

Single channel speech / music separation

Description	Training a neural network to separate speech from music.
Period	Fall 2021
Languages & Libs	Python, Keras
Tags	Signal Processing, FFT



Humans are naturally capable of separating an object from the background of an image, or speech from music on an audio clip. Photo editing is an easy task, but personally I don't know how to remove music from the background. A first approach would be to use band-pass filters, but it wouldn't result in a satisfactory end result since there is so much overlap between the frequencies. This article describes a supervised learning approach on solving this problem.

The used dataset consists of four separate podcasts (8.4 hours in total) and four different genres of music (ideally without any lyrics, 7.1 hours in total). The used podcasts were:

- Tom Segura & Bert Kreischer, 1.2 h: [Ep. 105 | 2 Bears, 1 Cave w/ Tom Segura & Bert Kreischer](#)
- Lex Fridman, 2.4 h: [Jimmy Pedro: Judo and the Forging of Champions | Lex Fridman Podcast #236](#)
- Jordan Peterson, 2.2 h: [Death, Meaning, and the Power of the Invisible World | Clay Routledge | The JBP Podcast S4: E54](#)
- Joe Rogan, 2.6 h: [Joe Rogan Experience #1169 - Elon Musk](#)

The used music genres and clips were:

- Classical music, 3.5 h: [The Best of Classical Music Mozart, Beethoven, Bach, Chopin, Vivaldi Most Famous Classic Pieces](#)
- House music, 1 h: [Pavel Khvaleev pres. PARAFRAME Inhale & Exhale Live Show @ Radio Intense / Progressive House DJ mix](#)
- Liquid Drum & Bass, 1.3 h: [Liquid Drum and Bass Mix #127](#)
- Progressive rock (some lyrics), 1.3 h: [Progressive Rock Mix by Prog Rock Dock - Volume 01](#)

All audio files have a sampling rate of 44.1 kHz, which was downsampled by 1:3 to 14.7 kHz to speed up the network training. They were loaded to Python using [Pydub](#), which is a wrapper for the FFMPEG. The files were split into two second long clips, and were normalized so that their standard deviations are one. The input to the network is a music clip + a weighted random speech clip, where the weight was picked uniformly from the interval of $[0, 1]$. So there are three types of clips (music, speech and combined), and their averaged spectrums are shown in Figure 1.

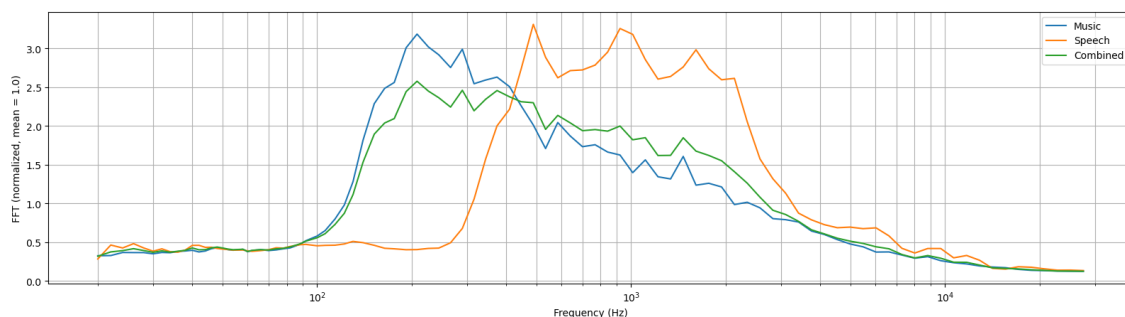


Figure 1: Spectrums of three categories of audio: music, speech and their combination. Obviously the "combined" class is found between the two other classes, since Fourier transform is a linear operation (although the plots are normalized, their mean is one). The music is higher on the 100 - 400 Hz range, and speech is higher between 400 and 3000 Hz.

Splitting the 8.4 hours of podcasts into samples of two seconds produced 15254 clips, of which 20% was used for model validation. The input (1 channel) and expected output (2 channels) is stored in a matrix of shape $15254 \times 29400 \times 3$, which requires only 5.4 GB of memory when stored in float32 format. This fits easily in RAM, but much more would be needed if more different podcasts were included, or the original sampling rate of 44.1 kHz was used. Nevertheless there are about 360 million data points used for training, and the loss is calculated against 720 million data points.

Naturally several network architectures were experimented with, but here just the best one is described. It has three tunable hyperparameters, and they were chosen so that the number of model parameters was approximately 2 million. The network attempts to normalize the incoming signal's volume (the variable stds) and re-scale the output back to the original level, but experiments show that for some reason this isn't working as expected.

```
dim, w, n_iter = 60, 2**6, 3

inp = Input((len_samples, 1))
x = inp

# Normalize the signal for the network, although this doesn't seem to work
# as intended. Experiments show that the network isn't linear, meaning that
# model.predict(X) * k != model.predict(X * k).
stds = K.maximum(0.01, K.std(inp, axis=(1,2)))[:,None,None]
x = x / stds

x = BN()(Conv1D(dim * 2, 16*w+1, activation='elu')(x))
x = BN()(Conv1D(dim-1, w+1, activation='elu')(x))

# Pass the original input to refinement layers for a resnet-like architecture:
c = (inp.shape[1] - x.shape[1]) // 2
x = K.concatenate([x, inp[:,c:-c,:]])

for _ in range(n_iter):
    x = BN()(Conv1D(dim, 2*w+1, activation='relu')(x) + x[:,w:-w,:])

x = BN()(Conv1D(dim // 2, w+1, activation='elu')(x))
x = Conv1D(1, 1, activation='linear')(x) * stds

crop = (inp.shape[1] - x.shape[1]) // 2

# It was evaluated that it was slightly better for the network to generate
# the speech (2nd output) rather than the music (1st output) channel.
model = Model(inp, K.concatenate([inp[:,crop:-crop,:] - x, x]))
```

On a GTX 1080TI each epoch took 13.8 minutes, but upgrading it to a RTX 3070TI (paying +100% over MSRP) reduced this by 58%, bringing it down to 5.8 minutes. Anyhow, the evaluated network was trained with the older GPU, and full convergence took 105 epochs or 24.2 hours! Separate [EarlyStopping](#) callbacks were used for loss and val_loss metrics, and [Adam's](#) learning rate was adjusted down from $10^{-2.5}$ to $10^{-3.5}$ and 10^{-4} automatically by [ReduceLROnPlateau](#). The model was trained to a Mean Squared Error of 0.0585 for train and 0.0663 for validation.

The variance of the reference music signal was 1.0 as expected by the normalization, but for speech it was only 0.33! Since the speech's weight is a random uniform variable with a mean of 0.5, it was expected that the signal's mean was 0.5 as well. It was then noted that the used normalization is $x /= \max(1, x.\text{std}())$, meaning that quiet signals aren't going to be amplified. The same formula was used for music, but it is plausible that a conversation have quiet periods of a few seconds. Summing these two uncorrelated signals together produces the input to the network, with an average variance of 1.33. Comparing this with the MSE loss, it is seen that the model is able to correctly model about 95% of the variance. Example inputs and outputs are shown in figures 2 and 3.

The model's summary is shown below:

Layer (type)	Output Shape	Params	Connected to
input_187 (InputLayer)	[(None, 29400, 1)]	0	
tf.math.reduce_std_15 (TFOpLamb	(None,)	0	input_187[0][0]
tf.math.maximum_12 (TFOpLambda)	(None,)	0	tf.math.reduce_std...
tf.__operators__.getitem_802 (S	(None, 1, 1)	0	tf.math.maximum_12...
tf.math.truediv_10 (TFOpLambda)	(None, 29400, 1)	0	input_187[0][0]
			tf.__operators__.g...
conv1d_547 (Conv1D)	(None, 28376, 120)	123120	tf.math.truediv_10...
batch_normalization_491 (BatchN	(None, 28376, 120)	480	conv1d_547[0][0]
conv1d_548 (Conv1D)	(None, 28312, 59)	460259	batch_normalizatio...
batch_normalization_492 (BatchN	(None, 28312, 59)	236	conv1d_548[0][0]
tf.__operators__.getitem_803 (S	(None, 28312, 1)	0	input_187[0][0]
tf.concat_181 (TFOpLambda)	(None, 28312, 60)	0	batch_normalizatio...
			tf.__operators__.g...
conv1d_549 (Conv1D)	(None, 28184, 60)	464460	tf.concat_181[0][0]
tf.__operators__.getitem_804 (S	(None, 28184, 60)	0	tf.concat_181[0][0]
tf.__operators__.add_150 (TFOpL	(None, 28184, 60)	0	conv1d_549[0][0]
			tf.__operators__.g...
batch_normalization_493 (BatchN	(None, 28184, 60)	240	tf.__operators__.a...
conv1d_550 (Conv1D)	(None, 28056, 60)	464460	batch_normalizatio...
tf.__operators__.getitem_805 (S	(None, 28056, 60)	0	batch_normalizatio...
tf.__operators__.add_151 (TFOpL	(None, 28056, 60)	0	conv1d_550[0][0]
			tf.__operators__.g...
batch_normalization_494 (BatchN	(None, 28056, 60)	240	tf.__operators__.a...
conv1d_551 (Conv1D)	(None, 27928, 60)	464460	batch_normalizatio...
tf.__operators__.getitem_806 (S	(None, 27928, 60)	0	batch_normalizatio...
tf.__operators__.add_152 (TFOpL	(None, 27928, 60)	0	conv1d_551[0][0]
			tf.__operators__.g...
batch_normalization_495 (BatchN	(None, 27928, 60)	240	tf.__operators__.a...
conv1d_552 (Conv1D)	(None, 27864, 30)	117030	batch_normalizatio...
batch_normalization_496 (BatchN	(None, 27864, 30)	120	conv1d_552[0][0]
conv1d_553 (Conv1D)	(None, 27864, 1)	31	batch_normalizatio...
tf.__operators__.getitem_807 (S	(None, 27864, 1)	0	input_187[0][0]
tf.math.multiply_22 (TFOpLambda	(None, 27864, 1)	0	conv1d_553[0][0]
			tf.__operators__.g...
tf.math.subtract_91 (TFOpLambda	(None, 27864, 1)	0	tf.__operators__.g...
			tf.math.multiply_22[
tf.concat_182 (TFOpLambda)	(None, 27864, 2)	0	tf.math.subtract_91[
			tf.math.multiply_22[
=====			
Total params: 2,095,376			
Trainable params: 2,094,598			
Non-trainable params: 778			

The used error metric was just the plain MSE, which doesn't correspond well on how the generated signal sounds to a human ear. A better standard would be the [Perceptual Evaluation of Audio Quality](#) or [... Speech Quality](#). The latter is implemented in a Python library [python-pesq](#), which could be used for model evaluation. But since it isn't differentiable, it cannot be used as the model's loss function.

There is also a [Deep Perceptual Audio Metric \(DPAM\)](#) ([Arxiv](#)) metric, which is actually differentiable! However it wasn't tested, mainly because it doesn't support Tensorflow 2.x and it requires a different sampling rate (22050 Hz, all experiments were done with $44100 / 3 = 14700$ Hz).

Yet another approach would be to train a network to calculate the signal's [spectrogram](#), and then use it as a loss (with frozen weights) in signal reconstruction. I have no idea whether it would work as a stand-alone loss, or it should be used alongside with MSE. Intuition tells me that if the spectrogram matches, then it should sound good regardless of phase shift etc.

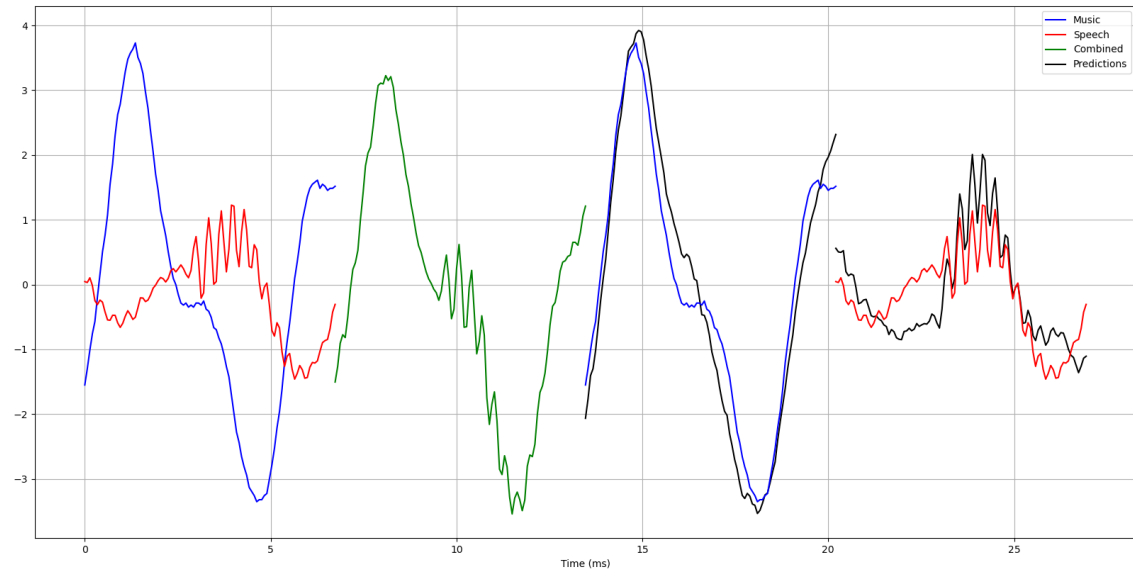


Figure 2: Examples of original audio (music in blue and speech in red), their combined signal (green) and corresponding predictions (black). Each shown signal is only 100 samples long, so at the sampling rate of 14700 Hz it represents only about 7 milliseconds of audio. Note that the network uses a "context" of 768 samples before and after each predicted sample. It is very hard to visually estimate the quality of the produced signal, since it is represented in a time and not in frequency domain. The model generates properly smooth "music" signal, and the high frequencies are allocated to the "speech" output. And there seems to be a low-frequency offset, so the black prediction of "speech" has a rather large MSE afterall.

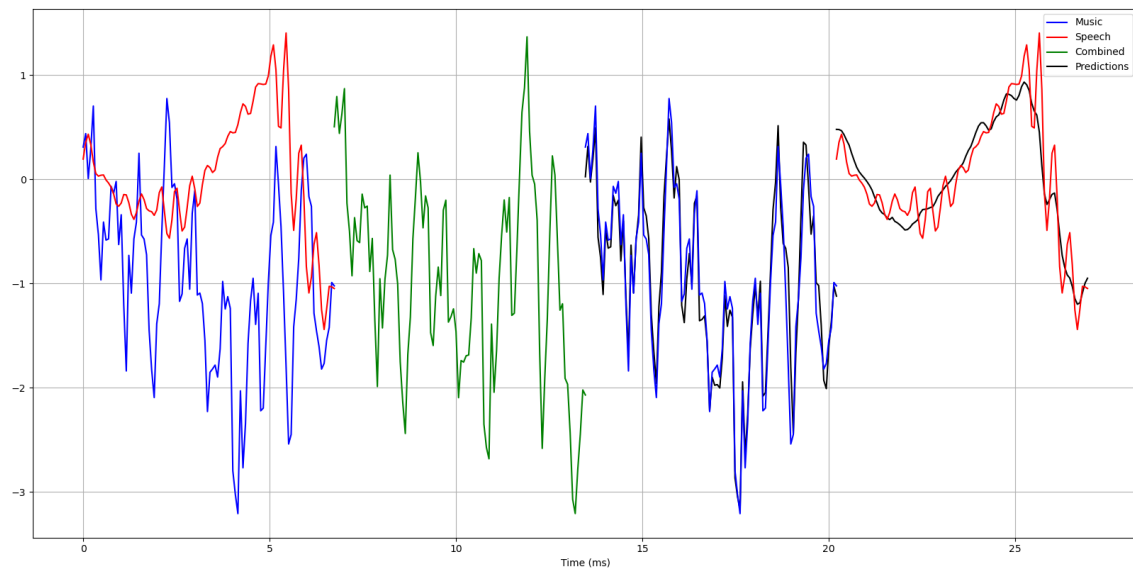


Figure 3: An other example of input and outputs, this time the speech output is missing some higher frequencies and is thus too smooth. Those frequencies are leaked to the music prediction, but they are visually very hard to judge.

White noise (also known as [Gaussian noise](#)) contains an equal amount of all of the frequencies, and is thus often used as a "test input" to a signal processing unit or an algorithm. Figure 4 shows that what percentage of each frequency range gets passed to the "music" or "speech" output. The network isn't constrained to preserve the total energy, so the sum of the two outputs may have a higher variance than the input signal. According to [Accusonus.com](#), human speech spans from a range of 125 Hz to 8000 Hz. Earlier on Figure 1 it was seen that this dataset's speech is distinct from music at around frequencies between 300 and 3000 Hz. But it should be noted that this dataset doesn't have any woman or child speakers. Those would be really good test cases for additional model validation.

The network's response to white noise of varying amplitudes is shown in Figure 4. 80 - 99% of the signal is passed to the "music" output and the rest to the "speech" output. As expected, the speech output is strongest at frequencies between 300 and 1000 Hz. The network was intended to be covariant (or linear) in respect to the input signal's volume, meaning that ideally $\text{predict}(x) \cdot k = \text{predict}(x \cdot k)$. The figure shows that this is not the case. At the moment the reason for this behavior is unknown.

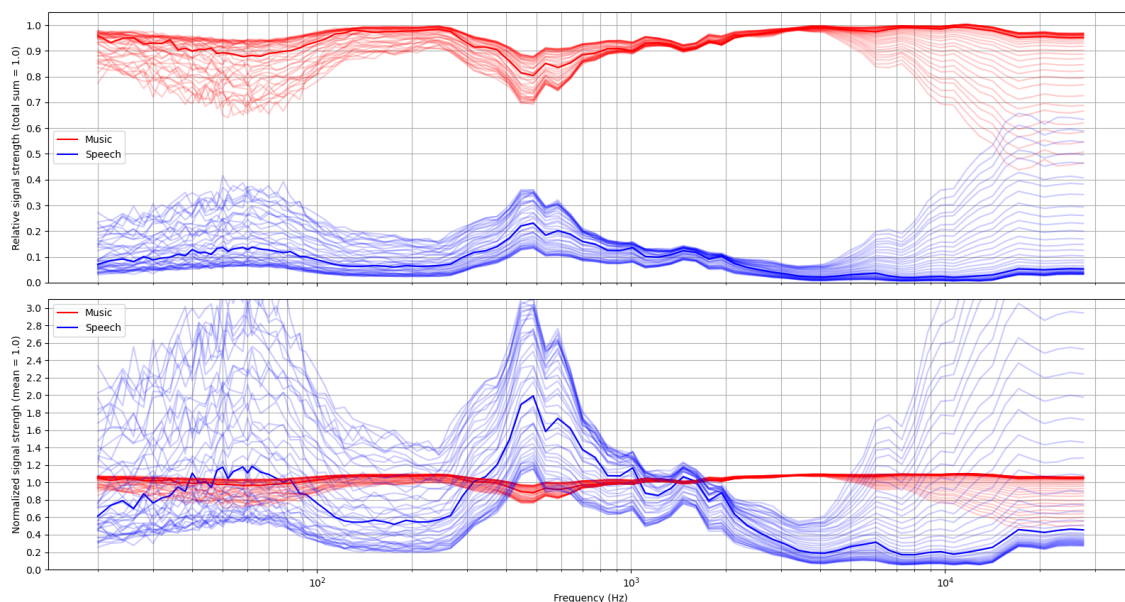


Figure 4: Frequency response of the "music" and "speech" outputs to white noise with varying standard deviations (between 0.6 and 1.6). A scale-invariant (actually covariant!) network wouldn't show this kind of spread on this normalized visualization.

In addition to white noise, the network's signal passing was also studied using pure sinusoidal signals. The result is shown in Figure 5, and it shows some surprising results. Although frequencies between 300 and 3000 Hz are most distinct for speech, they don't get passed to the "speech" output as a stand-alone signal. It seems that the network has learned some patterns in speech, and it only lets those pass through. It is also unclear why the network behaves so oddly at high frequencies, maybe because they aren't much present in the training data.

The input has 29400 samples, but the output has only 27864 ($768 \cdot 2 = 1536$ fewer). This happens because the network uses Conv1D layers without padding, so at each layer the output gets a little smaller. Because of this, extra care has to be taken when processing longer input sequences. The first input is samples from 0 to 29399 (inclusive), but the second input is not from 29400 to 58799! Instead the "crop factor" has to be taken into account, so the second range is shifted left by 768. This gives us $29400 - 768$ to $2 \cdot 29400 - 768 - 1$, or 28632 to 58031. The third range is from $2 \cdot 29400 - 768$ to $3 \cdot 29400 - 2 \cdot 768 - 1$. Also the model doesn't give predictions to the first 768 samples, so the predicted output has to be padded with that many zeros (actually I forgot to do the padding, so audio tracks are a bit out of sync).

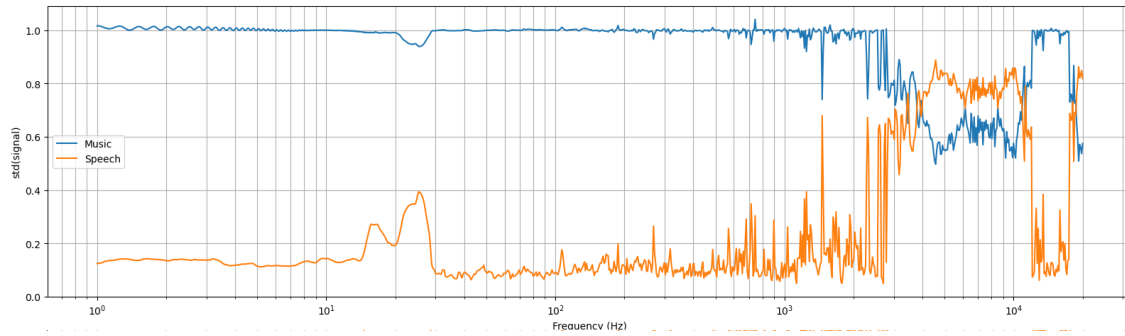


Figure 5: Frequency response of the network to pure sin functions. Interestingly the speech output doesn't let the pure frequencies of 300 - 2000 Hz through, even though it is the distinctive range for speech. An other oddity is that it lets higher frequencies of 3000 - 10000 Hz through, but this wasn't observed with white noise.

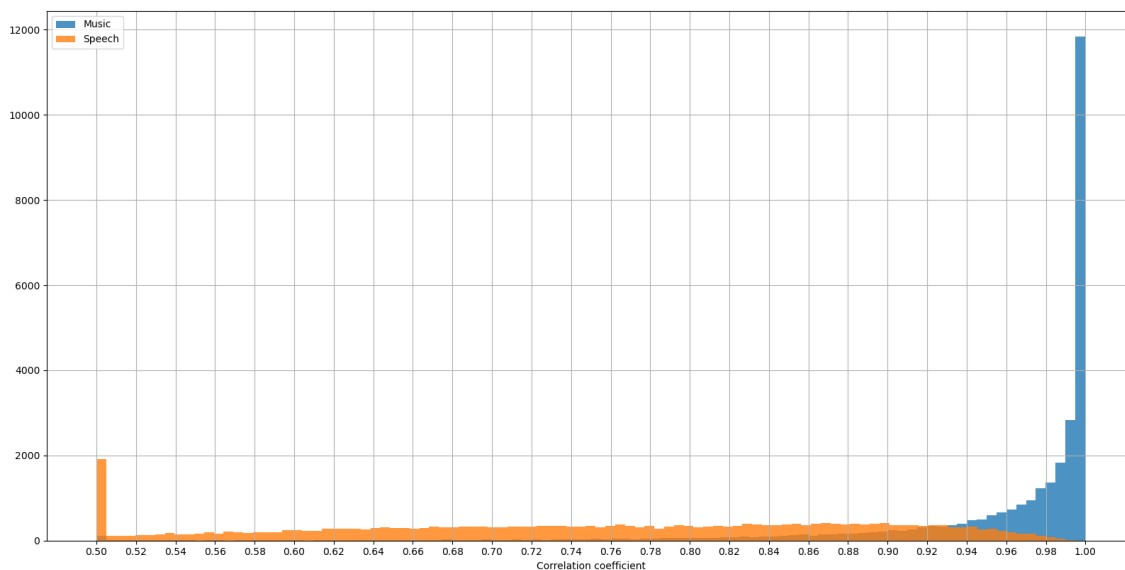


Figure 6: The histogram of correlation coefficient between $\text{reverse}(\text{predict}(x))$ and $\text{predict}(\text{reverse}(x))$ shows that the model isn't time-reversible, especially on the "speech" category. This means that reversing the input results in a different outcome than reversing the output. This is analogous to a CNN classifier not being invariant to the image being flipped horizontally. Maybe music is much more periodic, and the model is quite reversible for those signals.

Examples of "original", "music" and "speech" versions can be heard from an [unlisted youtube video](#). Applied audio processing of each clip is shown in the top left corner. It has short clips from videos [Eric gets overconfident against Hikaru](#), [DON'T WASTE YOUR TIME - Best Motivational Speech Video \(Featuring Eric Thomas\)](#) and [jackass forever | Official Trailer \(2022 Movie\)](#). Obviously these videos haven't been used for training the model, so puts the network into a true test. Sadly the output isn't good audio quality, most notably the "music" track has very audible speech and the "speech" has noticeably lower quality than the original. At least the "speech" doesn't have much audible music.

A better result might be obtained by using more a diverse set of training data, using a more complex model (but overfitting has to be mitigated somehow), and having a better loss function than the plain MSE. Some ideas were already mentioned earlier, but having the [Short-time Fourier transform](#) as a prediction target sounds interesting. Fundamentally the human ear doesn't care about the phase of the signal, only the composition of frequencies matter. But the spectrum calculation has to be differentiable so that Keras is able to propagate loss derivatives through it.