

# 236781 – Final project

## Audio Source Separation with DDSP

Tal Skverer  
Amit Zukier

ID: 312489560  
ID: 312500283

E-mail: [talsk@campus.technion.ac.il](mailto:talsk@campus.technion.ac.il)  
E-mail: [amitz@campus.technion.ac.il](mailto:amitz@campus.technion.ac.il)

---

### Abstract

This project explores the possibility of audio source separation using the newly suggested approach of the DDSP library.

Traditional audio synthesis using deep neural networks is hard as-is, even for monophonic audio. One large pledge of the DDSP library is that the large parameter size that was essential to previous works of audio synthesis will be largely reduced, allowing for easier training, which in turn might open avenues of synthesis that were originally not possible.

We would like to examine this claim by challenging the library on a problem that is closer to the "real world" than the one presented in the article: separating instruments from a polyphonic audio recording.

Using a dataset of music containing 3 monophonic sources; drums, bass and vocals and their combined mixture, we trained a deterministic autoencoder like presented in the original paper as a baseline for each source, as well as a network based on temporal convolutions that creates a mask used to separate audio to its original sources' audio features: fundamental frequency and loudness.

Combined, they create a full network that is theoretically capable of audio separation on these exact sources.

Unfortunately, we found that the quality of the resynthesized "difficult" sounds (vocals, bass, and drums) utilizing DDSP's elements was inadequate.

Furthermore, the suggested model for separating the audio features from a mixture of sources was incapable of yielding satisfactory results.

Eventually, this led to the entire model's result to be quite different from its input (if still quite amusing to listen to), while resembling only a little of its original features.

In a more personal note, we chose this project without any prior knowledge in field of digital signal processing in order to learn and challenge ourselves. During the project, we learnt a lot about signals, and related methods, and found the subject to be extremely interesting. We discovered that much work was done in this field (and specifically for audio source separation) using deep learning, by utilizing a large variant of model types, and achieving very interesting results. Most of these models were covered in the course and we would like to take this opportunity to thank the staff for their work and mention that we had very good experience taking it.

---

## 1 Intro

Differential Digital Signal Processing (or DDSP for short), is a library which enables the addition of signal processing elements into TensorFlow<sup>1</sup>, a well-known library with modern automatic differentiation.

This work was done to fill in the gap in neural networks training for audio generation.

While most generative models (such as WaveGAN<sup>2</sup>, SING<sup>3</sup>, MCNN<sup>4</sup>, WaveNet<sup>5</sup>, GANSynth<sup>6</sup> and others) directly generate samples in either the time or frequency domains (and sometimes both of them), DDSP integrates classic signal processing elements (synthesizers and effects), to improve neural networks' approximation by using the strong structural priors of these tools, which promotes generalization. Specifically, unlike the works mentioned before, using said DSP elements might successfully incorporate their ability to convey audio, as they align with the data domain.

This is, as explained in the paper, because these elements exploit the periodic structure of resonating, similarly to how the human ear has evolved, unlike other audio synthesis models.

For example, generative models such as WaveGAN generate waveforms directly.

Since audio usually includes many frequencies, the model must generate aligned waveforms, included with every filter applied to them, which is generally very challenging. Using a harmonic oscillator (called an additive synthesizer) in a neural network eliminates this issue by automatically outputting a signal with several sinusoidal components at harmonic frequencies (integer multiples of a fundamental frequency), requiring the network to solely learn the necessary single frequency.

More issues covered in the original paper are of Fourier-based models such as GANSynth which suffer from spectral leakage problem, and autoregressive waveform models such as WaveNet that bypass all aforementioned problems by generating the waveform a single sample at a time. However, this causes them to require larger networks to learn this complex model and exposes them to bias during generation.

The DDSP library<sup>7</sup> currently features 6 interpretable DSP elements implemented as TensorFlow layers: 3 synthesizers and 3 effects:

- Additive synthesizer (a harmonic oscillator)
- Filtered noise (“subtractive” synthesizer)
- Wavetable (interpolative lookup from small chunks of waveforms)
- Reverberation (3 sub-methods of reverb)
- FIR Filter (linear time-varying finite impulse response)
- ModDelay (variable length delay lines)

The implementation and the demo presented in the paper arises some issues and drawbacks<sup>8</sup> of the methods introduced:

1. While the experiments presented in the paper showed success with synthesizing monophonic audio, it was unclear how and if synthesizing polyphonic audio is

<sup>1</sup> [TensorFlow](#), a public tool for machine learning, which supports [auto-differentiation using computation graph](#)

<sup>2</sup> Chris Donahue, Julian McAuley, and Miller Puckette. [Adversarial Audio Synthesis](#)

<sup>3</sup> Alexandre Defossez, Neil Zeghidour, Nicolas Usunier, Leon Bottou, and Francis Bach. [SING: Symbol-to-Instrument Neural Generator](#)

<sup>4</sup> S. O. Arik, H. Jun, and G. Diamos. [Single-Image Crowd Counting via Multi-Column Convolutional Neural Network](#)

<sup>5</sup> Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, Koray Kavukcuoglu. [WaveNet: A generative model for raw audio](#)

<sup>6</sup> Jesse Engel, Kumar Krishna Agrawal, Shuo Chen, Ishaan Gulrajani, Chris Donahue, Adam Roberts. [GANSynth: Adversarial Neural Audio Synthesis](#)

<sup>7</sup> Code available in [GitHub](#), also a direct URL to the elements' implementation: [synthesizers](#) and [effects](#).

<sup>8</sup> Some were also discussed by reviewers on the [paper's OpenReview page](#).

possible. As synthesizing several notes, from different instruments will without doubt require different network architecture, such as more encoders, altering the latent space to be able to represent all necessary information, and including additional DDSP elements for synthesis.

Additionally, the demo used Spectral Modeling Synthesis<sup>9</sup> for synthesis, due to its large parametric size, which was expected to lead to high expressiveness. For larger networks, that might be required for polyphonic synthesis, this large parametric size might become a constraint to successful learning process, and a less expressive model will be required, which in turn might hinder quality polyphonic audio generation.

2. Since the output of the decoder goes simultaneously into different DDSP elements, the latent representation is highly entangled and therefore networks with many elements might fail to give any meaningful interpretability to the latent code.
3. We feel that there have been insufficient experiments to show the claims of the strength of DDSP. There are only three variants of DDSP elements that were used (additive synth, white noise filter and reverb), and for relatively simple task – synthesizing a monophonic, single, and somewhat simple instrument (at least in the sense that it's easy to recreate using digital methods as already implemented in the library).

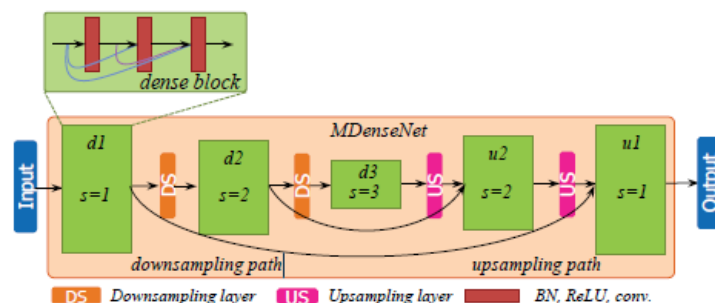
These drawbacks were the original reason we picked audio source separation as a task we are looking to solve using DDSP's suggested methods.

We then overviewed state-of-the-art Audio Source Separation (ASS) works.

A larger emphasis was put on models that might fit with the DDSP elements the library presents.

We further investigated models that seemed most relevant: MMDense-LSTM<sup>10</sup> and Conv-TasNet<sup>11</sup>.

**MMDense-LSTM** is an enhancement of the CNN based MMDense model with LSTMs with skip connections.



**Fig. 1.** MMDenseNet architecture. Multi-scale dense blocks are connected through down- or upsampling layer or through block skip connections. The figure shows the case  $s = 3$ .

The MMDense model works on a few frequency bands, in different sizes, thus the “MM” representing “Multi-scale Multi-band”. On each band, the model down-samples the input and then up-samples it.

<sup>9</sup> Xavier Serra, [A system for sound analysis/transformation/synthesis based on a deterministic plus stochastic decomposition](#).

<sup>10</sup> Naoya Takahashi, Nabarun Goswami, Yuki Mitsufuji, [MMDenseLSTM: An efficient combination of convolutional and recurrent neural networks for audio source separation](#).

<sup>11</sup> Yi Luo, Nima Mesgarani, [Conv-TasNet: Surpassing Ideal Time-Frequency Magnitude Masking for Speech Separation](#).

To further improve the basic MMDense, the authors of the MMDense-LSTM paper added LSTMs into the MMDense architecture, as shown in the following diagrams from their paper:

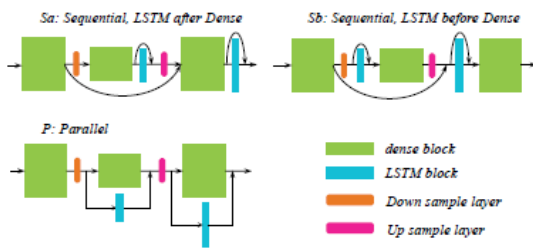


Fig. 2. Configurations with different combinations of dense and LSTM blocks. LSTM blocks are inserted at some of the scales

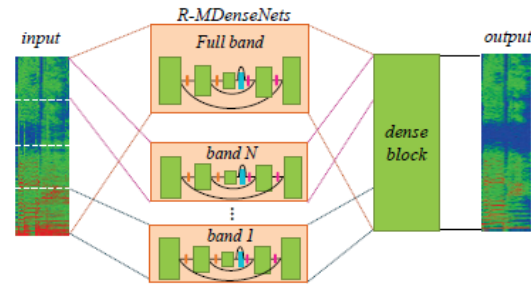


Fig. 3. MMDenseLSTM architecture. Outputs of MDenseLSTM dedicated to different frequency band including the full band are concatenated and the final dense block integrates features from these bands to create the final output.

This model was shown to achieve state-of-the-art results on the DSD100 dataset.

**Conv-TasNet** is a convolutional-based network, using mask-generation approach for speech separation - by finding a mask for each of the signals. That mask is found by using a Temporal Convolutional Network (TCN) on a representation encoded by a single convolution. After the TCN, the mask is applied to the encoded representation and then decoded.

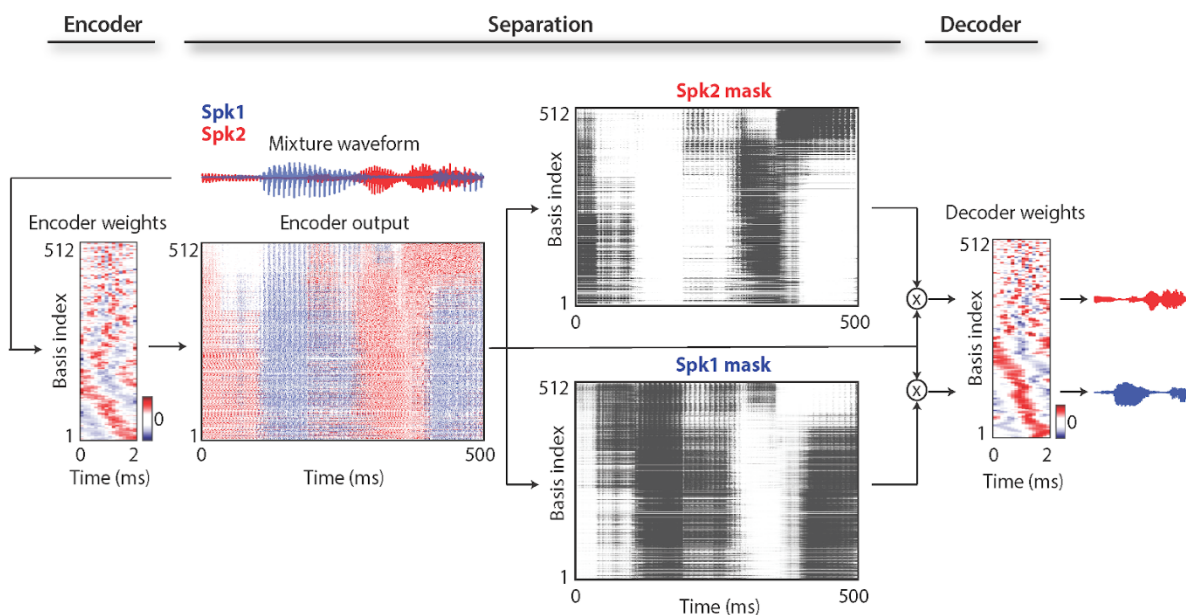


Figure 1. Conv-TasNet model

TCN performs time dilated convolution, thus learning with “memory” similar to how recurrent neural networks do, but without some of the issues that come with them such as exploding or vanishing gradients.

Diagram of a TCN taken from a relevant paper<sup>12</sup>:

<sup>12</sup> Shaojie Bai, J. Zico Kolter, Vladlen Koltun, [an Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling](#).

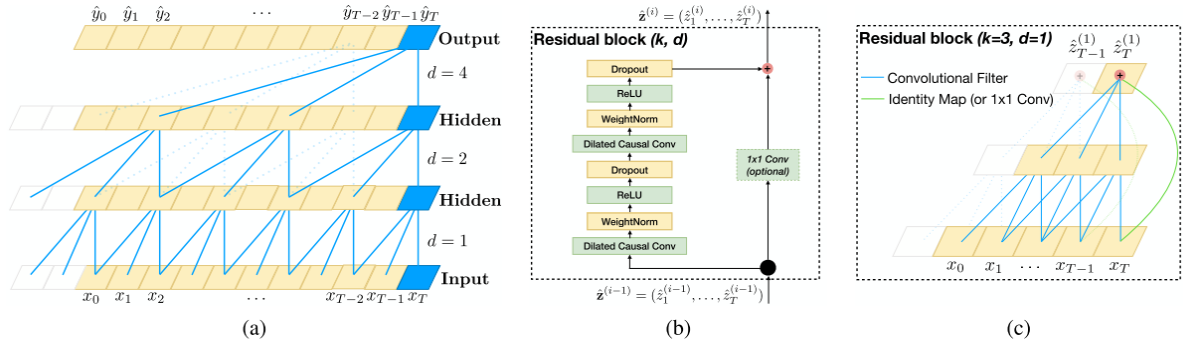


Figure 1. Architectural elements in a TCN. (a) A dilated causal convolution with dilation factors  $d = 1, 2, 4$  and filter size  $k = 3$ . The receptive field is able to cover all values from the input sequence. (b) TCN residual block. An  $1 \times 1$  convolution is added when residual input and output have different dimensions. (c) An example of residual connection in a TCN. The blue lines are filters in the residual function, and the green lines are identity mappings.

Conv-TasNet was shown to be able to achieve state of the art results for music source separation<sup>13</sup>.

<sup>13</sup> In a paper by David Samuel, Aditya Ganeshan, Jason Naradowsky, [Meta-learning Extractors for Music Source Separation](#) and a paper by Alexandre Défossez, Nicolas Usunier, Léon Bottou and Francis Bach, [Music Source Separation in the Waveform Domain](#).

## 2 Methods

Firstly, we will cover the implementation of two chosen elements described in the paper. This step was utmost important to us, as understanding the process is essential to figure out how the authors integrated these DSP elements into a neural network and made them differentiable.

At the heart of the code’s design is a *processor* object, the main type of object in the library, and the base class for both the synthesizers and effects detailed below.

A processor receives an input, which is generally an output of a neural network, and outputs some signal relevant to its function. Internally, such object converts the input vector(s) to a valid-formatted vector(s) for this specific processor, called *controls*, and then processes these controls to create a *signal*. This process is visualized in figure 2.

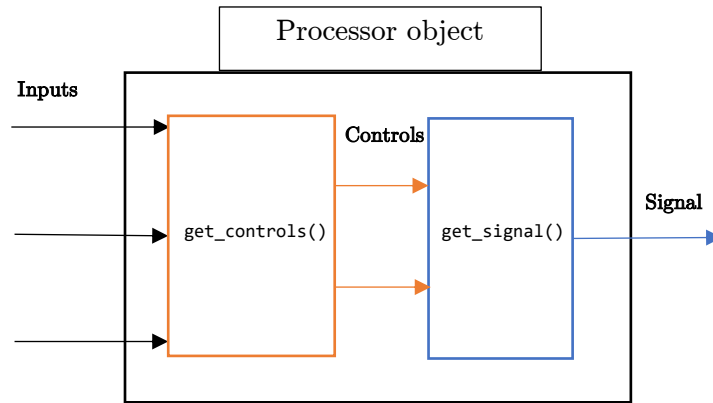


Figure 2. DDSP’s processor object

### 2.1 Additive Synthesizer

#### Hyperparameters

An additive synthesizer processor has 4 available hyperparameters:

1. Number of required samples from the resulted signal  $\mathbf{T}$  (defaults to 64000).
2. Audio sample rate  $\mathbf{S}$  (defaults to 16000, combined with the default samples this results in 4 seconds of audio).
3. Scaling function for the amplitudes and the harmonic distribution (defaults to an "exponentiated sigmoid" function).
4. Whether to eliminates frequencies above the Nyquist frequency<sup>14</sup> (default to enabled).

#### Controls

The additive synthesizer accepts 3 inputs (Denoting all time frames as  $\mathbf{F}$ ):

- **Amplitudes**, a 3-D tensor  $\mathbf{A}$  of shape  $(\mathbf{B}, \mathbf{F}, \mathbf{1})$ , that for each batch, contains a 2-D tensor that has an amplitude value for each time frame.
- **Harmonic distribution**, a 3-D tensor  $\mathbf{H}$  of shape  $(\mathbf{B}, \mathbf{F}, \mathbf{N}_h)$ , that for each batch, contains a 2-D tensor that has  $\mathbf{N}_h$  harmonic distributions for each time frame.

<sup>14</sup> [Nyquist frequency](#) is half of the sampling rate of a discrete signal. Sampling frequencies over it can cause aliasing.

- **Fundamental frequencies**, a 3-D tensor  $\mathbb{F}_0$  of shape  $(\mathbf{B}, \mathbf{F}, \mathbf{1})$ , that for each batch, contains a 2-D tensor that has a fundamental frequency value for each time frame. Treats the values as if they were in hertz.

Three issues with the input values may arise, and the processor fixes them to be valid controls:

1. Amplitudes must be non-negative values.
2. The sum of all distributions must be 1.
3. Any frequencies above the Nyquist frequency might lead to aliasing.

To handle these problems, the processor executes the following steps:

- Scales  $\mathbb{A}$  and  $\mathbb{H}$  using the scale function.
- If enabled, changes the amplitude of any frequency above the Nyquist frequency to 0.
  - o Creates the harmonic frequencies from  $\mathbb{F}_0$  by multiplying every integer frequency up to  $N_h$  new frequencies.
  - o Zeroes the probability of every harmonic frequency that is above half the sampling rate. (This is done to allow the next step to work properly, while also voids the need to zero out the actual amplitudes).
- Normalizes  $\mathbb{H}$  by dividing every value by the sum of all values, effectively making the sum of all distributions to 1.

### Signal

It is important to note that the generated audio is monophonic by design. Therefore, the processor receives fundamental frequencies, amplitudes, and harmonic distributions and creates the audio waveform by:

- Creating two 3-D tensors  $\mathbb{H}_f, \mathbb{H}_A$  of shape  $(\mathbf{B}, \mathbf{F}, N_h)$ , that for each batch, contains a 2-D tensor that has  $N_h$  harmonic frequencies/amplitudes for each time frame.
- Resamples both tensors by interpolating to total number of time steps (according to the hyperparameter) by up-sampling using hamming window method<sup>15</sup> for amplitude envelopes, and linearly<sup>16</sup> for frequency envelopes. This creates two new tensors of envelopes  $\mathbb{E}_f, \mathbb{E}_A$  of shape  $(\mathbf{B}, \mathbf{T}, N_h)$ .
- Generates audio. First, creates phases  $\varphi$ , where  $k$ -th element is:

$$\varphi[k] = \sum_{i=0}^k \frac{2\pi \mathbb{E}_f[i]}{S}$$

Summed over all samples, and the final audio  $A$ , where its  $k$ -th element is:

$$A[k] = \sum_{i=0}^{N_h} \mathbb{E}_A \cdot \varphi$$

Summed over the last dimension (all sinusoids).

## 2.2 Reverb

### Hyperparameters

A reverb effect processor has 4 available hyperparameters:

1. Whether the processor should learn the impulse response (defaults to false).
  - a. Length of the impulse response (defaults to 48000).
2. Whether to add the dry audio to reverberated signal (defaults to true).

<sup>15</sup> [Han and Hamming windows](#), specific type of a cosine-sum window function.

<sup>16</sup> [Bilinear interpolation](#), one of the basic techniques of resampling.



### Controls

The effect accepts 1 or 2 inputs – audio to add reverb to, and impulse response, if it is not learnable.

Creating the controls does not do much besides pad the impulse response as needed to match the batch size (since the learnable data is global for the whole dataset).

### Signal

The signal process expects two input – the audio and the impulse response.

It then proceeds to mask the first impulse response to mask the dry signal, and then convolve the signal with new impulse response using FFT.

In conclusion, every operation done on the input tensors detailed on controls and signal generation process of these DSP element is differentiable. Therefore, it is clear that each processor is differentiable, and can be simply added as a layer in a neural network, letting the auto-differentiation method of TensorFlow – the tape – to do all the work by going backwards through the operations when required.

In addition, while some elements are deterministic based on the input they receive from the neural network (such as the additive synthesizer), other elements are capable of learning like any part of the model (such as the impulse response of the reverb processor).

## 2.3 Audio Source Separation using DDSP

Utilizing the premise of the DDSP library, as was shown in its short demo<sup>17</sup>, we try to transform few encoders that learnt a specific musical instrument to the task of ASS.

Thus, unlike existing works in the field that try to separate entire signals, we only have to separate the fundamental frequency and the loudness of each source.

Since we extract much less information, we hope that this will prove to be an easier task and will require less trainable parameters and shorter training time than extracting an entire signal correctly.

We have decided to pre-train an adaptation of Conv-TasNet to be used in-place for the fundamental frequency and loudness extraction.

A specific encoder will be pre-trained to handle each expected source.

Afterwards, these models will be combined so our Conv-TasNet will output the source's fundamental frequency and loudness to the relevant autoencoder, which will then forward its output into an additive synthesizer + white noise combination.

Their output should be the final separated audios.

## 2.4 Dataset – DSD100

There are few requirements from the sources in the dataset we use:

1. They must have each source split into a monophonic recording, and a mixed version of all sources. This is so we can train stand-alone encoders successfully.
2. The mixtures should have most of the following requirements: similar style, same instruments, or same artists. This is to decrease how much the model needs to generalize, since we are looking to see if it can successfully separate sources at all.
3. Total tracks length should be around 15 minutes of audio. This number is derived from the DDSP demo that empirically shows that this amount of audio is enough for decent learning.

---

<sup>17</sup> Timbre transfer demo in the [DDSP online supplement](#).



Originally, we looked at the ‘Mixing Secrets’ multitrack library<sup>18</sup>.

In this archive, the original contributors have uploaded raw audio and their combined output, without additional effects or processing.

This library turned out to be too disorganized for our needs, since we required samples which had the same instruments, with combined tracks for each one (while ‘Mixing Secrets’ usually had a different track for every microphone in the room, live performances and other complications).

By looking at branches of this project, we found DSD100<sup>19</sup>, a dataset of full-length music tracks of different styles along with their isolated drums, bass, vocals and other stems, which was derived from the ‘Mixing Secrets’ library and that its subset (MUSDB18) was recently used for signal separation evaluation challenges.

Taking the drums, bass and vocals as our 3 sources “killed” two birds with one stone: we got to test the basic DDSP model on difficult source, vocals, rather than strings, along with having good sources to try and reconstruct the mixture from. Note that we decided to remove the “other stems” from the dataset we will use, as it contains several different musical instruments playing simultaneously, which does not fit to our model of a DDSP decoder per musical instrument.

It is important to note that similar works in this field also incorporated DSD100 as an evaluation dataset, which allowed us to easily compare results.

We were also able choose a single artist, James May<sup>20</sup>, as the dataset had enough track time for both training and evaluation.

Specifically, track 21 (“On The Line”) was used for testing, while the other tracks were split into small sub-tracks with 1/3 used for validation and the rest for training.

### Data preprocessing

Using Audacity<sup>21</sup>, we preprocessed the data by:

- Turning the audio tracks into single-channel from two-channel (stereo to mono).
- Concatenating the separated sources into a whole mixture.

For the DDSP autoencoder, we used the built-in DDSP preprocessing, which:

- Splits audio into 4-second excerpts.
- A sliding window that hops 1 second every time creates records with some neighbor overlap.

For both the Conv-TasNet adaptation and the DDSP decoder, we used the DDSP implemented preprocessor mentioned above.

We combined each split with the mixed audio, the fundamental frequency and loudness of each source, which were extracted using CREPE and a deterministic method respectively.

---



---

<sup>18</sup> [‘Mixing Secrets’](#), a free multitrack library for mixing practice purposes by Cambridge music technology.

<sup>19</sup> [DSD100](#), a dataset of 100 full lengths music tracks for signal separation evaluation, derived from ‘Mixing Secrets’ library.

<sup>20</sup> James May, pop/rock artist, [tracks 20, 21, 70 and 71](#) from the DSD100 dataset.

<sup>21</sup> [Audacity](#), free multi-track audio editor and recorder.

### 3 Implementation and experiments

In this section, we will cover the implementation of our whole network model, starting from the changes done to the DDSP autoencoder, how we adapted the Conv-TasNet model, and the integration of the pre-trained networks to a full model with ASS capacity.

#### 3.1 DDSP autoencoder model

The model presented in the paper expects a monophonic, single-channel audio recording as an input  $I(t)$  that for each time step  $t$  its amplitude is sampled according to the chosen sample rate.

The processed input is then passed through an encoder that consists of 3 parts:

1.  $f_0(t)$  – fundamental frequency extracted using a pre-trained CREPE model
2.  $l(t)$  – loudness extracted using a deterministic DSP method<sup>22</sup>.
3.  $z(t)$  – latent encoding of the residual information.

Afterwards, these 3 tensors are forwarded to a decoder, that reconstructs them into inputs to an additive synthesizer and white noise filter, that are combined into a reverb effect (FIR) that finally outputs the reconstructed audio.

Below are the diagrams of the encoder (of  $z(t)$ ) and the decoder.

(Over all diagrams, we used a notation similar to the paper: red components are part of the neural network architecture, green components are some latent representation, and yellow components are deterministic synthesizers and effects)

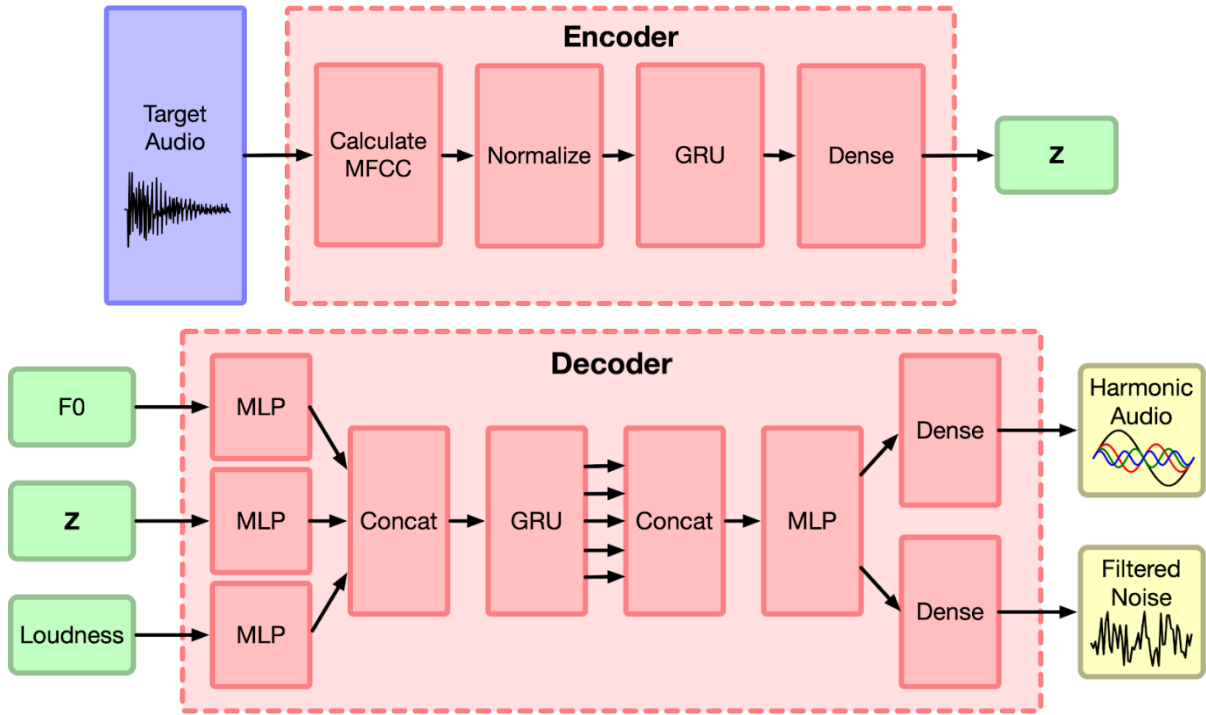


Figure 2. DDSP's autoencoder model

For each isolated instrument in our dataset ([see above](#)), we will train a dedicated DDSP autoencoder separately, using a similar model as described above, but eliminating  $z(t)$  for easier training (the demos in the library demonstrated that this is feasible), and the reverb to try and decrease the model's parameter size.

<sup>22</sup> Implemented in the `spectral_ops` file, under `compute_loudness` function, [Source code](#). This method is based on the paper [Fast and Flexible Neural Audio Synthesis](#), section 3.3.1

This, in fact, makes the encoder part of the model deterministic, but we will still call it an autoencoder.

Diagram of the autoencoder training, for each  $i \in [1, 2, \dots, S]$ :

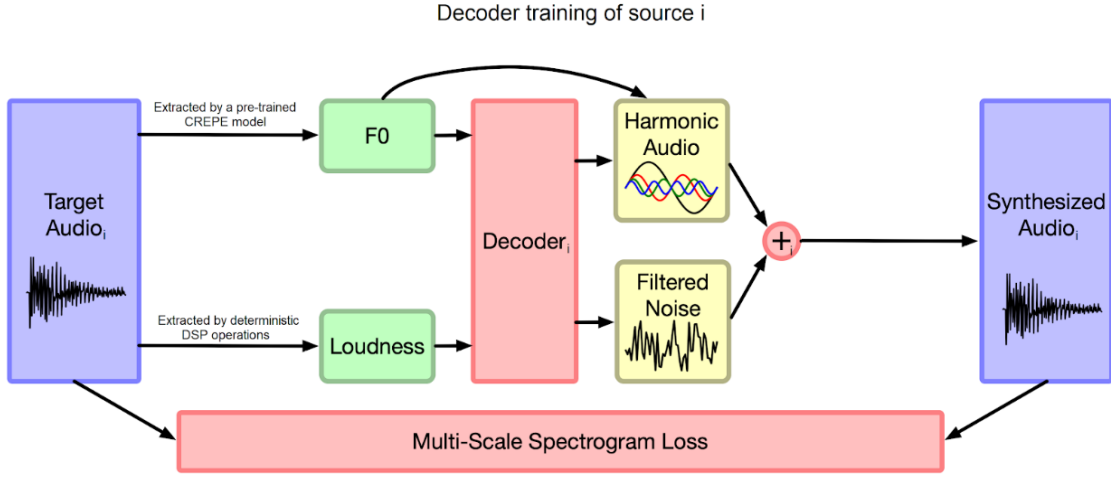


Figure 3. Autoencoder pre-training model

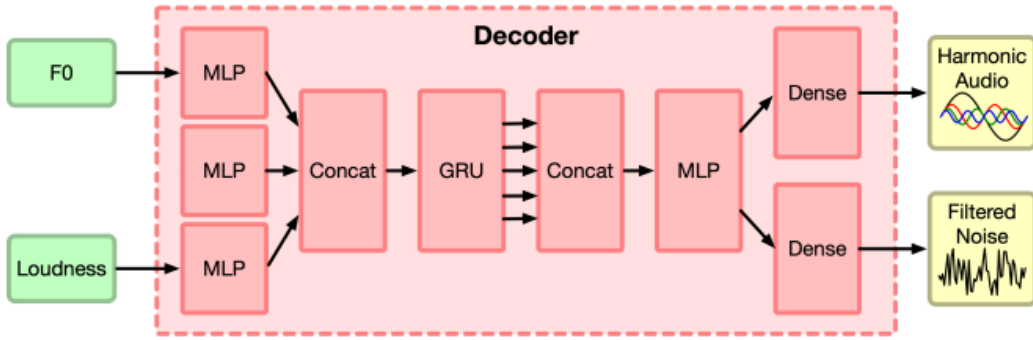


Figure 4. Internal decoder diagram

For the autoencoder training, we used the DDSP library as is, using its Gin<sup>23</sup> library implementation by creating our own Gin configuration file: `our-solo-instrument.gin`, which differed from the original autoencoder model by not utilizing the  $z(t)$  encoder and the reverb element (as seen above).

For the decoder we used ‘GRU’ RNN type with 512 channels. The MLP layers was 3 layer-deep with each layer having a dense network with 512 units, layer normalization and RELU activation layer.

The hyperparameters of the model were similar to the ones used in the paper’s demo: learning rate of  $3 \cdot 10^{-4}$ , batch size of 32, and the optimizer used was Adam with an exponential decay rate of 0.98 every 10,000 steps.

Every learning step, the gradients were clipped by a global norm by a ratio of 3.

We used the multi-scale spectral loss as presented in the paper (section 4.2.1).

The training was initially executed for a up to 1,000,000 steps. Using the library’s best practices as mentioned in one of the training examples, we stopped early when the loss dropped to an average range of 4.5, which took around 14,000 steps for vocals, 11,000 steps for bass and 8,100 for drums.

<sup>23</sup> [Gin](#), a lightweight configuration framework for Python.

### 3.2 Conv-TasNet adaptation

Contrary to the approach presented in the DDSP paper, our input consists of  $S$  different sources that might (or might not) appear on the amplitude sampling for each time step. Therefore, our input can be seen as the sum of each source's amplitude:  $Input(t) = \sum_{i=1}^S s_i(t)$  where  $s_i(t)$  is the amplitude of the source  $i$  at time step  $t$  (can be 0).

Therefore, we cannot simply use a pre-trained CREPE model to extract the  $f(t)$  of every source since it is only suitable for single-sourced, monophonic audio.

There's a second variant of model presented in the DDSP model, which learns the fundamental frequency as well as the latent representation. This method achieved mediocre results, comparable to the CREPE model, and nevertheless the architecture was not designed with multiple sources in mind. In fact, it lacks a necessary “memory” of the input in a previous time  $t' < t$  which is required, in our opinion, to correctly distinguish which fundamental frequency belongs to which source.

Furthermore, the task of differentiating a single source from multiple naturally sounds like some filter needs to be applied on the audio, the same way the Conv-TasNet model works.

The Conv-TasNet model is an auto-encoder. It firstly encodes the input audio samples into a latent representation (denoted as  $w$ ) with a 1D convolution with kernel size  $L$  and stride  $L/2$ , so each output element is generated from an input audio sample with 50% neighbor overlap. The encoder also expands the input single channel to  $N$  channels.

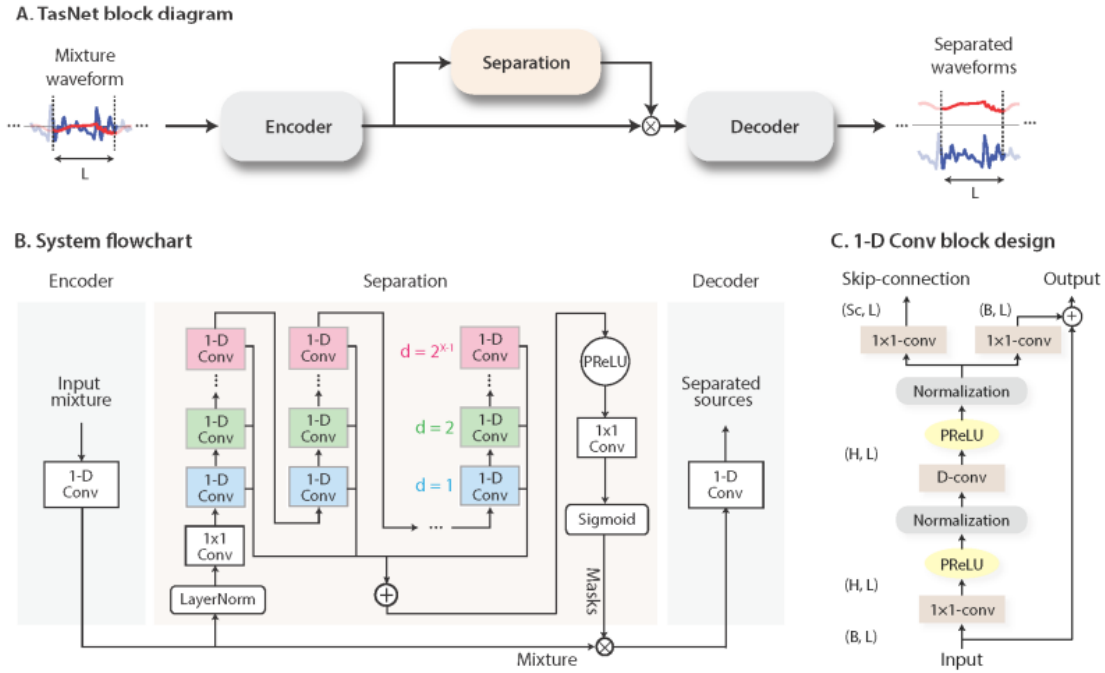


Fig. 1. (A): the block diagram of the TasNet system. An encoder maps a segment of the mixture waveform to a high-dimensional representation and a separation module calculates a multiplicative function (i.e., a mask) for each of the target sources. A decoder reconstructs the source waveforms from the masked features. (B): A flowchart of the proposed system. A 1-D convolutional autoencoder models the waveforms and a temporal convolutional network (TCN) separation module estimates the masks based on the encoder output. Different colors in the 1-D convolutional blocks in TCN denote different dilation factors. (C): The design of 1-D convolutional block. Each block consists of a  $1 \times 1$ -conv operation followed by a depthwise convolution ( $D$ -conv) operation, with nonlinear activation function and normalization added between each two convolution operations. Two linear  $1 \times 1$ -conv blocks serve as the residual path and the skip-connection path respectively.

The model then passes  $w$  through a Temporal Convolution Network (TCN) which is comprised of 1D convolutions with exponentially growing dilation values with skip connections. This ultimately aims to find a mask for each of the  $S$  sources (denoted as  $m_i$ ). Then, for each output source  $i$ , the latent representation of its waveform (denoted as  $D_i$ ) is found by performing an element wise multiplication:  $D_i = w \odot m_i$ .

Then the process finishes by passing  $D_i$  through a decoder consisting of a de-conv using overlap and add to up-sample into a full waveform.

Our version of the Conv-TasNet will be an adaptation of the model presented above (which we denote Mini-Conv-TasNet (MCTN)).

By changing the decoder model we hope to reduce the number of parameters and training time needed, while still being able to extract the fundamental frequencies and loudness of each source for every timestep utilizing the mask for source separation.

We adapted an implementation of the Conv-TasNet for TensorFlow<sup>24</sup>.

Since outputting the whole waveform was no longer needed, we took inspiration from DDSP’s “unsupervised” model, and after applying the mask to get a latent representation of the waveform, we pass it twice through an MLP consisting of a dense layer, batchnorm and an activation of RELU.

One MLP has 128 output units, each representing a single MIDI note. Then these are passed through a softplus layer to generate a probability distribution, a weighted average is calculated and the result MIDI note is converted to frequency in Hz<sup>25</sup>.

The second has just 1 output unit, representing the extracted loudness.

We extract the ground-truth fundamental frequency  $F_0^i(t)$  and loudness  $l^i(t)$  for each source using a pre-trained CREPE and a deterministic DSP method (as referenced [above](#)).

For the training we used  $L_1$  loss between the ground-truth and the model’s output.

The original paper cross-validated many hyperparameter combination. We used similar value with  $N$  and  $R$  reduced due to the time it takes to train such model.

We used  $N = 128$ ,  $B = 256$ ,  $H = 512$ ,  $P = 3$ ,  $X = 8$ , and  $R = 1$ .

HYPERPARAMETERS OF THE NETWORK.

Symbol	Description
$N$	Number of filters in autoencoder
$L$	Length of the filters (in samples)
$B$	Number of channels in bottleneck and the residual paths’ $1 \times 1$ -conv blocks
$H$	Number of channels in convolutional blocks
$P$	Kernel size in convolutional blocks
$X$	Number of convolutional blocks in each repeat
$R$	Number of repeats

Figure 5. Hyperparameters of the Conv-TasNet model

For  $L$ , we used 64, while also changing the stride of the encoder’s convolution to  $L$ .

This change was done to ensure the framerate of the output matches the ground-truths extracted using CREPE (250 Hz.), and therefore the latent representation of waveforms did not look at overlapping sections.

<sup>24</sup> [Implementation of a specific settings](#) of Conv-TasNet by GanTheory.

<sup>25</sup> See function `_compute_f0_hz` in `tasnet_tf.py`

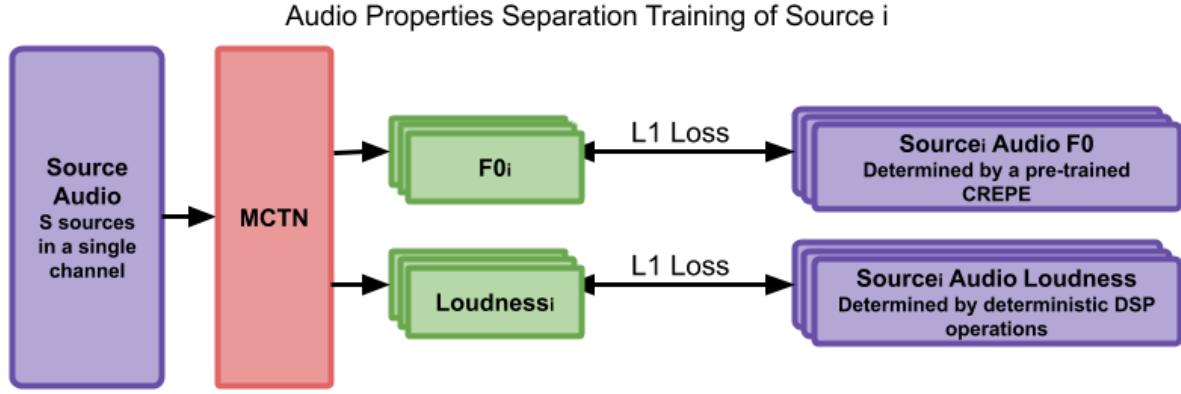


Figure 6. MTCN pre-training model

The training was done using Adam optimizer, with a starting learning rate of 0.001, which decreased by 20% every 3 consecutive epochs without improvement.

### 3.3 Our music source separation model

Starting with an input audio featuring multiple sources, we pass it through the pre-trained MTCN to extract  $F_0^i(t)$  and  $l^i(t)$  for each  $i \in [1, \dots, S]$ , and then pass it to the matching pre-trained decoder which will output its instrument's synthesized audio:

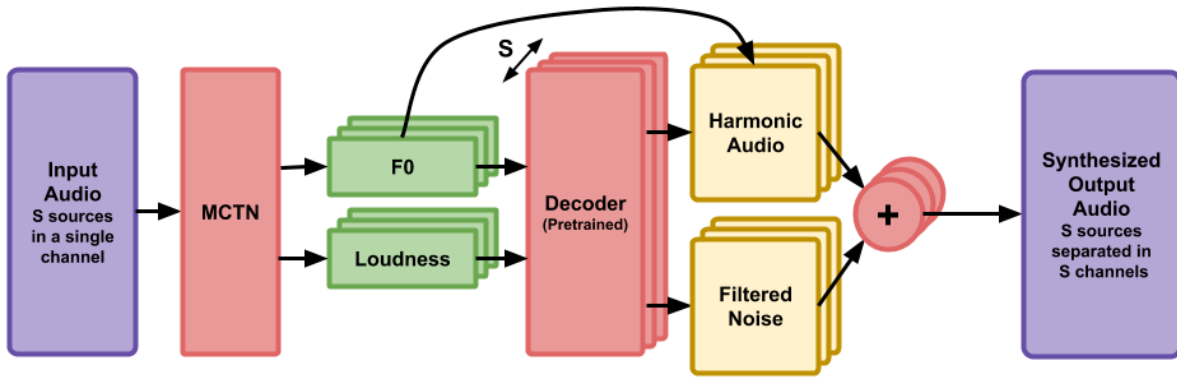


Figure 7. Audio source separation model

We believe that, compared to other audio source separation models, the small parameter size of the DDSP autoencoder, and the relatively simpler task of extracting fundamental frequency and loudness from multiple sources will result in a simpler training, and overall a successful audio separation.

### 3.4 Metrics and evaluation

**Loudness  $L_1$  distance:** The loudness is extracted from the synthesized audio using a deterministic DSP process.  $L_1$  distance is computed against the input's loudness (ground truth). A better model should produce lower  $L_1$  distances, indicating synthesized audio closely matches loudness of the original audio.

It is worth mentioning that, since the audio consists of multiple sources, this extracted loudness vector contains the average loudness of all sources in each time step.

**Multi-scale spectral metric:** A metric presented in the DDSP paper<sup>26</sup> that tries to calculate the distance between two audio waveforms and to hint about their similarity, since usually point-wise loss on raw waveforms is not ideal because two perceptually identical audio samples may have distinct waveforms, while similar waveforms may sound very different.

<sup>26</sup> [DDSP paper](#), section 4.2.1

To calculate this loss, we compute the (magnitude) spectrogram of the original and synthesized audio  $S_i$  and  $\hat{S}_i$ , with a given FFT size  $i$ , and define:

$$L_i = \|S_i - \hat{S}_i\|_1 + \alpha \|\log S_i - \log \hat{S}_i\|_1$$

Where  $\alpha$  is a weighting term set to 1.0 in our experiments. The total reconstruction loss is then the sum of all spectral losses  $L_{MSS} = \sum_i L_i$ .

This metric aims to match the distance of two audio tracks similar to how they are as perceived by a human ear.

**Source-to-distortion ratio (SDR):** An abuse of the name for scale-invariant source to noise ratio (SI-SNR) which is a method commonly used as an evaluation metric for source separation replacing the “standard” source-to-distortion ratio. Given estimated and original sources  $\hat{s}$  and  $s$ , it is defined as:

$$\begin{cases} s_{target} := \frac{\langle \hat{s}, s \rangle s}{\|s\|^2} \\ e_{noise} := \hat{s} - s_{target} \\ \text{SI-SNR} := 10 \log_{10} \frac{\|s_{target}\|^2}{\|e_{noise}\|^2} \end{cases}$$

And is given in *dB*. Higher values mean the estimated is closer to the original source.

### 3.5 Alternative solutions

We investigated a few other solutions for the audio separation task, as detailed below:

1. Having a similar model to the one presented with the MCTN, without modifying the architecture of the Conv-TasNet.  
This suggestion relies on the hypothesis stated earlier in the document – that the basic frequency and the loudness can be extracted from noisier sound. So by only reducing the dimensionality parameters of the Conv-TasNet (such as the hyperparameters  $N$ ,  $R$ ,  $B$ , and  $H$ ), we will be able to produce a noisier result and pass the output of every source to a pretrained CREPE and a loudness extractor - just as done in the DDSP paper’s solo violin example.  
Since Conv-TasNet was shown to work on musical source separation, we hope to achieve a similar result with a model that still has less parameters than the original. The rest of the network is as suggested above – the fundamental frequency and loudness of every source will go to a pretrained DDSP decoder of the target musical instrument.
2. Having a single end-to-end model, without pre-training, that is inspired by the second demo autoencoder model of the DDSP library:  
Taking a similar approach to ideas used in image processing, we suggest using convolutions to create another dimension of size  $S$ , along with utilizing the  $z_i(t)$  generated for each  $i$  in this dimension to encode the necessary details in the latent space.  
This fits our idea of source separation as the encoder provides some sort of “memory” by using a GRU.  
Treating each  $i$  in the new dimension as an output source, we will calculate the loss for each original source against it, expecting to learn accordingly.



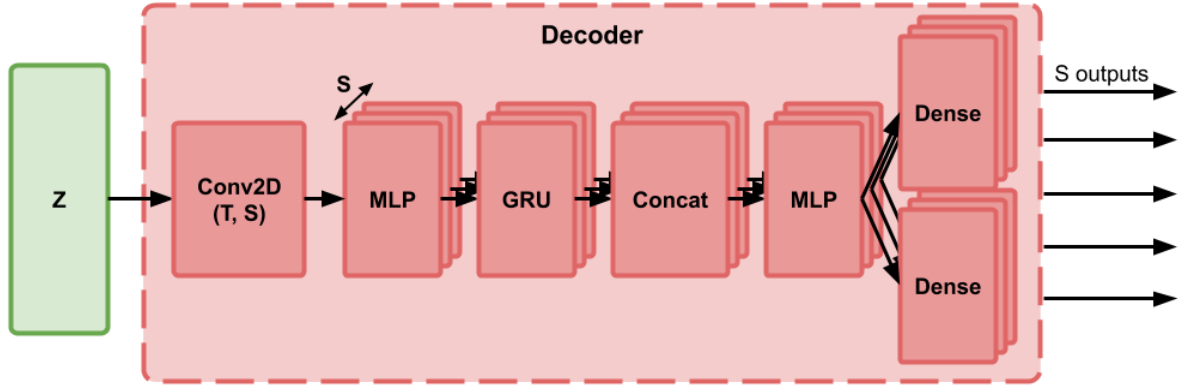


Figure 8. Suggested z decoder diagram

Because we cannot rely on anything to separate the required  $f_0^i(t)$  and  $l^i(t)$  all the work is put on the latent encoding, larger network elements will be then required in order to learn something meaningful, increasing the parameter size considerably. A similar issue was discovered with the second demo in the DDSP paper, as a very large residual net was required to effectively learn to reconstruct audio from single source.

Model	Parameters
WaveNet Autoencoder (Engel et al., 2017)	75M
WaveRNN (Hantrakul et al., 2019)	23M
GANSynth (Engel et al., 2019)	15M
DDSP Autoencoder (Unsupervised)	12M
DDSP Autoencoder (Supervised, NSynth)	7M
DDSP Autoencoder (Supervised, Solo Violin)	6M
DDSP Autoencoder Tiny (Supervised, Solo Violin)	0.24M

Figure 9. Parameter comparison of the DDSP model

We will obviously require larger parameter size for the multiple sources, therefore much more parameters than the Conv-TasNet-gLN network will be required, without any promise it will function any better.

3. Taking inspiration from adversarial models and a paper<sup>27</sup> discussing the following method, we also had a (very unpolished) idea of using a combination of variational autoencoder with the generator/discriminator pair of GANs, combined with DDSP elements for audio source separation.

Passing the original mixed audio track (containing multiple sources) through an encoder, we will get a latent representation of it.

The generator will receive this representation. Its generated output will then be split into  $S$  parts and into  $S$  different discriminator, which will then compare and decide which input is real or fake. Both the generator and the encoder will learn from the loss calculated using the discriminator output.

The model will also incorporate DDSP elements in it, for example, adding  $S$  additive synthesizers which will be the actual output of the generator will remove the need for manual split, while also requiring the generator to solely learn fundamental frequency extraction.

<sup>27</sup> Anders Boesen Lindbo Larsen, Søren Kaae Sønderby, Hugo Larochelle, Ole Winther, [Autoencoding beyond pixels using a learned similarity metric](#).

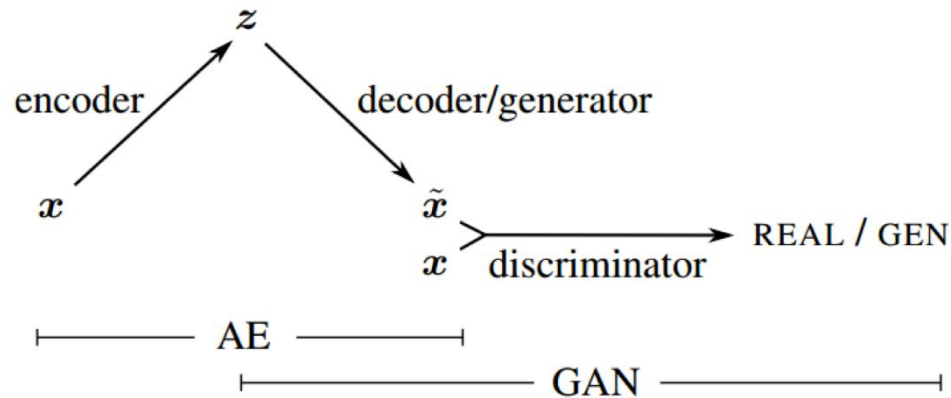


Figure 10. AE-GAN model

Since there is no promise that this model will do any better than the deterministic autoencoder used in the original paper, along with seeing that the VAE-GAN model is not thoroughly explored, we decided to go with a simpler solution (which is also already implemented).

### 3.6 Project sources structure

In this subsection we will detail the structure of our code in the `src/` directory:

- The directory `ddsp/` contains the code of the DDSP library along with its preprocessor.  
No changes were done to the original code.
- The directory `TasNet/` contains the code of the MCTN, where the main differences are as detailed in the [section above](#) – changes to the decoder, some hyperparameters and general refactoring of the original code (that was originally written for TensorFlow v1).
- The directory `results/` contains the output of all experiments held, with its subdirectories and file names detailing the specific experiment.
- The directory `training/` contains all scripts that were used in the pre-training of the models.
  - o The file `our_solo_instrument.gin` – a base gin file for the DDSP’s autoencoder training.
  - o The files `py-sbatch_{bass, drums, vocals}` – bash files that were used for training. They ran DDSP autoencoder training using the gin file above, with a path to the relevant `.tfrecord` file in our dataset directory.
  - o The files `py-sbatch_{bass, drums, vocals}_resynth` – bash files that were used to resynthesize the relevant instrument using the `resynth.py` evaluation file.
- The file `resynth.py` that runs an evaluation by passing a single instrument audio file through the pre-trained DDSP autoencoder.
- The file `eval-mctn.py` that runs an evaluation by passing a given audio file through the pre-trained MCTN model.
- The file `eval_ddsp.py` that runs an evaluation by passing extracted fundamental frequencies and loudness of each source (for example, those extracted by the MCTN model) through each of the pre-trained DDSP autoencoders.

Notes:

- We also had a `dataset/` directory and few other data directories that were not included in this project’s files due to their large size. It contained the raw mixed and separated audio files, along with the `.tfrecords` files that were generated by preprocessing the audios files.
- Most of the scripts used to train and evaluate the model use relative imports and therefore should be executed when the current working directory is `src/`

## 4 Results

As our model consists of two primary parts that are trained separately – we have separated the experiments we held:

1. Testing the quality of the audio resynthesized by the DDSP decoders.
2. Testing the fundamental frequency and loudness extracted by the MCTN.
3. Testing the quality of the resynthesized separated audio from the full assembled model.

All experiments were done solely on audios containing the expected instruments, as the DDSP decoders learn to resynthesize these specifically. Therefore, applying our model on an audio with any other instrument instead of or in addition the expected ones will just not work; the MCTN model expects to split into exactly 3 sources, and learned its mask on the same artist playing specific instruments. In addition, there are exactly 3 DDSP autoencoders, and any given input will be converted to sound like the instrument they were trained on.

Therefore, evaluating our results compared to similar papers or by automatic metrics will not be productive – the results would not mean anything. This also shows in similar works in the field – the output is compared on the same dataset that was earlier split for testing purposes.

Thus, we conducted experiments to test the metrics against an audio of the same artists that the model did not learn on.

For comparison against other works, because they did not deal with fundamental frequencies and loudness, we had no other option but to compare the average SDR of the outputs and the number of parameters of the model.







### 4.1 Evaluating DDSP resynthesized audio quality

In these experiments we resynthesized each instrument by first extracting its audio features ( $f_0(t)$  and  $l(t)$ ) using a pre-trained CREPE model and a deterministic DSP method, respectively. The extraction frequency is 250Hz (4ms) as used in our full model.

We evaluate the quality of the synthesized sound by the average  $L_1$  distance on the audio features separately, and the average Multi-Scale spectral metric.

The metrics are an average on the whole song.

(Note – all samples does not work from the PDF, sadly, therefore, there's a [link to the git](#) containing these examples in the `src/results/` folder)

	$l(t)$ $L_1$	$f_0(t)$ $L_1$	Multi-scale spectral	Resynth sample	Original sample
Vocals	5.136	519.352	7.617	 vocals_resynth_13200.wav	 original_vocals.wav
Drums	3.511	55.146	7.687	 drums_resynth.wav	 original_drums.wav
Bass	9.287	189.225	8.176	 bass_resynth_11100.wav	 original_bass.wav
Violin <sup>28</sup>	0.07	0.02	-		

<sup>28</sup> The violin accuracy is taken from the original DDSP paper, section 4.1, table 1.

Table 1. Evaluation of pretrained DDSP decoders

The resynthesized bass sounds similar to the input bass in its melody but is missing the strumming artifacts. Also, it does not sound clear as the original.

Similarly, while the drumming effect of the drums is evident in the resynthesized audio, it is noisy, and the sounds of drums are “smoothed”.

One would think the low value of the Multi-scale spectral metric of the resynthesized vocals (which in its training reached the value recommended by the authors of the DDSP paper<sup>29</sup>), means that the autoencoder did well. However, the voice in the resynthesized sample evidently does not sound human, and this seems to reflect in the  $f_0(t)$  distance. We rule out overfitting as the cause, as testing resynthesized trained data from a model with higher or lower Multi-scale spectral loss sounds even less human.

By listening carefully to the resynthesized vocals, we noticed that some consonants are not synthesized at all – especially throat-originated. We believe this is because they are produced by an inharmonic sound that our harmonic additive synthesizer cannot reproduce.

Furthermore, it seems that the harmonic frequency distribution output is not quite perfect – As can be seen in the spectrogram below as an example - the lower frequencies of the vocals should have larger weight in the synthesized audio, and in general the strength distribution seems a bit uniform compared to the original vocals audio. We believe this contributes to the robotic feeling of the sound.

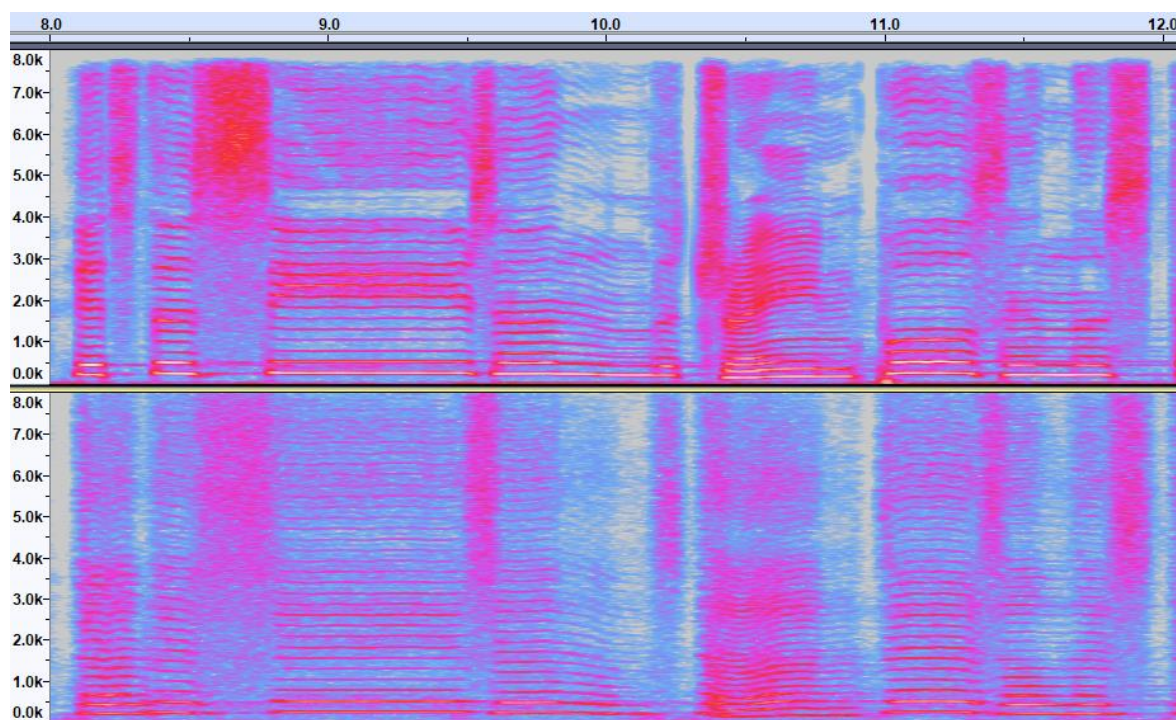


Figure 11. Spectrogram comparison between the original vocals (upper) with the resynthesized vocals (lower). Demonstrating that the synthesized vocals frequency strength distribution is too uniform.

In conclusion, it is clear that the our autoencoders were not up-to-par with the one presented in the paper.

We assume that due to the general difficulty in learning these sources (in opposed to the violin), our data size was not enough (though surpassing the suggested amount), removal of the reverb training, along with the fewer resources we used due to time limitations (the

<sup>29</sup> In [this](#) colab notebook demo for autoencoder training, under the Training Notes section: “Models typically perform well when the loss drops to the range of ~4.5-5.0.”

violin autoencoder was trained for 20 hours on a stronger GPU) we got this sub-par result of the autoencoders.

This draws an interesting result: while DSP elements, as presented the DDSP library, work well for learning specific type of sources (such as strings and wind instruments), they lack (at least those currently implemented) the necessary expressiveness required for others (such as vocals and bass).

## 4.2 Evaluating MCTN separation quality

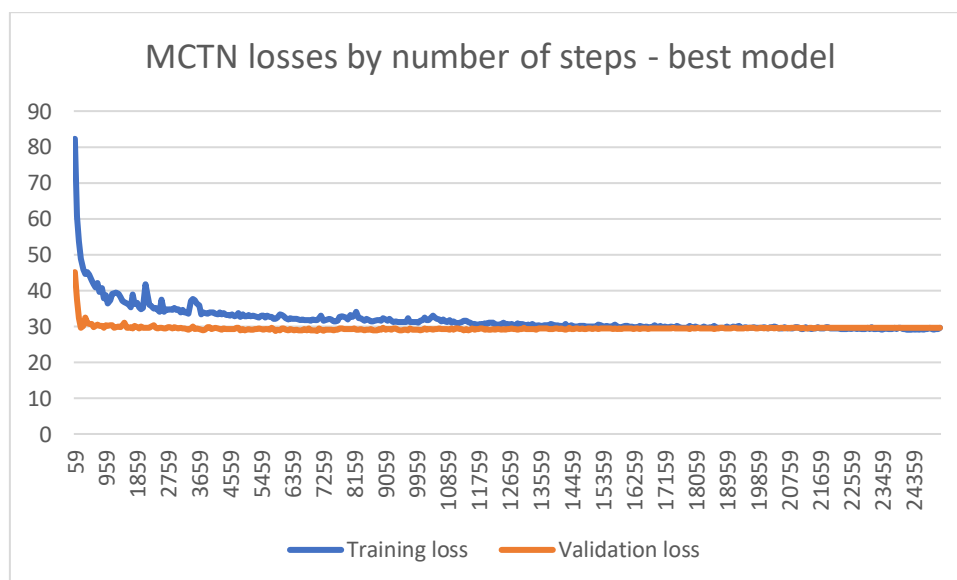
We evaluate the MCTN model source separation quality by checking the average  $L_1$  distance of the audio features ( $f_0(t)$  and  $l(t)$ ) separately, for each musical instrument, and the average weighted loss we used for training  $((1 - \alpha) \cdot L_1(l(t)) + \alpha \cdot L_1(f_0(t)))$  for  $\alpha = 0.1$

	$l(t)$ $L_1$	$f_0(t)$ $L_1$	W. Loss
Vocals	17.976	428.326	59.011
Drums	10.248	95.976	18.821
Bass	7.809	165.163	23.545

Table 2. Evaluation of MCTN separation

These experiments proved what we already expected from the training – these high losses show that the MCTN model we adapted did not manage to extract the audio features reasonably from the audio and possibly caused the whole model to fail.

We took a lot of time investigating the MCTN model: extensively searching hyperparameters, separating it into two MCTNs where each separates one of the audio features, and changing the decoder's MLP layer model but never managed get better results than an average validation loss of 30 (and the final version of the adaptation is as detailed [above](#)).



This led us to believe a different approach to the Conv-TasNet adapted model is in order, such as mentioned in the [additional solutions section](#) (option 1). Unfortunately, due to time constraints, we did not get to try and implement it.

### 4.3 Evaluating separated synthesized audio quality

After passing a test audio file from the same artist containing our 3 sources through the MCTN, in this experiment we look into comparing the final, separated synthesized sources with the original ones.

As above, we compared their audio features  $L_1$  distance, and the multi-scale spectral metric.




	$l(t)$ $L_1$	$f_0(t)$ $L_1$	Multi-scale spectral metric	Separated sample	Original sample
Vocals	18.047	449.106	14.658	 sep_vocals.wav	See above
Drums	9.979	141.221	10.642	 sep_drums.wav	See above
Bass	9.985	204.656	8.421	 sep_bass.wav	See above

Table 3. Evaluation of the full separation model

Comparison of SDR in dB with state-of-the-art models for source separation done on the same database:

Method	Bass	Drums	Vocals
	SDR (dB)		
<b>Our model</b>	<b>-5.681</b>	<b>-1.14</b>	<b>-7.512</b>
MMDense-LSTM <sup>30</sup>	3.73	5.46	6.31
MMDenseNet <sup>31</sup>	3.91	5.37	6.00
BLEND <sup>32</sup>	2.98	4.13	5.23

Table 4. SDR comparison with other works

From these comparisons, it is clear (and expected) that our model sadly failed to separate and resynthesize the original audio in any reasonable sense.

Below is the mixed audio generated back from the separated sources for your enjoyment. We can at least take comfort from the fact that (if you try hard) it does have similar sound features to the original audio, such as quiet or instrumental parts.



sep\_mixed.wav

Below is a comparison of the number of trainable parameters of our model compared to the works compared to before.

<sup>30</sup> Taken from the relevant [MMDense-LSTM paper](#)., section 3.3, table 3 (comparison on DSD100 dataset).

<sup>31</sup> See footnote 30, taken from same table. [Relevant paper about MMDense](#).

<sup>32</sup> See footnote 30, taken from same table. [Relevant paper about BLEND2](#).



Model	N. of parameters ( $\times 10^6$ )
Conv-TasNet <sup>33</sup>	5.1
MMDense-LSTM <sup>34</sup>	1.22
<b>Our model – DDSP-based, MCTN<sup>35</sup></b>	<b>16.9</b>
BLEND2 <sup>36</sup>	30.36
LSTM-TasNet <sup>37</sup>	32

Table 5. Comparison of number of parameters with similar works

While the total number of parameters in our model is larger than the state-of-the-art models, it is important to mention that since our model is comprised from 4 sub-models (1 MCTN and 3 DDSP autoencoders) that do not rely on each other for training, and therefore can be trained parallelly, in effect requiring a simultaneous training of up to just  $4.8 (\times 10^6)$  parameters, lower than most comparable models.

In a final conclusion, our model did not show that the reduction of the ASS problem into a “audio features extraction” problem eases the separation. Furthermore, the synthesizer which we relied on its high fidelity was not as good as we thought, and not yet capable of synthesizing a realistic natural sound. The combination of the two inadequate parts created a model generating unsatisfactory results.

However, it is still possible that there are other network architectures that may succeed in separating the audio features with greater ease than separating entire signals (such as described in the [alternative solutions section](#)), and can be combined with DDSP’s elements to create a relatively lightweight model that will separate sources well.

<sup>33</sup> Taken from the relevant [Conv-TasNet paper](#), section IV.c, table IV (comparison on WSJ0-2Mix dataset).

<sup>34</sup> Taken from the relevant [MMDense-LSTM paper](#), section 3.3, table 4 (comparison on MUSDB18 dataset).

<sup>35</sup> 2.5m (MCTN) +  $3 \times 4.8$ m (Vocals, bass and drums DDSP autoencoder)

<sup>36</sup> See footnotes 34 and 32.

<sup>37</sup> A version of TasNet that uses LSTM internally. Data taken from the same table pointed at by footnote 33.