Extracting Single Voices from Musical Ensembles

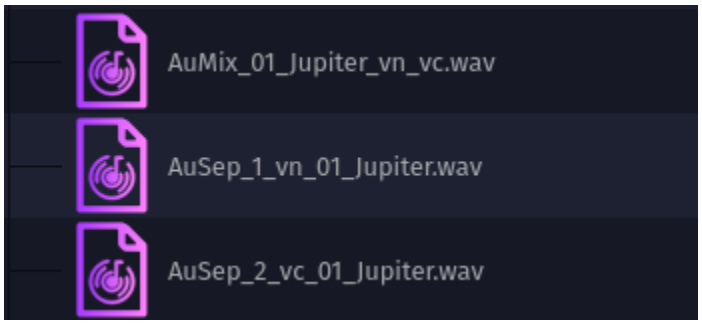
Nikhil Iyer

Problem

Given an audio track of musical ensembles ranging from 2 to 5 instruments, the network is tasked with separating out the audio into tracks that contain only a single solo.

Data

The acquired dataset is composed of recordings of 44 different musical selections. Each player has recorded their part of the musical selection independently, the musical selection is then produced by adding together the individual tracks.



A musical selection with 2 parts (violin + cello) has 3 tracks: only violin, only cello, and mix.

The audio files are in WAV format with a sampling rate of 48 KHz and bit depth of 24 bits, mono channel. The audio mixture is the direct sum of the individual audio tracks without additional individual gains. Individual track numbers are ordered following the score track order.

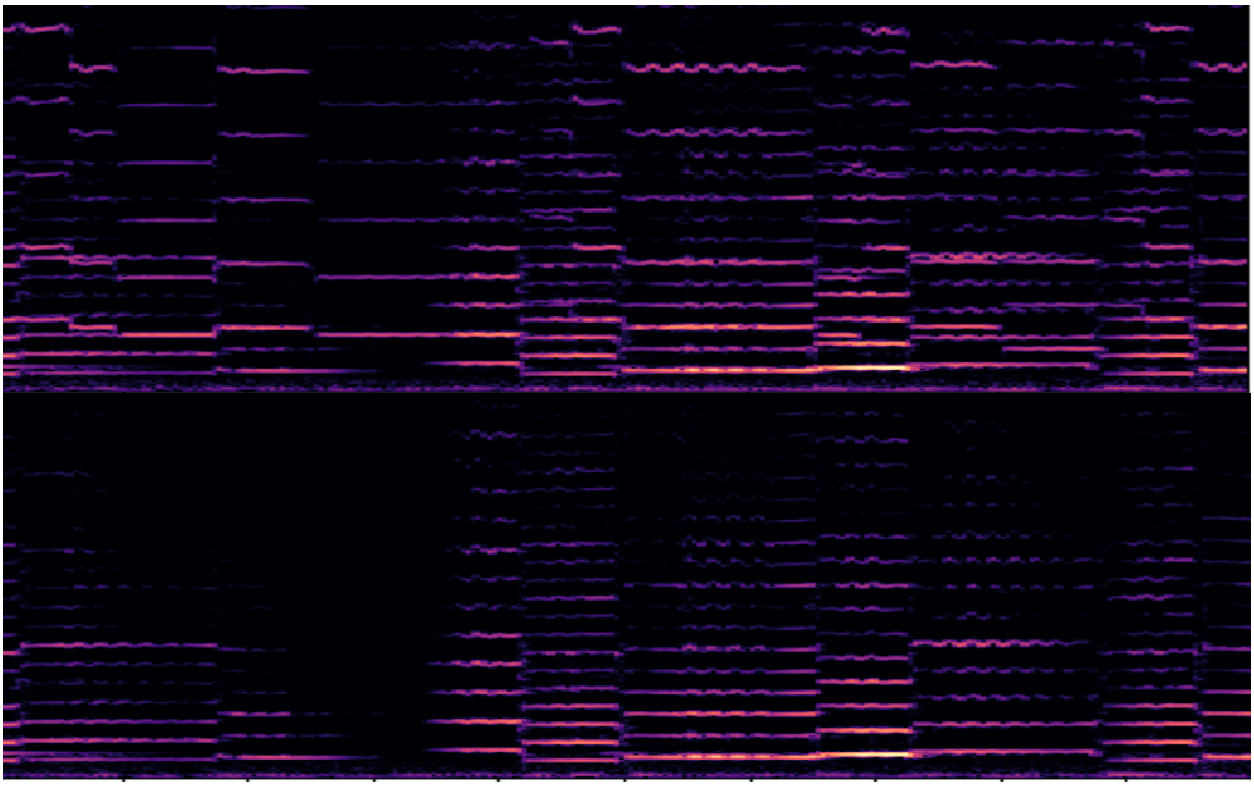
In this form, the entire dataset contains about 11GiB of audio.

Note that a mix need not be complete (containing all parts) to be useful training data. The powerset of tracks was taken to increase the variety of available examples. In this form, the entire dataset contains about 88GiB of audio.

This amount of data is infeasible for my desktop computer to manage in memory and a great deal of time was spent wrangling a working method for applying and caching expensive transformations of the dataset.

First, 48 kHz is an extremely high sampling rate, a 30-second clip contains 1.44 million samples. To be able to fit the neural network in memory the songs were divided into 10-second segments so the input dimension could be reduced. Still, 0.48 million inputs is not tractable on my personal computer. Further research yielded the Fast Fourier Transform (FFT) which takes time-series data from the time-amplitude space to the time-frequency-amplitude space. This helps squeeze the 10-second clips into a subspace without losing much information. If anything the FFT makes separating the two instruments easier (visually at least).

(top) Raw Audio: mix (blue), solo (orange)
(bottom) FFT: mix (upper), solo (lower)



The final transformed dataset was a flattened FFT of downsampled 10-second segments taken from the powerset of instrumental recordings with frequencies greater than 8192Hz truncated.

Final split was 60:20:20::train:test:validation

Benchmarking

Because this is a multivariate regression problem I planned to use linear regression as a benchmark. Unfortunately, the pixel-by-pixel regression actually turned out to be more expensive than the neural network so the attempt was abandoned as it kept crashing my computer [even with how much I had reduced the data] (RES^2 > RES * hidden + hidden * RES).

A really rough benchmark I used was to listen to each piece and try to separate the different lines (hum them) out loud. That gives a general idea of how easy a given example should be for a trained musician. Not ideal but it was the best I could do.

Training and Tuning

For the first neural network, a complete training cycle of 100 epochs took around 9 hours and rendered my computer almost unusable for the duration (RAM Usage at >80% and 1 free CPU core). Additionally, it ran into overflow errors during exponentiation.

At the end of it all. The loss came back as a flat line and the network just spit out noise.
My hypothesis was that all the ReLU nodes entered the negative area and “died” very early on in training. I replaced the sigmoid nodes with tanh (which seems to be more numerically stable) and replaced all the ReLU nodes with ELU.

This trial trained much faster but still yielded a flat line loss.

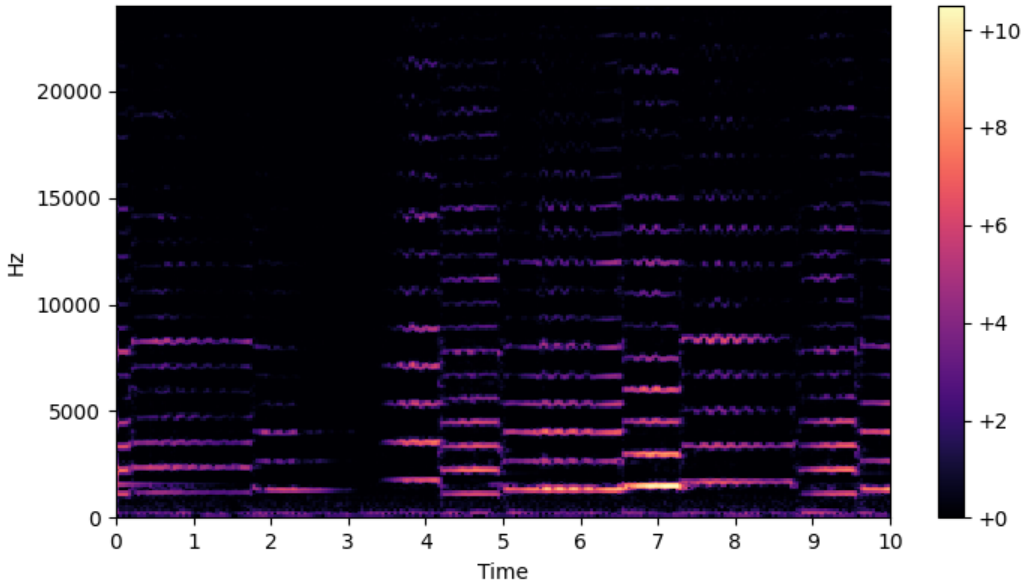
Using a small subset of the data and only 10 epochs of training, I attempted to further tune the hyperparameters.

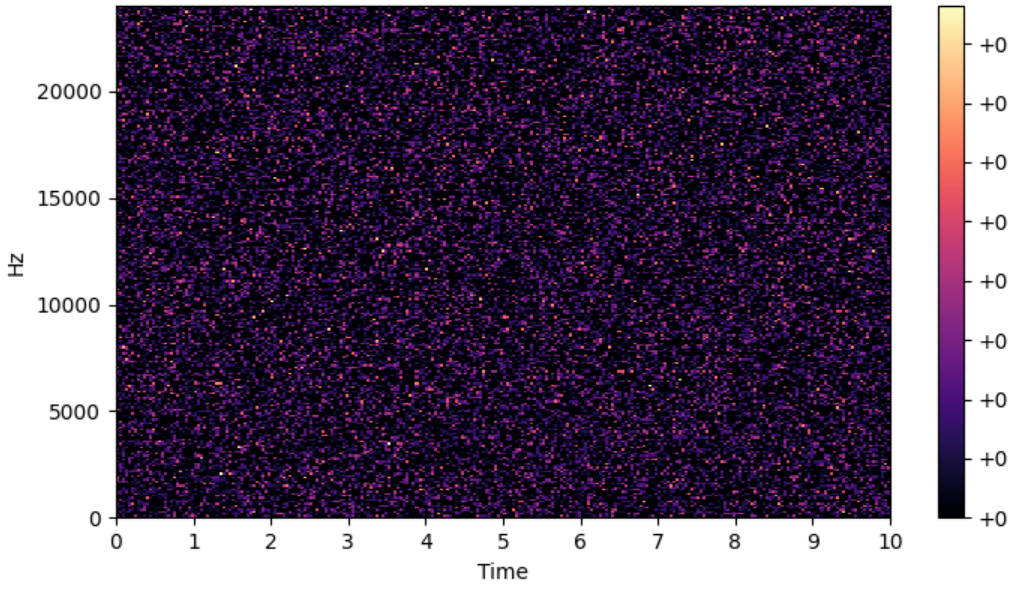
Momentum was implemented. The loss was still a flat line.
The learning rate was increased. The loss was still a flat line.
The depth of the neural network was increased (3 layers -> 6 layers). The loss was still a flat line.
Replaced tanh activations with ELU. Overflow errors returned. The loss was still a flat line.
Reverted.
Increased width of the neural network. Overflow error occurred. The loss was a noisy flat line.
Decreased learning rate and increased momentum. Modified weights initialization (normal -> Kaiming). Infinite loss encountered.
Add weight decay to address overflows.

unable to address overflows
50503816.3046686
350503390.79967576
350502784.00355566
350502014.12379706
350501097.52246934

```
350500048.88350385
350498881.35807323
350497606.6961453
350496235.56699544
350494778.2603429
350494299.1046586
/home/kapow_12/Development/MTH4320/PRJ1/feedforward.py:193: RuntimeWarning: overflow
encountered in exp
  return np.where(z > 0, z, (np.exp(z) - 1))
6.312174572423909e+19
5.033111362407018e+142
/home/kapow_12/Development/MTH4320/PRJ1/feedforward.py:19: RuntimeWarning: overflow
encountered in square
  def __init__(self, input_shape, layers, loss=lambda y_hat, y: np.mean((y-y_hat)**2),
d_loss=lambda y_hat, y : 2*(y_hat - y), l1_penalty = 0, l2_penalty = 0):
inf
/home/kapow_12/Development/MTH4320/PRJ1/feedforward.py:88: RuntimeWarning: overflow
encountered in square
  l2 += np.sum(w ** 2)
nan
```

Overdue no time for further testing best result:





Conclusion

Despite my best efforts, I was unsuccessful in producing a functional neural network to address the task. This network would likely benefit from convolutional layers. The network was too small to accurately model the distribution.