to finish. In addition, some complained about the lack of ability to load multiple songs into different tabs.

The questionnaire helped suggesting new features to add in the future. One of them is the ability to loop parts of the song. Another addition would be the option to change the volume of individual tracks. The rest of the suggestions would be easier to implement, such as renaming the tracks or a better progress indicator.

Lastly, the respondents indicated that they would use the program again in guitar-related tasks. For example, learning guitar directly from the application or just separating the guitar sound.

6.3 Limitations and plans

Based on the objective evaluation of trained neural networks, there is still room for improvement. The models achieved results well below those of existing state-of-the-art methods. However, these models do not separate guitar sounds, unlike our models. Therefore it is expected they would perform at least slightly worse. Nonetheless, they serve well enough to demonstrate integrating a neural network into a practical application. Our models would benefit from additional training time. Fortunately, it is simple to swap models in the application if needed.

The next step in future development would be to use different architecture for the neural network, namely the Hybrid Demucs. Unfortunately, we could not do this in this thesis due to time constraints. Nevertheless, using this architecture should help improving guitar sound separation performance.

User testing of the application concluded that there are possible ways to improve, ranging from new functionality to improvement in responsivity. During the planning stages, we intended to include a looping mechanism allowing users to replay a small portion of the song. However, due to the complexity of the task and time constraints, we decided to abandon this feature to focus on improving base functionality.

Another feature users would welcome, the volume control over individual tracks, is impossible to add to the current application because PyQt does not support playing multiple audio files at once. If we were to change the volume on, for instance, the bass track, we would have to overlay all tracks together, which takes a couple of seconds. This additional delay would detract from the user experience.

In the future, we intend to continue working on the application. In addition, we are considering reworking it thoroughly using a different GUI framework that would provide better audio control.

Chapter 7

Conclusion

In conclusion, this thesis presented the development of an application for guitar sound separation which utilizes a neural network.

Work involved the creation of a custom dataset from remixes of popular songs and adjusting the existing MedleyDB dataset.

Next, we adapted the Demucs model to include additional output – the guitar sound. We trained and evaluated five distinct models in MetaCentrum. These models varied in training dataset used and architecture – namely, the number of feature maps produced by each convolutional layer and the number of recurrent layers in the model.

All models were evaluated on the test portion of the prepared dataset, and we present several metrics for objective evaluation. None of the trained models achieve results comparable to state-of-the-art methods but are usable in a practical application. This is mainly because these methods do not separate guitar sounds. Furthermore, the models were subjectively assessed by human listeners to rate their performance relative to each other.

We implemented an application for separating songs into five components – the bass, drums, guitars, vocals, and rest. In addition, the application serves as a music player and an educational tool for learning musical instruments. The application was evaluated through a questionnaire. The findings also suggest that the application has the potential to be a valuable tool for musicians and sound engineers.

Future research could explore the new architectures of neural networks, namely swapping the current Demucs architecture for the Hybrid Demucs. In addition, the application would benefit from additional functionality suggested by respondents, such as the looping function or more robust audio controls.

Bibliography

[1] Anwar, A. *What is Transposed Convolutional Layer?* March 2020. [cit. 2022-09-11]. Available at: https://towardsdatascience.com/what-is-transposed-convolutional-layer-40e5e6e31c11.

[2] Bittner, R., Wilkins, J., Yip, H. and Bello, J. MedleyDB 2.0: New Data and a System for Sustainable Data Collection. In:. International Conference on Music Information Retrieval (ISMIR-16), 2016. DOI: 10.5281/zenodo.1715175.

[3] Bittner, R., Salamon, J., Tierney, M., Mauch, M., Cannam, C. et al. MedleyDB: A Multitrack Dataset for Annotation-Intensive MIR Research. In:. October 2014. DOI: 10.5281/zenodo.1649325.

[4] Dauphin, Y. N., Fan, A., Auli, M. and Grangier, D. Language Modeling with Gated Convolutional Networks. In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. JMLR.org, 2017, p. 933–941. ICML'17.

[5] Défossez, A., Usunier, N., Bottou, L. and Bach, F. Music Source Separation in the Waveform Domain. *ArXiv preprint arXiv:1911.13254*. 2019.

[6] Défossez, A. Hybrid Spectrogram and Waveform Source Separation. In: *Proceedings of the ISMIR 2021 Workshop on Music Source Separation*. 2021.

[7] Défossez, A., Zeghidour, N., Usunier, N., Bottou, L. and Bach, F. SING: Symbol-to-Instrument Neural Generator. In: *Conference on Neural Information Processing Systems (NIPS)*. 2018.

[8] Fitzpatrick, M. *Multithreading PyQt5 Applications with QThreadPool*. 2022. [cit. 2023-18-04]. Available at: https://www.pythonguis.com/tutorials/multithreading-pyqt-applications-qthreadpool/.

[9] Fitzpatrick, M. *PyQt5 Signals, Slots & Events*. 2023. [cit. 2023-18-04]. Available at: https://www.pythonguis.com/tutorials/pyqt-signals-slots-events/.

[10] Goodfellow, I. *Deep learning*. Cambridge, MA: MIT press, 2016. Adaptive computation and machine learning series. ISBN 978-0-262-03561-3.

[11] He, K., Zhang, X., Ren, S. and Sun, J. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. 2015. arXiv:1502.01852.

[12] Hendrycks, D. and Gimpel, K. Gaussian Error Linear Units (GELUs). 2020. arXiv:1606.08415.

[13] Hochreiter, S. and Schmidhuber, J. Long Short-Term Memory. *Neural computation*. One Rogers Street, Cambridge, MA 02142-1209, USA: MIT Press. 1997, vol. 9, no. 8, p. 1735–1780. ISSN 0899-7667.

[14] Kamp, P. *What is a mean opinion score (MOS)?* [cit. 2022-29-10]. Available at: https://www.twilio.com/docs/glossary/what-is-mean-opinion-score-mos.

[15] Kelleher, J. *Deep Learning*. MIT Press, 2019. MIT Press Essential Knowledge series. ISBN 9780262537551. Available at: https://books.google.sk/books?id=1wICwQEACAAJ.

[16] Li, J. *L1-L2 Norm and regularization Comparisons*. June 2020. [cit. 2022-05-10]. Available at: https://medium.com/@jingli_57859/l1-l2-norm-and-regularization-comparisons-a0f45065593d.

[17] Luo, Y. and Mesgarani, N. Conv-TasNet: Surpassing Ideal Time-Frequency Magnitude Masking for Speech Separation. *IEEE/ACM transactions on audio, speech, and language processing*. United States: IEEE Press. 2019, vol. 27, no. 8, p. 1256–1266. ISSN 2329-9290.

[18] Manilow, E., Seetharman, P. and Salamon, J. *Open Source Tools & Data for Music Source Separation*. October 2020. Available at: https://source-separation.github.io/tutorial.

[19] Manilow, E., Wichern, G., Seetharaman, P. and Le Roux, J. Cutting Music Source Separation Some Slakh: A Dataset to Study the Impact of Training Data Quality and Quantity. In: IEEE. *Proc. IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*. 2019.

[20] MetaCentrum. *O MetaCentru*. 2021. [cit. 2023-15-03]. Available at: https://metavo.metacentrum.cz/cs/about/index.html.

[21] Mitsufuji, Y., Fabbro, G., Uhlich, S., Stöter, F.-R., Défossez, A. et al. Music Demixing Challenge 2021. *Frontiers in Signal Processing*. 2022, vol. 1. ISSN 2673-8198.

[22] O'Shea, K. and Nash, R. *An Introduction to Convolutional Neural Networks*. 2015. ArXiv:1511.08458.

[23] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J. et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. 2019. arXiv:1912.01703.

[24] Rafii, Z., Liutkus, A., Stöter, F.-R., Mimilakis, S. I. and Bittner, R. *The MUSDB18 Corpus for Music Separation*. December 2017. DOI: 10.5281/zenodo.1117372.

[25] Rafii, Z., Liutkus, A., Stöter, F.-R., Mimilakis, S. I. and Bittner, R. *MUSDB18-HQ - an Uncompressed Version of MUSDB18*. December 2019. DOI: 10.5281/zenodo.3338373.

[26] Ronneberger, O., Fischer, P. and Brox, T. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. ArXiv:1505.04597.

[27] Rouard, S., Massa, F. and Défossez, A. Hybrid Transformers for Music Source Separation. In: *ICASSP 23*. 2022.

[28] Scarrott, B. *MP3, AAC, WAV, FLAC: all the audio file formats explained*. July 2022. [cit. 2020-28-10]. Available at: https://www.whathifi.com/advice/mp3-aac-wav-flac-all-the-audio-file-formats-explained.

[29] Singh, K. *Complete tutorial on how to use Hydra in Machine Learning projects*. March 2021. [cit. 2022-20-11]. Available at: https://towardsdatascience.com/complete-tutorial-on-how-to-use-hydra-in-machine-learning-projects-1c00efcc5b9b.

[30] Sivaram, T. *Skip connections: All you need to know about skip connections*. Aug 2021. [cit. 2022-08-11]. Available at: https://www.analyticsvidhya.com/blog/2021/08/all-you-need-to-know-about-skip-connections/.

[31] Stoller, D., Ewert, S. and Dixon, S. Wave-U-Net: A Multi-Scale Neural Network for End-to-End Audio Source Separation. *19th International Society for Music Information Retrieval Conference (ISMIR 2018)*. 2018.

[32] Takahashi, N., Goswami, N. and Mitsufuji, Y. MMDenseLSTM: An Efficient Combination of Convolutional and Recurrent Neural Networks for Audio Source Separation. 2018. arXiv:1805.02410.

[33] Takahashi, N. and Mitsufuji, Y. D3Net: Densely Connected Multidilated DenseNet for Music Source Separation. 2020. arXiv:2010.01733.

[34] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L. et al. Attention is All you Need. In: Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R. et al., ed. *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2017, vol. 30. Available at: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[35] Vincent, E., Gribonval, R. and Fevotte, C. Performance Measurement in Blind Audio Source Separation. Piscataway, NJ: IEEE. 2006, vol. 14, no. 4, p. 1462–1469. ISSN 1558-7916.

[36] Vincent, E., Sawada, H., Bofill, P., Makino, S. and Rosca, J. P. First Stereo Audio Source Separation Evaluation Campaign: Data, Algorithms and Results. In: *Independent Component Analysis and Signal Separation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, p. 552–559. Lecture Notes in Computer Science. ISBN 9783540744931.

[37] Willman, J. M. *Beginning PyQt: a Hands-on Approach to GUI Programming*. Hampton, USA: Apress, 2020. 449 p. ISBN 978-1-4842-5856-9. Available at: https://doi.org/10.1007/978-1-4842-5857-6.

[38] Yadan, O. *Hydra - A framework for elegantly configuring complex applications* [Github]. 2019. Available at: https://github.com/facebookresearch/hydra.

[39] Yang, Y.-Y., Hira, M., Ni, Z., Chourdia, A., Astafurov, A. et al. TorchAudio: Building Blocks for Audio and Speech Processing. 2021. arXiv:2110.15018.

[40] Zvornicanin, E. *Differences Between Bidirectional and Unidirectional LSTM.* November 2022. [cit. 2022-09-11]. Available at: https://www.baeldung.com/cs/bidirectional-vs-unidirectional-lstm.

Appendix A

Contents of the included storage media

The included medium contains:

- **documentation** – directory with thesis text source code and `pdf`
- **demucs** – directory with code and datasets used for training the neural networks
- **musicSeparationGUI** – directory with code and instructions on how to launch the practical application
- **examples** – directory with examples of separated songs
- **music_separation_training.xlsx** – an Excel file containing more information on the training process and results

Appendix B

Installing the application

The application was primarily developed and tested on Ubuntu 22.04.

Prerequisites

- python >=3.8
- pip
- virtualenv
- ffmpeg

If any of them are not present in your system, install them using the following commands:

```
sudo apt install python3
sudo apt install python3-pip
sudo apt install python3-venv
sudo apt install ffmpeg
```

Installation

Navigate to the directory with the application code and run the `install.sh` or execute the following commands in given order:

```
python3 -m venv musicSeparatorEnv
source musicSeparatorEnv/bin/activate
python3 -m pip install -r requirements_app.txt
```

Launch

To launch the application, run the `run.sh` or:

```
source musicSeparatorEnv/bin/activate
python3 main.py
```

On certain versions of Ubuntu, there are known issues with playing media using PyQt, which can be fixed by applying the following:

```
sudo apt-get install libqt5multimedia5-plugins
```

Appendix C

Application testing questionnaire

Your task is to open the prepared example song in the application and separate the guitar track. Then, mute all the other channels except for the guitar. To be sure, try listening to it. Finally, export the guitar sound. After you are finished, please answer the following questions:

On a scale from 1 to 5 (1 being the worst and 5 being the best) rate:

1. Application design
   1:☐ 2:☐ 3:☐ 4:☐ 5:☐
2. Navigating the application without explaining the controls
   1:☐ 2:☐ 3:☐ 4:☐ 5:☐
3. Application responsivity
   1:☐ 2:☐ 3:☐ 4:☐ 5:☐
4. How likely you are to use the application again outside of this testing
   1:☐ 2:☐ 3:☐ 4:☐ 5:☐

Answer the next set of questions by writing one or more sentences:

1. What operating system do you usually use?

   .....

2. What did you like the best about the application?

   .....

3. What did you dislike the most about the application?

   .....

4. Is there any other functionality you would welcome or feel needs to be added?

   .....

5. Have you experienced any bugs during testing?

   .....

6. If you previously answered that you would use the application again, what would you use it for?

   .....