Source Audio

By Dominic Vian

An audio source separation application that
allows you to produce midi files from the extracted audio

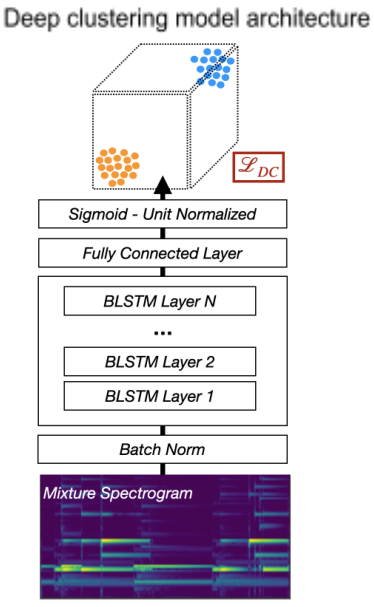
https://youtu.be/-065Bblagzs

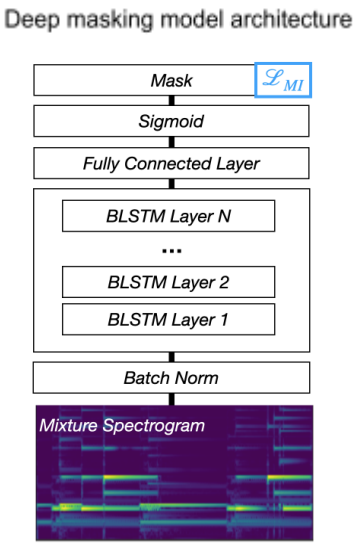
Introduction

My project delves into the utilization of audio source separation and audio-to-midi conversion techniques. The goal was to create a user-friendly application that empowers individuals to discover and experiment with the building blocks of their favourite songs, whether for personal curiosity or artistic expression. The inclusion of an audio-to-midi converter allows for even greater possibilities, enabling users to seamlessly repurpose their extracted audio into new creations. By making this technology accessible to anyone, regardless of musical background, we are unlocking the potential for boundless imagination

Concept

When I started this project I was researching audio feature extraction, a means of analysing audio and extracting information like tempo and spectral features that can be used for tasks like genre classification or speech recognition. It was during my research that I came across the concept of audio source separation and programs like 'LALALA.AI' (https://www.lalal.ai/). The idea occurred to me that this could potentially serve as a valuable tool for remixing purposes. I found that existing programs seemed to be very research-based and I wanted to build something that promotes creativity. I wanted to be able to easily extract the midi information of the audio tracks so that I can use it with my favourite synthesisers, instantly transforming the act of repurposing a track into a far more fascinating and dynamic process, as opposed to the mere importation of audio. Although the audio-to-midi algorithm is a different ballpark from the source separation, I wanted to integrate it into my app so that even non-musicians could easily start creating using the extracted audio.
I was intrigued by the many different ways you can approach audio source separation, such as deep mask estimation, deep audio estimation and deep clustering. So I set off researching libraries that would let me explore these techniques.





Technical Implementation

I decided to build my project as a python application, as this would allow for easy local running and give me a vast toolset to work with. At first, I decided I wanted to try and leverage a pre-trained model using Keras to see what results would be produced. I found ‘Open-Unmix’, a deep neural network implementation for audio source separation that advertised itself to be state-of-the-art and easy to use, however, I could not get it working in my environment, even just to perform a simple separation task. I decided to do some more research and look for another option. The next library I found was ‘Nussl’. This library prided itself on the fact that it was flexible and easy to build and train yourself. Nussl provides a multitude of architectures, such as deep clustering and masking, and contains the implementations of lots of different source separation algorithms such as spatialization algorithms, vocal melody extraction, and timbre clustering. I spent a long time getting to grips with how to handle data and build architectures with Nussl, I have attached my Jupyter notebook files showing my progress in the supporting files link at the bottom of this document. I was close to training my own model with this library until I came across a hurdle that I could not jump.

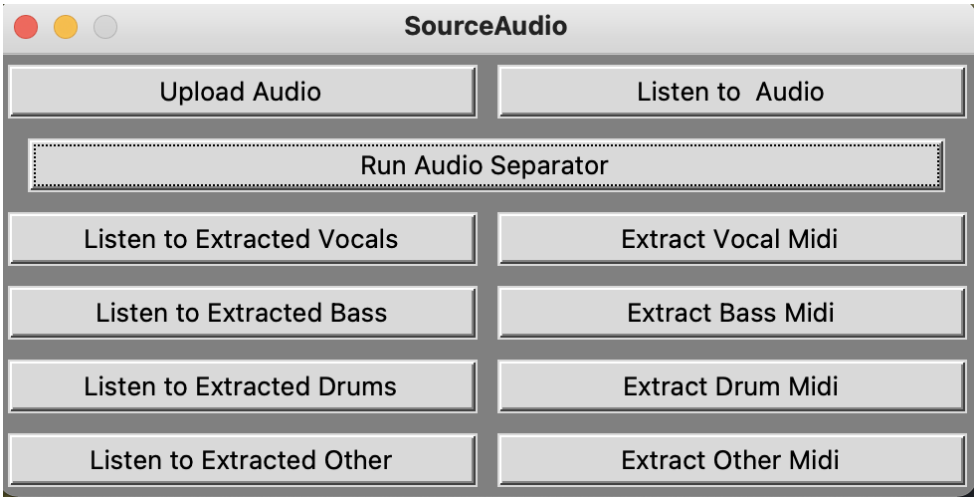
```
Output exceeds the size limit. Open the full output data in a text editor
---------------------------------------------------------------------------
OSError                                   Traceback (most recent call last)
Cell In[40], line 90
     88 nf = stft_params.window_length // 2 + 1
     89 nac = 1
---> 90 model = MaskInference.build(nf, nac, 50, 2, True, 0.3, 1, 'sigmoid')

Cell In[40], line 86, in MaskInference.build(cls, num_features, num_audio_channels, hidden_size, num_layers, bidirectional, dropout, num_sources, activation)
     79 config = {
     80     'name': cls.__name__,
     81     'modules': modules,
     82     'connections': connections,
     83     'output': output
     84 }
     85 # Step 3. Instantiate the model as a SeparationModel.
---> 86 return nussl.ml.SeparationModel(config)

File /opt/miniconda3/envs/tf/lib/python3.10/site-packages/nussl/ml/networks/separation_model.py:95, in SeparationModel.__init__(self, config, verbose)
     93 class_func = getattr(modules, module['class'])
     94 try:
---> 95     module_snapshot = inspect.getsource(class_func)
     96 except TypeError: # pragma: no cover
     97     module_snapshot = (
     98         "No module snapshot could be found. Did you define "
     99         "your class in an interactive Python environment? "
...
--> 785             raise OSError('source code not available')
    786     raise TypeError('{!r} is a built-in class'.format(object))
    787 if ismethod(object):

OSError: source code not available
```

Error from nussl training code

When I was putting all my code together for training, I came across the error above. The error was to do with the nussl.ml.SeparationModel that had to be integrated with the architecture for ease of deployment. I ran my code in google collaboratory and it did work, so it was just a local issue, however, I then encountered the dreaded Google collaboratory data limits. After many vigorous attempts at resolving the local issue and trying to work around the collab limits, I decided it was time to do some more research. In doing so I came across yet another library by the name of ‘Demucs’. This is another state-of-the-art music source separation model that had

been developed by the Facebook research team. Demucs uses a U-net convolutional architecture that performs a hybrid spectrogram/waveform separation using transformers. As I said I had originally hoped to train my own model, however, due to my lack of time left before the deadline I figured it would be wise to use a pre-trained model this time around. The model had been trained on the MUSDB HQ data set with an extra training dataset of 800 songs, as opposed to the nussl model that had been only trained on the MUSDB18 dataset. The MUSDB18 is a dataset of 150 full-length tracks from different genres, all separated into ‘vocals’, ‘drums’, ‘bass’ and ‘other’. The MUSDB HQ uses the same data but at a higher quality (.wav instead of .MP4) therefore, along with the fact it uses an extra 800 songs on top, it seemed actually wiser to use the Demucs pre-trained model, as my computer would not be able to handle that amount of data being processed, and It would theoretically produce better results.

The next stage in my process was to actually build the application to use this model. I decided to use the tkinter GUI library for the application as it is the standard python interface library. I also really like the style of interface you can get with the ttk widgets. As seen below.



Application made using tkinter

Once I had built the application and configured its interactivity I was able to implement the Demucs model. To do this I reconfigured the code from the Demucs Google collab tutorial.

```
def copy_process_streams(process: sp.Popen):
    def raw(stream: Optional[IO[bytes]]) -> IO[bytes]:
        assert stream is not None
        if isinstance(stream, io.BufferedIOBase):
            stream = stream.raw
        return stream

    p_stdout, p_stderr = raw(process.stdout), raw(process.stderr)
    stream_by_fd: Dict[int, Tuple[IO[bytes], io.StringIO, IO[str]]] = {
        p_stdout.fileno(): (p_stdout, sys.stdout),
        p_stderr.fileno(): (p_stderr, sys.stderr),
    }
    fds = list(stream_by_fd.keys())

    while fds:
        # `select` syscall will wait until one of the file descriptors has content.
        ready, _, _ = select.select(fds, [], [])
        for fd in ready:
            p_stream, std = stream_by_fd[fd]
            raw_buf = p_stream.read(2 ** 16)
            if not raw_buf:
                fds.remove(fd)
                continue
            buf = raw_buf.decode()
            std.write(buf)
            std.flush()
```

Copy process streams function from Demucs notebook

```
def separate(inp=None, outp=None):
    cmd = ["python3", "-m", "demucs.separate", "-o", str(outp), "-n", model]
    if mp3:
        cmd += ["--mp3", f"--mp3-bitrate={mp3_rate}"]
    if float32:
        cmd += ["--float32"]
    if int24:
        cmd += ["--int24"]
    if two_stems is not None:
        cmd += [f"--two-stems={two_stems}"]
    files = [inp, ]
    print(files)
    if not files:
        print("File not found!")
        return
    print("Going to separate the files:")
    print('\n'.join(files))
    print("With command: ", " ".join(cmd))
    p = sp.Popen(cmd + files, stdout=sp.PIPE, stderr=sp.PIPE)
    copy_process_streams(p)
    p.wait()
    if p.returncode != 0:
        print("Command failed, something went wrong.")
```

Separate function from Demucs notebook

Using the two functions above, all I had to do was call the separate function, passing it an input path and output path. The input is the file chosen by the user and the output will always be the downloads folder on the user's computer. One issue I had with this was that because the audio separation process is quite slow when calling the function synchronously it would freeze the application, so I had to call it in another thread. My issue was that once I had called the function and started the thread, I couldn't call it again without restarting my application. I eventually resolved this by creating a function that creates a new instance of a thread each time it is called and therefore you are not trying to restart a dead thread you are just creating and starting a new thread that does exactly the same thing.

For the audio-to-midi conversion, I used Spotify's basic-pitch model, a python library for Automatic music transcription that uses a lightweight neural network. The implementation of this was as simple as passing the model an input path and then writing the midi data to the same file path as the audio outputs.

The final outcome of my project works effectively and produces some very interesting results. I had originally looked at this project as a remixing tool, allowing users to get their favourite melodies from songs in a midi format so they could reprocess them any way they wanted, however, after completing the application I came to realise that this tool was a lot more experimental than I had originally thought. The audio output of the source separator isn't always perfect, depending on what song you upload, and sometimes you hear artefacts or bleed from other tracks that when converted to midi can create some very complex and interesting melodies. Below I have attached a link to a google drive with audio samples of my experimentation. Ideally, I would have trained my own model and/or tried multiple different architectures, however, due to my lack of resources I believe it was a lot more effective to use the pre-trained Demucs model.

Link to audio examples:
https://drive.google.com/drive/folders/1y2XO_L-dK5FBxjT_8Vh0GexrgwP08Vuj?usp=share_link

Link to supporting files:
https://drive.google.com/drive/folders/1ypyfG04JEIjq-YQZJ3k8VLatGr6nTqVA?usp=share_link

References

Demucs Model:
@inproceedings{defossez2021hybrid,
  title={Hybrid Spectrogram and Waveform Source Separation},
  author={D{\'e}fossez, Alexandre},
  booktitle={Proceedings of the ISMIR 2021 Workshop on Music Source Separation},
  year={2021}
}

Basic-pitch model:

@inproceedings{2022_BittnerBRME_LightweightNoteTranscription_ICASSP,
author= {Bittner, Rachel M. and Bosch, Juan Jos\'e and Rubinstein, David and Meseguer-Brocal, Gabriel and Ewert, Sebastian},

title= {A Lightweight Instrument-Agnostic Model for Polyphonic Note Transcription and Multipitch Estimation},

booktitle= {Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)},

  address= {Singapore},

  year= 2022,

}