Neuroment - Instrument Detection using Convolutional Neural Networks

Audio Engineering Project

Lorenz Häusler, Lukas Ignaz Maier
Supervisor: Univ.Prof. DI Dr. Alois Sontacchi

Graz, April 14, 2022

Institute of Electronic Music and Acoustics

Abstract

Source separation, in a musical context, is the task to retrieve the essential elements (in this case instrument tracks) of a given audio signal. State-of-the-art implementations often provide unsatisfying results containing respectable amounts of audible glitches and artifacts. In the proposed algorithm, Artificial Intelligence is used to determine activation functions of single instruments in a mix, which can be helpful in separation tasks.

The basis of the algorithm is a trained Convolutional Neural Network (CNN) which analyzes frames of an audio recording. These frames are transformed to frequency domain, using various methods with variable resolution. The transformed frames then are used as input features for the CNN.

The output of the network consists of frames of the same length but containing the separated envelopes of each instrument over time. The data for training the CNN is generated from a set of solo instrument audio tracks, which are subsequently mixed in various combinations to ensure flexible training with fixed envelopes.

Contents

1 Introduction

In recent decades neural networks have seen a considerable increase in popularity for their flexibility and use in a wide range of applications. One of the main tasks these networks are being designed for is pattern recognition and data interpretation. A prominent challenge in audio signal processing, which falls under this category, is source separation. Its goal is to extract a number different elements, often instruments or voices, from an audio signal containing mixes of these elements.

A big obstacle in source separation is to obtain reasonable activations. Activations indicate how much energy an element, here more specifically an instrument, possesses at a certain time instance within the audio stream. Instruments often sound similar when played in different ranges. For example, a bass being played in an unnaturally high range may sound very similar to a guitar, even for the experienced listener. Naturally, algorithms used for source separation have to prevent these confusions in order to produce a clean separation. Preprocessing the training data into a format which is interpretable for the structure of a neural network is therefore essential.

In the present work a framework for prediction with a Convolutional Neural Network (CNN) has been implemented with high flexibility regarding the amount of training data, methods of feature extraction and means of configuration. The end user setup of the framework is streamlined by an automatic dependency installation and the possibility of using either a CPU or (if available) a GPU for training. The quality of separation is also evaluated automatically during training and can be observed if necessary.

In chapter 2 the theoretical basics are explained. First neural networks are introduced, followed by the implemented feature extraction methods.

Chapter 3 depicts the structure of the whole framework. It considers the program configuration, data generation and preprocessing, and the training of our model. Finally, the algorithm for the prediction, which generates the results and its evaluation, is introduced. In this chapter also the programming environment, the setup as well as its dependencies and libraries are explained.

The implementation is depicted in chapter 4. Data processing, data selection, file management, tagging as well as the feature extraction are described here in detail. The training algorithm with its data flow and adaptive parameters are visualized here as well.

The results are shown and discussed in chapter 5. The training process is evaluated first. Then the output of the network is put to the test using simple plots and more comprehensive methods. Finally various examples of the separation performance are presented.

The underlying work finishes with a conclusion in chapter 6.

2 Theoretical basics

2.1 Artificial neural networks

In recent history artificial neural networks have proven to be useful for classification tasks in various applications. For example they are renowned for image recognition tasks where they achieved astounding results after sufficient training. In audio applications the favorable properties of neural networks can be taken advantage of. An audio stream can be converted into a (magnitude) spectrogram via a frequency transformation. The resulting spectrogram possesses similar properties to a monochromic image which allows to feed it into a neural network.

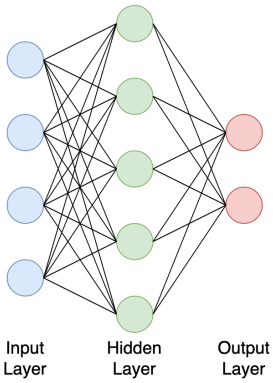


Figure 1: Structure of a sample artificial neural network (ANN) with three fully connected layers (the input layer, one hidden layer and an output layer).

A special form of artificial neural networks is a convolutional neural network (CNN). Instead of flat layers of neurons which are densely connected (see figure 1) 2-dimensional kernels are convolved with their input data. This allows for better recognition in the two-dimensional image domain. Earlier convolutional layers in the network usually detect simple shapes like lines and borders, while later/deeper convolutional layers allow for detection of more complex patterns.

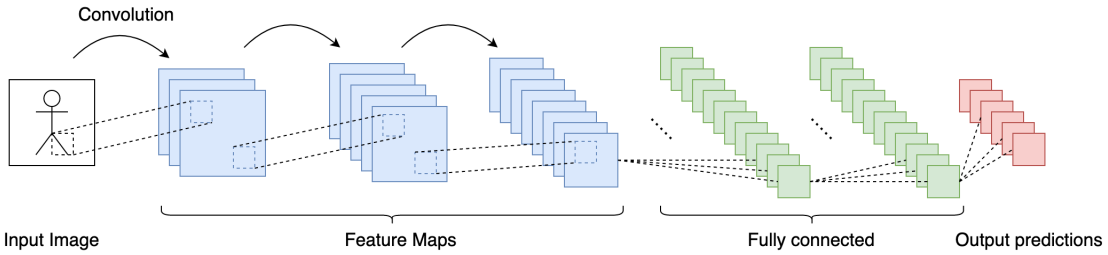


Figure 2: Structure of an example CNN.

2.2 Constant-Q transform (CQT)

The CQT (short for Constant-Q transform) transforms data from time to frequency domain. While this is something other frequency transformations also achieve, its difference lies in the equal spacing of frequency bands.

A quite handy way of explaining the CQT is to start with its parent, the DFT (Discrete Fourier Transform) [1]. The DFT transforms a discrete time signal into a discrete frequency signal. One variant of it, the FFT (Fast-Fourier Transform) algorithm

$$X[k] = \frac{1}{N} \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi k n}{N}} \quad k = 0, ..., N-1 \quad , \tag{1}$$

is one of the most commonly used algorithms in signal processing. N time signal bins $x[n]$ are transformed to N frequency bins $X[k]$. Due to its symmetry properties, the DFT's spectrum is mirrored around $\frac{N}{2} + 1$. To avoid redundancy, usually the mirrored part above this threshold is neglected during computations.

An important property of the DFT is an equal spacing of frequency bins. The spacing between bins is given by $\frac{f_S}{N}$, with f_S being the sampling frequency. To avoid the effect of aliasing, the audio signal may not contain frequencies above the Nyquist frequency of $\frac{f_S}{2}$ [2]. While an equal spacing is computationally easy to handle, it has drawbacks for the analysis of audio signals, specifically ones containing music.

Fundamental frequencies in music are distributed *logarithmically* over the frequency domain. To explain this one may assume the interval of an octave, which is nothing else than two frequencies being in a ratio of $2 : 1$ to each other. In a musical sense, an octave always comprises of 12 semitones (a twelfth of on octave), no matter how high or low the the fundamental frequency. Following, higher semitone intervals have a higher bandwidth $f_\Delta = f_{high} - f_{low}$, where f_{high} is the fundamental frequency of the higher tone and f_{low} the fundamental frequency of the lower tone. Depending on the use case it may therefore be rather unhandy to describe music signals with the equal frequency spacing of the DFT.

The CQT offers help. In its domain there is a minimal frequency in the transformation, named f_{min}. A usual value for this is 27.5Hz. Furthermore the CQT defines a fixed number of CQT bins k per octave. The number of bins per octave is defined by the value B. The formula for the frequency f_k of a bin with index k is

$$f_k = f_{min} 2^{\frac{k}{B}} \quad . \tag{2}$$

By using the bandwidth f_Δ between two tones we can define the Q factor Q as

$$Q = \frac{f_k}{f_\Delta} \quad . \tag{3}$$

Now in the traditional DFT domain the bandwidth f_Δ between two tones is fixed, which means that the Q factor increases with higher frequencies. As already mentioned this is not intuitive for musical intervals, where a semitone interval always sounds like a semitone interval, regardless of the actual Q factor.

The CQT offers a solution to this problem. In the simplest sense it keeps the Q-factor constant. It does so by essentially using a bank of K band-pass filters that share a common Q-factor, resulting in a logarithmically spaced frequency scale [3]. The CQT formula

$$X[k] = \frac{1}{N[k]} \sum_{n=0}^{N[k]-1} W[k,n]x[n]e^{-j\frac{2\pi kQn}{N[k]}} \tag{4}$$

shows two main differences to the DFT formula. First, the Q-factor is in the exponential term, which makes sense considering the aforementioned logarithmic spacing. Second, the window length $N[k]$ now depends on the frequency bin k. With f_s being the sampling frequency it can be computed per bin via

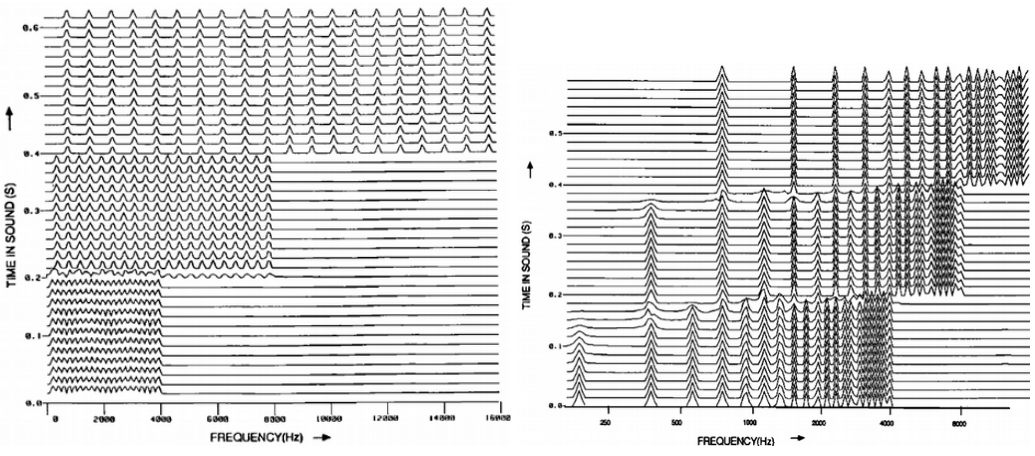
$$N[k] = Q\frac{f_s}{f_k} \tag{5}$$

The term $W[k,n]$ represents a window function, often a Hann window. In contrast to the DFT transform this window function now also depends on two dimensions (time and frequency) instead of only one dimension.

The frequency f_k of each bin depends on the musical interval that is chosen to be between two frequency bins. A usual interval would be a semitone. Considering that there are 12 semitones in each octave we could, as an example, set $B = 12$. This leads to a frequency f_k of each bin k being expressed as

$$f_k = f_{min}2^{\frac{k}{12}} \quad . \tag{6}$$

Now figure 3 further points out the differences between the DFT and the CQT domain.



(a) FFT. (b) CQT.

Figure 3: Comparison of FFT and CQT using 3 complex sounds with fundamentals G1 (196 Hz), G4 (392 Hz) and G5 (784 Hz). Each sound has 20 harmonics with equal amplitude. [4]

In the figure we see 20 harmonics on a frequency scale. Each of the harmonics is equally spaced. A linear frequency scale in this plot results in equal bin spacing for the FFT and in a decreasing bin spacing for the CQT. Thus a logarithmic scale leads to a decreasing spacing for FFT bins and a continuous spacing for CQT bins. Considering that the human ear works logarithmically this feels natural in a musical sense.

2.3 Logarithmic Mel-Spectrogram

Another frequency transformation approach similar to the CQT is the so called mel-spectrogram. Here a filter bank is used to achieve a logarithmic spacing in the frequency axis. The basis for the spectral transformation again is a STFT which is then multiplied with the mel filter bank in order to produce the mel-spectrogram. The mel filter bank with which the STFT spectrum is multiplied consists of triangular windows with varying window lengths that depend on the frequency. Some implementations also use normalization to avoid skewing the energy density of the input frames.

In figure 4 an exemplary filterbank is shown.

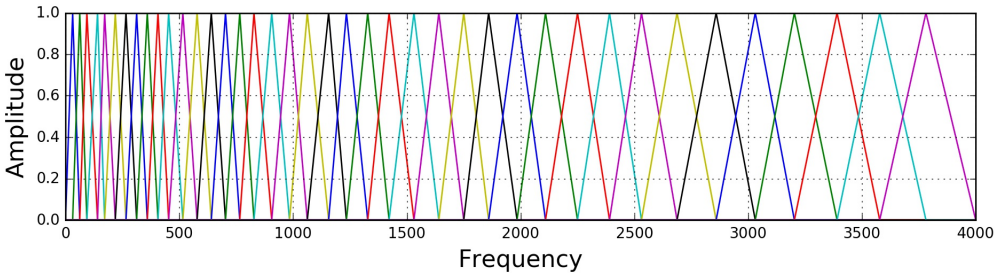


Figure 4: Example of a mel filterbank.

The most important characteristic of a mel-spectrogram is its size. While it is theoretically possible to make it bigger than the STFT spectrogram it usually offers a significantly smaller spectrum than the STFT spectrum without losing much of the information.

3 Framework design

3.1 Programming language and libraries

The code base in this work has been implemented in Python 3.9. Python is a programming language which offers a lot of well-developed libraries for data handling, music information retrieval and visualization as well as a relatively good readability. It is also one of the most commonly used languages for data science which benefits code accessibility for many users [5].

3.1.1 Libraries and dependencies

Here is a list of the most important Python libraries and dependencies that were used:

NumPy: A library for numeric calculations and linear algebra. Expensive numerical algorithms in this library use a C backend to allow for much faster computations than with pure Python [6].

LibROSA: A library offering a collection of music information retrieval methods and utilities [7].

PyTorch: A deep-learning API written in Python. It enables easy implementation of neural networks using multiple fast methods of defining and setting up the structure. It allows to execute neural networks on a GPU as well, which makes training and prediction with a network much faster than with a CPU [8].

Hydra: A framework which allows for easy and configurable configuration management [9].

3.2 Configuration

Our application can be configured via a configuration file written in YAML format. This file format is well known for its simplicity and readability and therefore enhances easy access to the project [10].

There are three main scripts in our application (see next section). Their parameters can be set via the aforementioned configuration file or via command line overrides. There are numerous parameters grouped by category which can be modified to fit a given separation task.

3.3 Structure

The three main scripts performing the main tasks of our framework are as follows:

data_generation.py Loads raw audio data, generates features and labels as well as does the mixing.

train.py Initializes, loads and trains the CNN model.

inference.py Executes all tasks related to creating the prediction values, which includes reading the audio, segmenting it and storing the output in a given directory.

4 Implementation

4.1 Data generation

Our untrained network needs a dataset of labeled audio mixtures for training. This means we require either a ready-to-use labeled dataset, or we need to create and label the mixes for the dataset ourselves. Since labeled datasets of mixtures are seemingly unavailable, we chose to pursue the latter.

4.1.1 Data selection

First and foremost, labeled audio files of single instrument recordings are needed. There are a lot of datasets to choose from but we settled on Medley-Solos-DB [11]. It provides samples from a number of instruments, while being presented in a file structure that can be parsed easily by our framework.

4.1.2 Dataset modifications

Due to some shortcomings present in the dataset we needed to modify the Medley-Solos-DB dataset. Neural networks learn from training samples. If said samples are predominantly from one class, the network will get biased towards favouring this class in the prediction stage. If the number of samples from each class is evened out, the model will generalize better after training. To ensure a balanced dataset we analyze the dataset in figure 5.

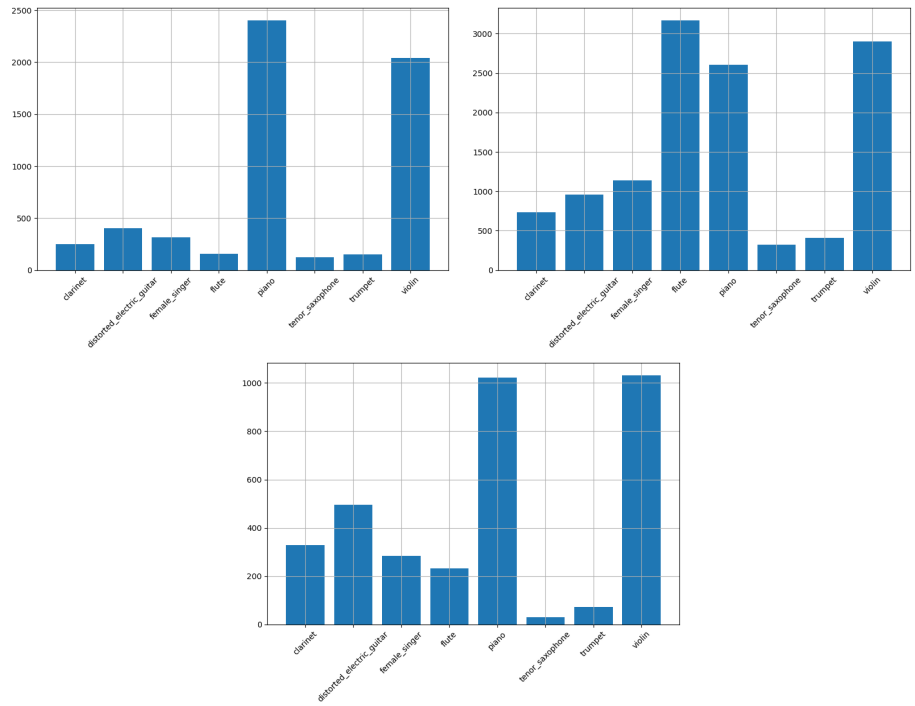


Figure 5: Distribution of the number of samples per instrument for training, test and validation set respectively.

We see that the distribution of instruments is not nearly evened out. If we train our network using this distribution of instruments it will bias towards the instruments with more samples. We can minimize the prediction bias if we oversample instrument segments from classes containing little samples while undersampling segments from classes like the piano, which has plenty of recordings. This implies that not every piano samples will be used, but samples from the clarinet class will most likely be part of a mix many times over. Overfitting should not occur since the solo samples in a mix are very unlikely to ever be grouped together again.

We chose a maximum oversampling factor of 2. This led to approximately 500 samples per instrument, and about 4000 samples in total.

4.1.3 Mixing

Real world audio recordings often consist of multiple instruments playing together. The decision to train the model not only with single instrument signals but also with mixed instrument signals therefore was pretty natural. However, directly using samples with multiple instruments for training bears the problem of not having a proper target vector or label (i.e. the "real" activation of each instrument). In order to solve this problem our framework generates mixed instrument samples itself by mixing recordings of solo instruments.

The mixing process is characterized by three main parameters:

- The number of mixes to create (in total)
- The number of instruments to take per mix
- The level of each instrument in a mix

The implementation of the mixing process is straightforward. Creating one mix requires choosing a set of recordings (either from one instrument or from multiple, different instruments), multiplying them with randomly drawn levels and summing them up.

Number of mixes The total number of mixes to create. Together with the batch size that is being used this parameter specifies the number of training steps in each training epoch.

Number of instruments to take per mix To generate more diverse mixes, the number of instruments per mix was modeled via a uniform distribution. This means, that between a minimum and a maximum number of instruments, each number of instruments is equally probable.

$$p[n_{instr}] = \begin{cases} \frac{1}{n_{instr,max} - n_{instr,min}} & n_{instr,min} \leq n_{instr} \leq n_{instr,max} \\ 0 & else \end{cases} \tag{7}$$

We used $n_{instr,min} = 1$ and $n_{instr,max} = 4$. The lower limit represents a single instrument recording. The upper limit seemed reasonable since our benchmark is the human hearing and it proves difficult to distinguish more than 4 instruments when only being able to hear them for a short period of time.

Levels of instruments in a mix To simulate different levels of instruments in a mix the levels l were also modeled by the help of a probability distribution. We used a normal distribution $\mathcal{N}(\mu_l, \sigma_l^2), 0)$ for that. Via

$$l = \max(\mathcal{N}(\mu_l, \sigma_l^2), 0) + \epsilon \tag{8}$$

it was also ensured that the level l is bigger than 0 (i.e. silence). We chose to use $\mu_l = 0.45$ (approximately -6dB) and $\sigma_l^2 = 0.01$.

The total level of the instruments is $l_{total} = \sum_{i=1}^{N_{instr}} l[i]$. A total level $l_{total} < 1$ is perfectly fine. However, a total level $l_{total} > 1$ is problematic as it may introduce clipping. To prevent that we chose to normalize each respective instrument level $l[i]$ by the total level l_{total} in case it becomes bigger than 1, like in

$$l[i] = \begin{cases} \frac{\mathcal{N}(\mu_l, \sigma_l^2)}{l_{total}} & l_{total} > 1 \\ \mathcal{N}(\mu_l, \sigma_l^2) & else \end{cases} . \tag{9}$$

4.2 Feature extraction

4.2.1 Feature extraction algorithm

The mixing stage generates the audio data as a temporal waveform. To train the network, this audio data now needs to be converted to features from which the network can draw predictions.

When handling audio data a conversion to the frequency domain comes to mind. We decided to compare multiple different frequency representation to see with which transformation our network works best. Therefore we agreed to compare the following representations: a STFT spectrum, a CQT spectrum and a mel-frequency spectrum.

For the STFT spectrum we used a frame size of 2048 and a hop size of 1024 together with a Hann window. Note that these basic parameters were also used for the CQT and the mel-spectrogram. Furthermore we used used $f_{min} = 27.5Hz$ as well as 24 bins per octave and 8 octaves for the CQT. The mel-spectrogram used 128 bins, spanning a frequency range of 27.5Hz to 20kHz.

4.2.2 Output matrix for training (Y-matrix)

Often neural networks predict a one-dimensional output consisting of classes and their likelihood for a given sample. In our first approaches we also used one-dimensional output but quickly realized that this leads to problems with audio data.

If one-dimensional output is used the network is presented a few frames of a spectrogram and estimates one frame of predictions. If a given instrument is present over consecutive frames it will slide through multiple observation windows and the calculated probability will rise and fall depending on its position.

Instrument activations however tend to have a envelope in form of a so called ADSR curve (Attack, Decay, Sustain and Release) and have sharp increases of value especially in the first moments of excitation. This can not be predicted very well via a one dimensional output frame. Since we were trying to enable the model to output predictions as precisely as possible, a better solution was needed.

We decided to create a two-dimensional output matrix instead, which consists of the instruments and their envelopes/activations over time. The activations extend over a fixed number of frames, which in return is defined by the length of the observation window (approximately 300ms).

For supervised training we need labels. These labels in our case are the reference output matrices containing envelopes which are calculated from the base samples. Their modification throughout the mixing process (i.e. the leveling) had to be considered too. The model is trained with the provided envelope matrices instead of one-hot class vectors and, as already mentioned, also outputs a prediction in matrix form.

The activations are computed for each observation window in an audio sample. During training time only one observation window at a time is of interest. However, during

inference we want to have the output matrix of the whole audio file returned by the network. This means that we needed to somehow concatenate the output matrices of each observation window. We decided to stitch together all output matrices of an audio file via overlap-add, using a Hann window and 50% overlap.

4.2.3 Dataset

We created our dataset by mixing samples of single instrument recordings. During the mixing process we made sure to use each available sample at least once. Additionally the dataset was created using five mixing runs. This means, that in a training epoch each single instrument file appeared five times, each time in a different mix.

For the computation of the label values (activations) we decided to use the root-mean square (RMS) value. It describes the energy of an instrument at a given time instance and it can be computed very efficiently in the time domain.

4.3 Model Structure

The structure of the model is visualized in figure 6. The network consists of convolutional layers with increasing kernel sizes (i.e. kernel shape), increasing channel counts (see parameter B) and ReLU activations in between layers. Additionally, after each second convolutional layer there is a MaxPooling layer in order to reduce the size of the feature map, and thus the total number of parameters (weights in the network). After each MaxPooling layer there is also a spatial dropout layer, which drops entire channels in the feature map with a given probability.

After six convolutional layers the feature map is flattened and subsequently fed to a FullyConnected layer. This layer outputs exactly the number of elements in the output matrix in vector form. This vector then is reshaped to the size of our output matrix ($n_{time\_frames} \times n_{instruments}$) in the next layer. In the final layer a sigmoid activation is applied in order to bound the output to $[0, 1]$ (or $[-\infty, 0]$dB).

The model has a total complexity of 1.92M model weights (when CQT features are being used).

4.4 Training

During training we used a batch size of 32. We also used the Adam optimizer with a base learning rate of 0.001 and a weight decay of 0.001 in order to avoid network weights becoming too large. The dataset containing the mixes was shuffled in between epochs.

For the dropout layers we used a dropout rate of 0.1. To avoid overfitting after a certain number of epochs we implemented a learning rate reduction scheduler, which divided the learning rate by 5 each time there was no improvement in validation loss for at least 8 epochs. The minimum learning rate was set to 0.00001. Because of the learning rate

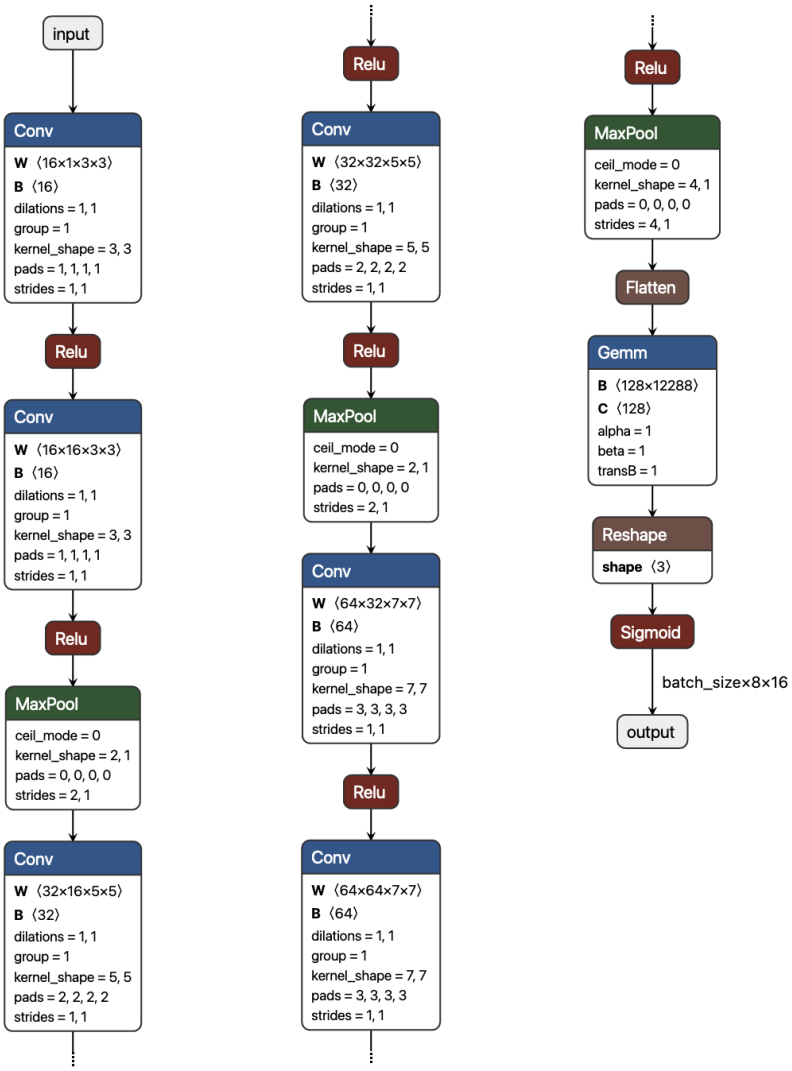


Figure 6: Neural Network structure (left-to-right).

reduction we decided that an early stopping mechanism would not be necessary, so we trained our network for a fixed number of 120 epochs.

For the loss function we used the binary crossentropy (BC) loss. This is pretty unusual for a regression task, as BC loss is usually used in classification. However, with the more prominently used mean-square error (MSE) loss, the predictions of our network collapsed during training. This meant, that after some epochs suddenly all values in the output matrix of the network started being constantly zero for the remaining training process.

We could not find a sophisticated, scientifically backed explanation for this. However, we did research and realized that this often happens in the case of sparse labels being used for training. Our label matrix always consists of eight instruments, while maximally four instruments are playing in a mix. This means that each label matrix is at least 50% sparse. We therefore assume that our label matrices are sparse enough such that collapsing occurs.

This collapsing issue did not occur with the BC loss, so we agreed to use it. We also tried to add additional additive loss terms, like the Kullbeck-Leibler divergence, or a Frobenius norm based loss. However, the best results were produced when solely the BC loss was used.

5 Results

5.1 Comparison of feature types

In the first step we wanted to compare the feature types: STFT, CQT and mel-spectrogram. We generated three separate datasets using each of the feature types. For each of these datasets we trained the model from scratch in order to compare the feature types.

The STFT features did not work at all. The predictions were mostly zero when using these features, not grasping any instrument. This is probably due to the relatively long STFT feature vector (1025 bins) in comparison to the CQT vector (192 bins) and the mel-spectrogram vector (128 bins). Our network is simply not complex enough for 1025 bin feature vectors.

Both CQT and mel-spectrogram features worked well during training of the network. They showed some slight advantages in certain instruments where the other feature type was not that good. Overall their performance was nearly equally well. The CQT features were slightly better though, which is why we decided to use them for the remainder of this work. All the results shown in the subsequent sections also use CQT features.

5.2 Training process

Now we look at the results of the training process. It is best described by the loss curve over the training steps, shown in figure 7.

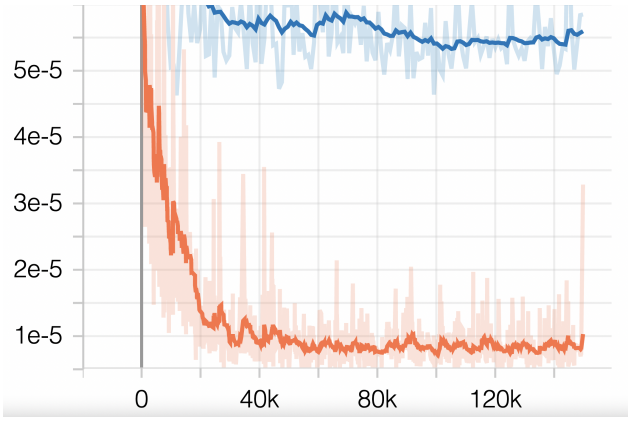


Figure 7: Binary cross entropy loss over training steps, using 120 epochs. The training loss is shown in orange, the validation loss in blue. The blurry lines in the background show the real values, while the thick lines show a moving average over the number of epochs.

At lower number of epochs the loss decreases faster. With a growing number of epochs however, the slope of the decrease becomes lower. The lower the slope becomes the more likely overfitting may happen, which is why we implemented the learning rate reduction. The averaged curves show that after approximately 60 epochs (70k steps) there is no real increase in performance anymore.

This also makes sense if we look at the learning rate in figure 8. The learning rate is at its minimum possible value of 0.00001 from the 35th epoch on (step 40k).

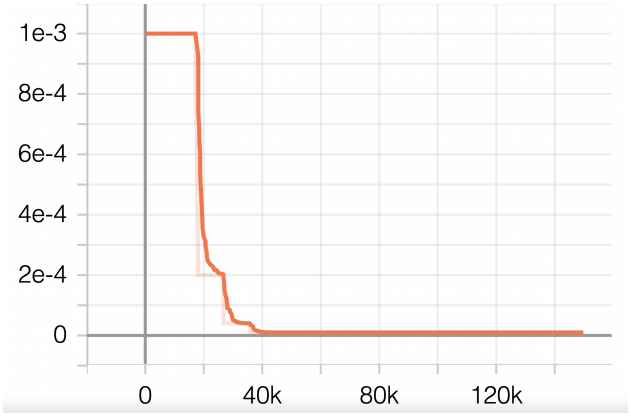


Figure 8: Adaptive learning rate over training steps.

We observe that the network converges pretty fast overall. This may be explained by our training dataset containing each original single instrument recording five times (in five different mixes).

5.3 Predictions

To evaluate the predictions three audio samples were created. The first two samples are named "Sequence1" and "Sequence2". They contain a sequence of all 8 instruments the network was trained with. Each instrument is played sequentially for 5 seconds. By playing them sequentially it can be determined how well the network differentiates between instruments when they're not playing concurrently and how much "leakage" or confusion exists between said instruments.

The third sample was a self-generated example from a piece from the video game Zelda - Ocarina Of Time including the piano, clarinet and flute. This example is especially fitting since only two instruments are playing simultaneously at any given instance. This sample is a simulation of a real world example where multiple instruments are playing at a time, while providing more information, due to the accessibility of the solo tracks.

We limited the dynamic range of our network output (and the labels) to 60dB. Figure 9 shows the results for Sequence1, figure 10 shows the results for Sequence2 and figure 11 shows the results for the Zelda sample.

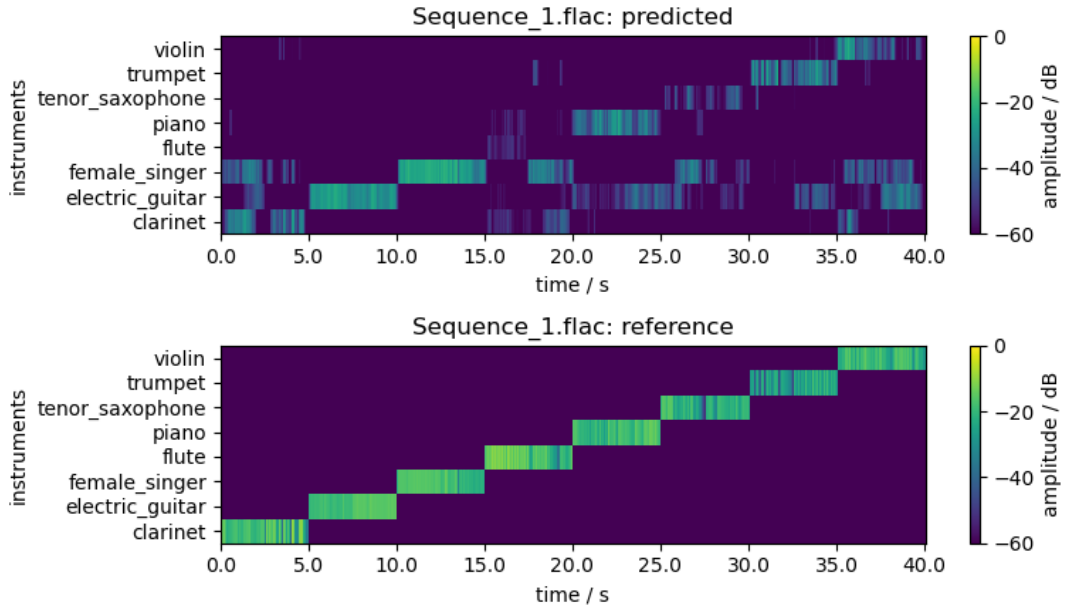


Figure 9: Predictions for the first sequence sample.

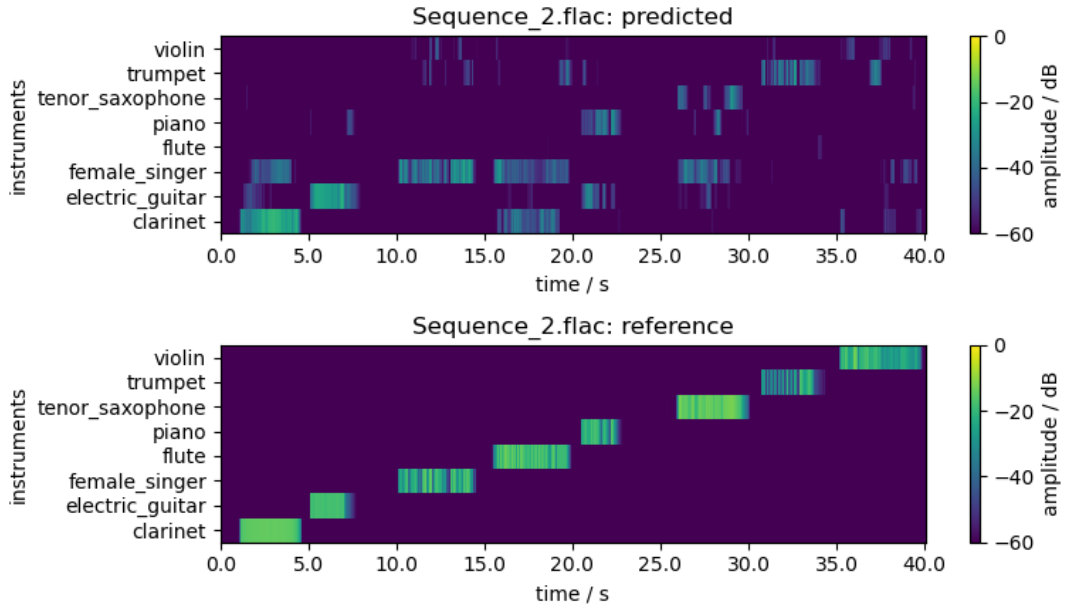


Figure 10: Predictions for the first sequence sample.

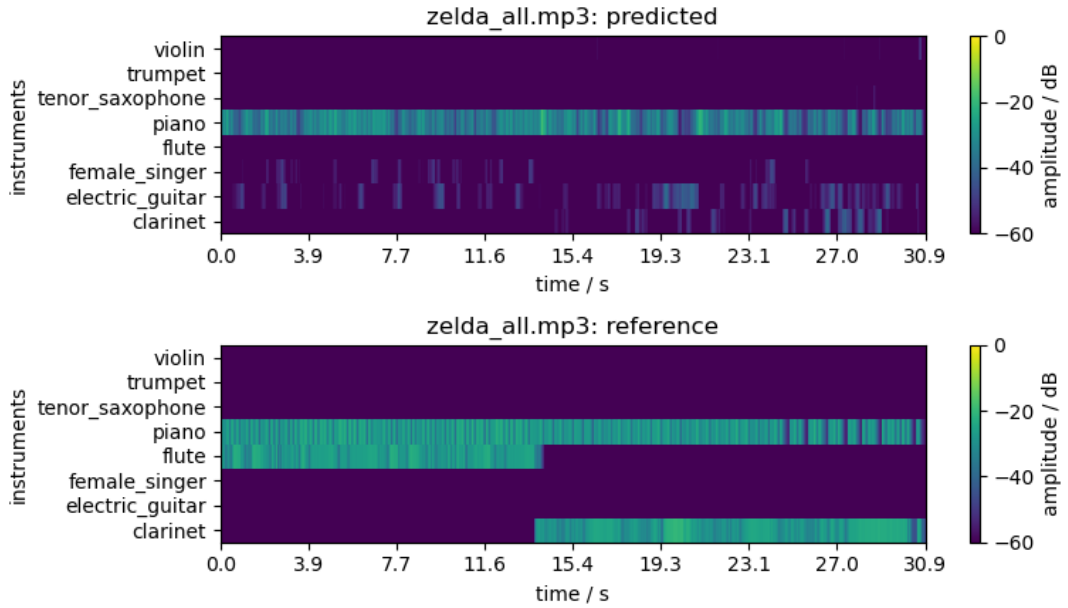


Figure 11: Predictions for the generated sample.

Some of the instruments like the piano, the electric guitar and the female singer are being detected quite well and distinctly. Others like the flute and the tenor saxophone are nearly not being detected at all. A more detailed analyzation can be found in the chapters below.

5.4 Prediction Error

In order to evaluate the performance of a common neural network the prediction results can be analyzed in many different ways. In most cases the error from the expected results is evaluated and the confusion with other classes is being investigated. Evaluating the confusion of classes is tricky for our given task.

Instead of computing a confusion matrix we decided to evaluate the more fitting measures *noise* and *leakage*. We evaluate these measures in a similar way a confusion matrix is normally determined. Note that for the noise and leakage matrices only the two sequence samples were used.

5.4.1 Noise Matrix

Here the resilience to faulty excitation is being investigated. In other words, how much is an instrument being detected when it should not be. This is evaluated by averaging the absolute deviations from the RMS value of the prediction of any instrument over the 5 second window, even if they are not playing (the envelope is zero in this case).

This matrix does not describe the confusion, like the visualization suggests, but only gives insight about how much the network learned to follow the intended envelope (even if it is zero). Furthermore, the values were cut below -60 decibels for visualization purposes since this would be considered not perceptually relevant if another sound source is present.

The noise matrix for Sequence1 is shown in figure 12, the matrix for Sequence2 in figure 13.

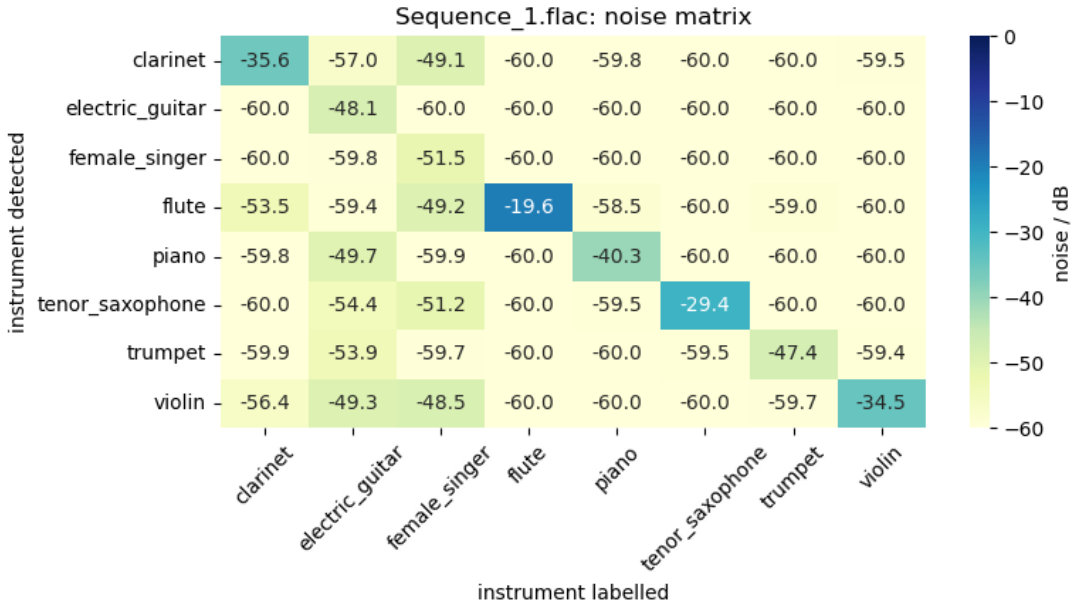


Figure 12: Noise matrix for the first sequence.

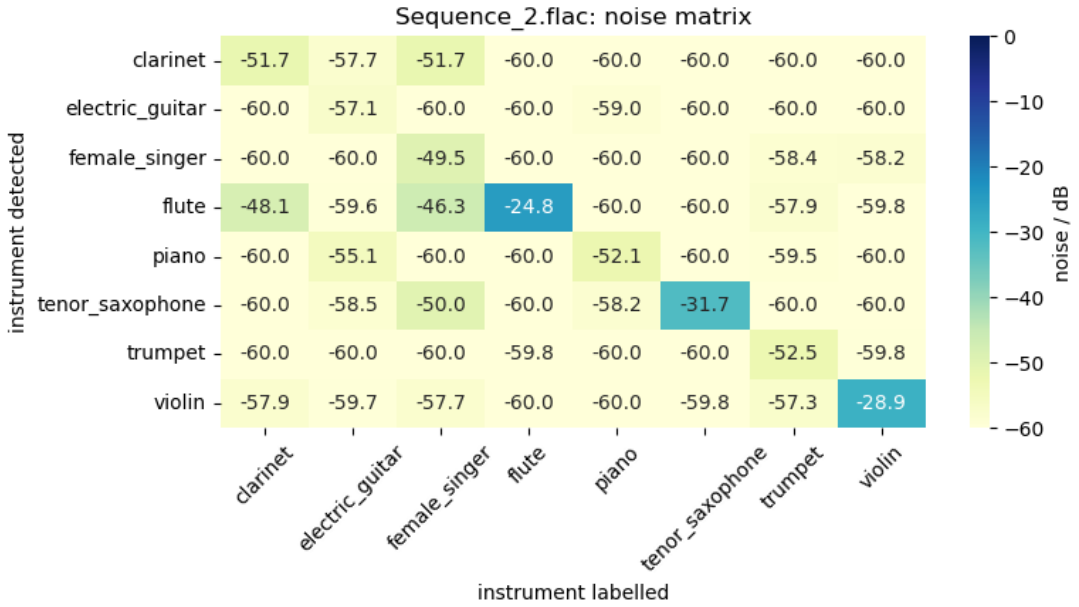


Figure 13: Noise matrix for the second sequence.

The results of the noise matrices are best interpreted by inspecting the female singer and the guitar: the female singer and the electric guitar were falsely detected within nearly every other instrument, since the values of their respective columns are comparably high. On the other hand when looking at their rows, no other instrument was falsely detected while they were playing.

This means that the presence of these instruments in the network output is generally very dominant, and that the network is probably biased towards these instruments. This is why they also tend to generate noise (i.e. predictions where should be silence).

5.4.2 Leakage Matrix

Now we try to construct a confusion matrix from the label matrix and the predicted matrix. In the leakage matrix, we want to analyze how much of the activation values of the currently playing instrument is being detected in another instrument. This means that we compare the predicted activation value of each instrument with the one currently playing. Then, the difference derived from this comparison is subtracted from our defined dynamic range (-60dB).

Note that for the leakage it does not make sense to compare the labeled and predicted value of the same instrument, which is why there are no main diagonal values.

The leakage matrix for Sequence1 is shown in figure 14, the leakage matrix for Sequence2 in figure 15.

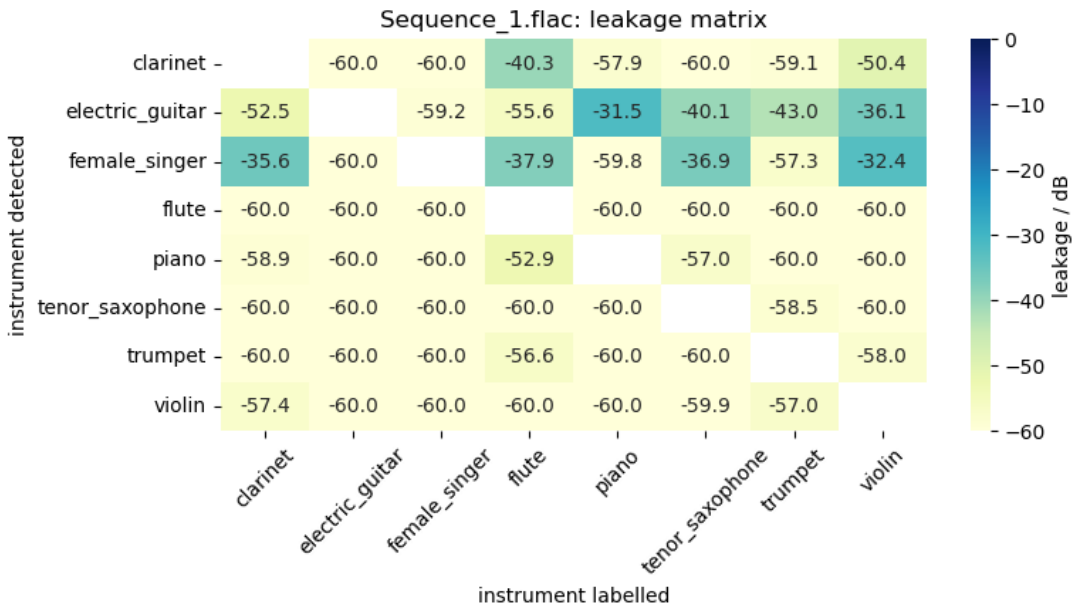


Figure 14: Leakage matrix for the first sequence.

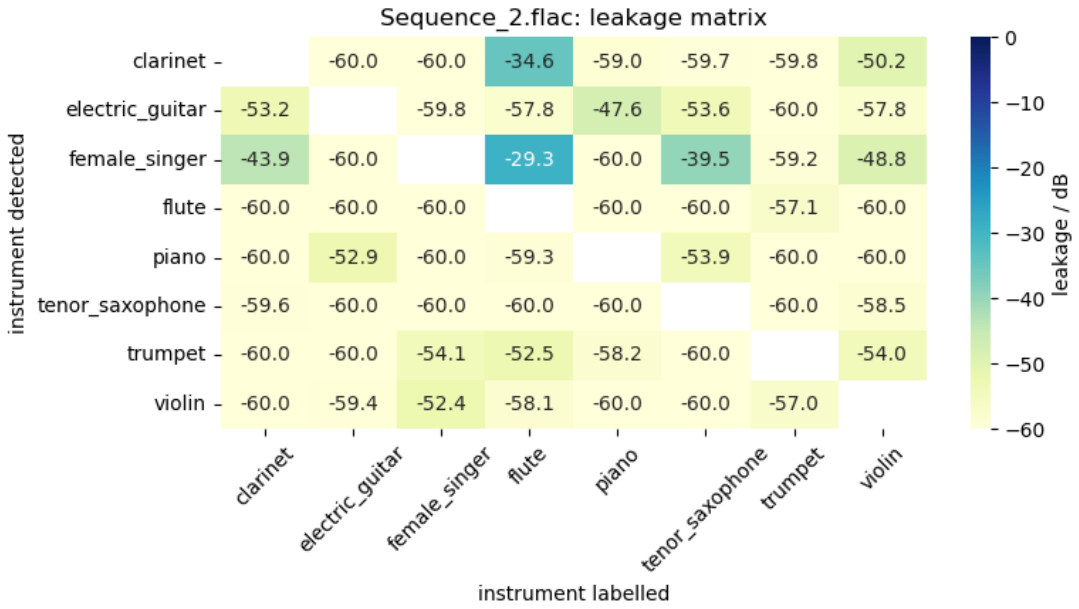


Figure 15: Leakage matrix for the second sequence.

Again we inspect the electric guitar and we find that it nearly does not leak into other instruments, as in the second column of the matrix (where electric guitar is playing) nearly all values are at -60dB. The flute on the other hand is greatly leaking into clarinet, the female signer and the piano, which we can see in its column within the matrix.

5.5 Envelope

For the evaluation of the envelope the "real world" example was put under investigation. Besides evaluating the output of the mixed audio sample, all separate instrument tracks were fed into the network to evaluate the differing performance in respect to mixed and solo signals.

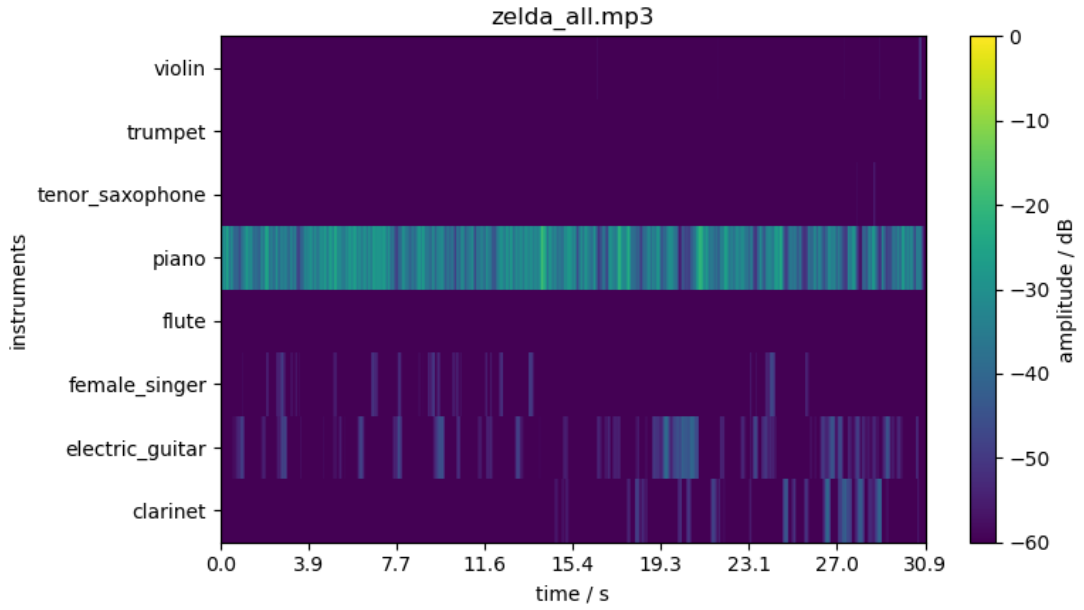


Figure 16: Prediction result of the mixed signal.

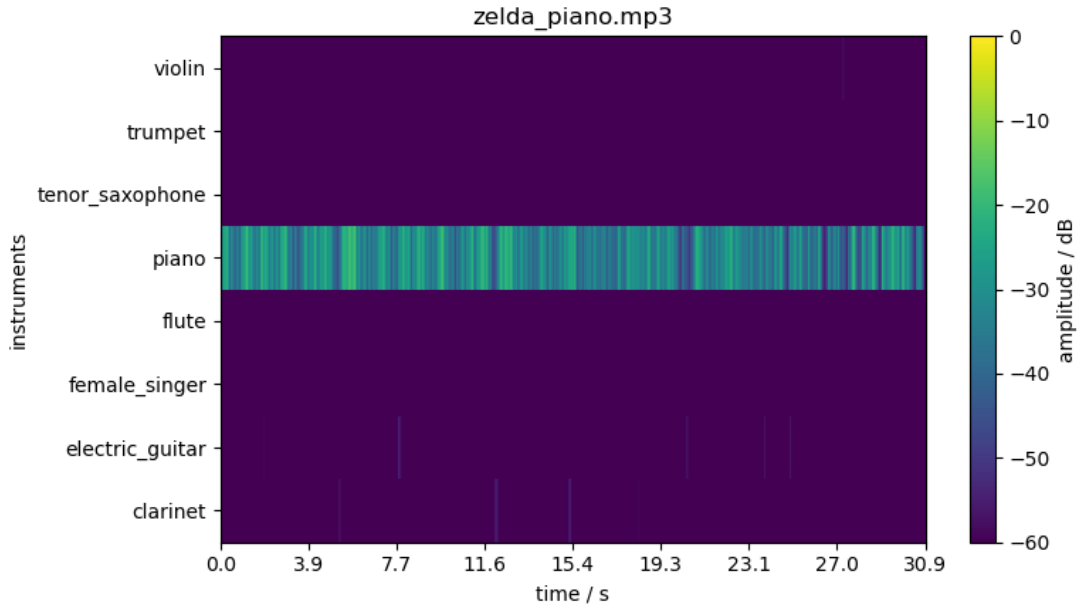


Figure 17: Prediction result of the piano.

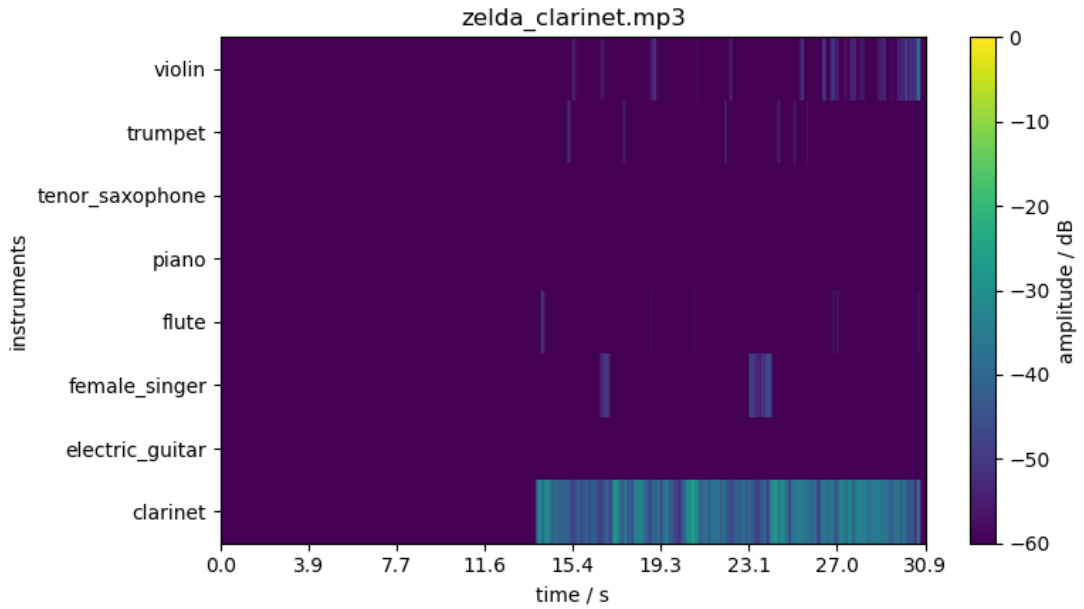


Figure 18: Prediction result of the clarinet.

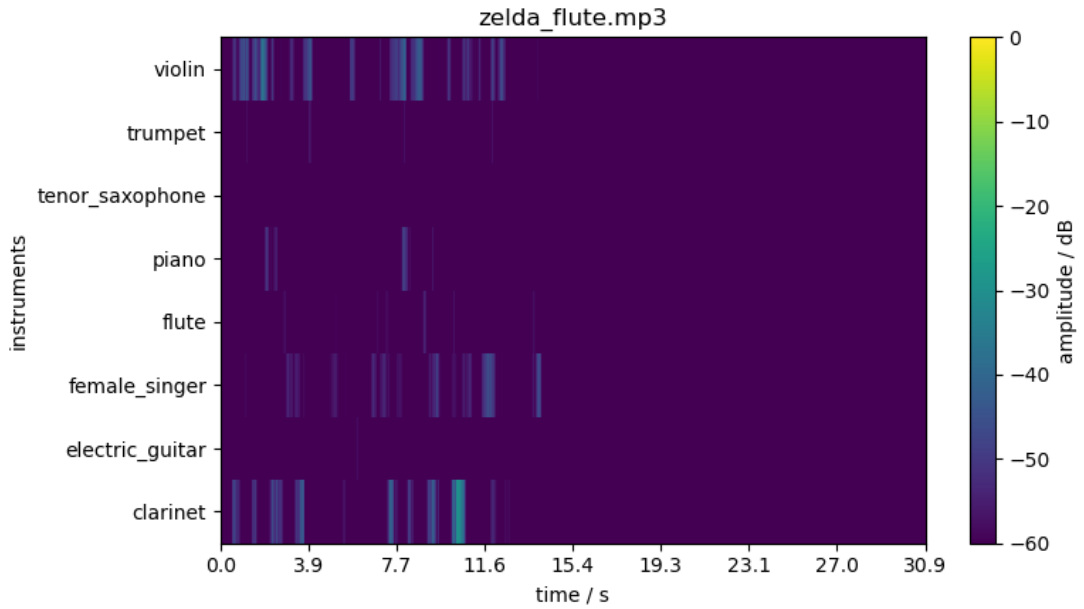


Figure 19: Prediction result of the flute.

The first thing to be stated is the difference of detection quality between the mixed and the solo signals. The network has obvious problems telling the instruments apart, even when being trained on mixed signals. The detection of solo instruments on the other hand performs well on certain instruments (as can be seen for the piano and clarinet) but performs bad for others like the flute.

One thing the network seems to be well capable of (in respect to instruments it can detect sufficiently) is enforcing silence if an instrument is not playing. On the other hand, the envelopes show similarities to the input, while still varying in amplitude and trend. They are also better detected for solo signals when compared to the mixes.

6 Conclusion

In conclusion it can be stated that our CNN is capable of learning and identifying various instruments. However, it only functions properly if they are presented in a successive manner. It has problems distinguishing all the sources of a complex mixture of instruments. More generally it fails to predict where the human ear has troubles too.

The network is able to learn and reproduce complex envelopes. To achieve this we developed a data generation approach which included mixing and leveling the raw audio signals and converting them to features which could be reasonably interpreted by the network. The data set imbalance of the original data set used for training the CNN could be improved significantly as well.

There is room for improvement like more complex pre-processing of the observation frames using source separation algorithms, or applying audio effects to the raw audio

recordings in order to create more diverse training data. Another way of improving the performance would be to take advantage of the two channels present in stereo recordings in order extract more information from the mixtures. Furthermore, a tempo detection could be used to show the networks only relevant slices of the signal. This was implemented in a prototype of this project, but was not investigated further due to the difficulties it brings in relation to synchronicity and varying tempos. Lastly the network structure could be extended with, as an example, an Encoder-Decoder approach.

Usage for this instrument detection approach could be guidance for source separation using algorithms such as Non Negative Matrix Factorization (NNMF) or automated transcription of music.

References

[1] A. Oppenheim and R. Schafer, *Discrete-time Signal Processing: International Version*. 01 2010.

[2] H. Austerlitz, "Chapter 4 - analog/digital conversions," in *Data Acquisition Techniques Using PCs (Second Edition)* (H. Austerlitz, ed.), pp. 51 – 77, San Diego: Academic Press, second edition ed., 2003.

[3] C. Schoerkhuber, "Applications of a constant-q transform for time- and pitch-scale modifications," 12 2011.

[4] J. Brown, "Calculation of a constant q spectral transform," *Journal of the Acoustical Society of America*, vol. 89, pp. 425–, 01 1991.

[5] G. LLC, "2018 kaggle machine learning and data science survey," 2018.

[6] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[7] Brian McFee, Colin Raffel, Dawen Liang, Daniel P.W. Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto, "librosa: Audio and Music Signal Analysis in Python," in *Proceedings of the 14th Python in Science Conference* (Kathryn Huff and James Bergstra, eds.), pp. 18 – 24, 2015.

[8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer,

F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[9] O. Yadan, "Hydra - a framework for elegantly configuring complex applications." Github, 2019.

[10] "YAML: yaml ain't markup language." https://yaml.org/. Accessed: 2020-04-26.

[11] V. Lostanlen, C.-E. Cella, R. Bittner, and S. Essid, "Medley-solos-DB: a cross-collection dataset for musical instrument recognition," Sept. 2018.