Drum Beat Detection and Transcription

MSc Dissertation

Haralambos Dafas (C1738660)

Under the supervision of
Prof. David Marshall
Moderated by
Dr. Jing Wu

November, 2021

Acknowledgements

The author would like to thank Professor David Marshall for his invaluable guidance throughout the project, without which the work would not have been possible. Further thanks are given to Efstathia Economopoulou and Ayla Morris for their emotional support and advice.

Abstract

Drum patterns form a very important rhythmic component of music. This is no different for music composed digitally. While digital composers have access to many drum loops to use in their music, adapting drum patterns from other songs they have heard relegates them to either manually transcribing them, usually by ear (which is fallible), or directly sampling them (which greatly limit how much the pattern can be modified to suit a musician's personal interpretation.

This dissertation reviews the field of signal processing with regards to music and proposes a deep-learning-based, source-separation-focused program for *Automatic Drum Transcription (ADT)* wherein drums are separated from the rest of the instruments in a song, the different parts of the drum kit are separated from each other, and the onsets obtained from the isolated parts are used to transcribe the song's drum parts into MIDI format, which is the basis of most modern digital music composition. It is based on Spleeter, a Python source separation library. The program results in an F-measure of 0.65 given a commonly used onset tolerance window of 50ms. It was found that, in a dataset of kick drum, snare drum and hi-hat, the main limiting factor for this F-measure was successfully separating hi-hats from the others, as kick and snare were relatively easy to isolate. The importance of a sizeable and relevant dataset, as well as of accurate methods, is emphasized, due to limitations that may have hindered the program's performance.

Contents

Note: this dissertation uses clickable links. When references to other sections are made and underlined like this, they may be clicked to lead to the section to which they refer. The same applies to references to figures (e.g. fig. 2) - the number is clickable and leads to the figure.

1 Introduction

Notated music is an invaluable tool for music education and creation. For many novice and aspiring musicians, it can be sometimes be too difficult or time-consuming to manually notate music they have already listened to and wish to learn to play/adapt themselves. Digital music composition can have a low barrier of entry; there are many *Digital Audio Workstations (DAWs)* such as FL Studio[1], Ableton Live[2], GarageBand[3], and many more that allow users to compose music using samples, virtual instruments and even real instruments, if the user possesses one. Despite the low barrier of entry, composing an entire song can be a daunting task for beginners as usually there are numerous instruments which must be involved for a song to be created, some of which the user may not be familiar with. One such instance is in drums; while a lot of musicians, especially beginners, gravitate towards melody and harmony, most songs still require the use of drums or some other form of percussion in order to establish and sustain a rhythm for the song.

In order to aid digital composers in establishing a rhythm, there exist a variety of pre-made drum loops. However, these are usually composed manually, and may not always align with what the musician wishes to compose. Furthermore, there lies a possibility of a musician wishing to adapt an existing rhythm they may have heard in some other song. To this intent, they may sample parts of the song directly, but this does not allow them to modify the rhythm such that it may better suit the song in any way besides adjusting the tempo at which it plays. Musicians could attempt to manually copy a pattern, but this can be difficult to do without knowing the original's transcription, especially for someone unfamiliar with the instrument. While transcriptions may exist online for certain songs, not all songs will have a transcription for the musician to use. Furthermore, the transcriptions may be in a format unfamiliar to the musician; DAWs usually use a MIDI-based interface to notate their music, which is different from classical music notation.

Were such a process to be automated, this would open up a realm of possibilities for musicians. Composers would have easy access to patterns from other songs and would more easily be able to change or remix them instead of outright sampling them. This would especially be of benefit to those wishing to change or adapt music that is more niche or obscure, which is less likely to have been transcribed by others.

Drums tend to produce a lot of different sounds simultaneously as they are comprised of different parts. However, each part sounds roughly the same whenever it is played. Hence, one could theoretically separate all parts of a drum from each other, and use changes in volume to detect the presence of a drum part being played. Combining this for all parts of everything, one could theoretically be able to fully transcribe a drum pattern automatically.

This project aims to develop software that is capable of transcribing audio of a drum pattern into MIDI, directly to be used by a musician in their DAW of choice.

[1] `https://www.image-line.com/`
[2] `https://www.ableton.com/en/live/`
[3] `https://www.apple.com/uk/mac/garageband/`

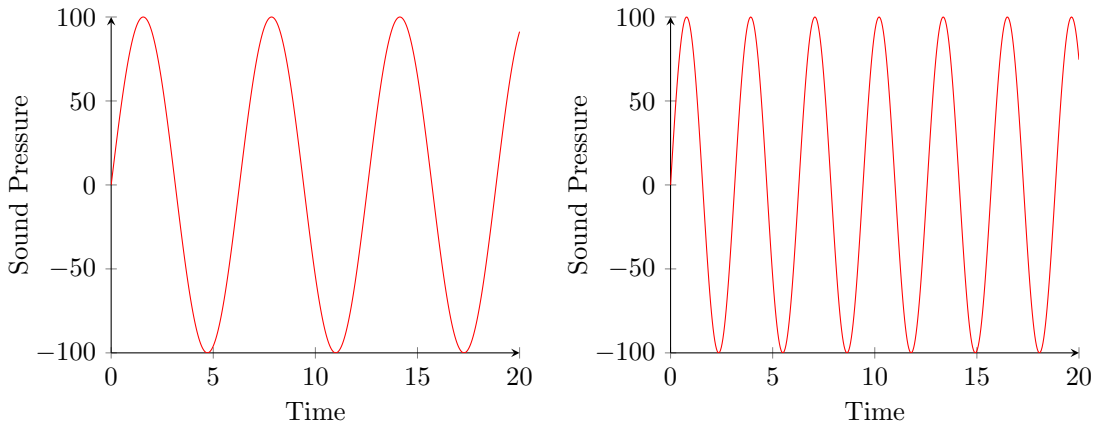
2 Background

2.1 Audio Signal Processing

The main area of research that deals with detecting audio is known as *Audio Signal Processing*. Sound travels through air and various other media via longitudinal waves which cause vibration, and has 4 main characteristics:

- Frequency is how often the waves occur per second, measured in Hertz (Hz).
- Amplitude or sound pressure is the difference between the average pressure of the medium the sound is travelling through and the pressure in the sound wave. This directly correlates to the loudness of the sound and is measured in decibels (dB), a logarithmic scale.
- Speed of the sound.
- Direction the sound is travelling in.

2.1.1 Frequency



(a) Low frequency wave.

(b) High frequency wave.

Figure 1: An example of sound pressure over time, at different frequencies, depicted by a sine wave.

For the purposes of processing audio signals for the analysis of music, amplitude and frequency are the most important aspects of a sound wave. Amplitude, as mentioned, correlates with how loud a sound is. Frequency corresponds to how high the pitch of the sound is. For example, children, who have higher pitched voices than adults, tend to speak at frequencies ranging from 250-400 Hz, while adults usually cover ranges from 150-200 Hz [1]. Both, of course, can generate sounds higher or lower - this is just the frequency of an average speaking voice. Humans can usually hear sound that is between 20 and 20 000 Hz [9].

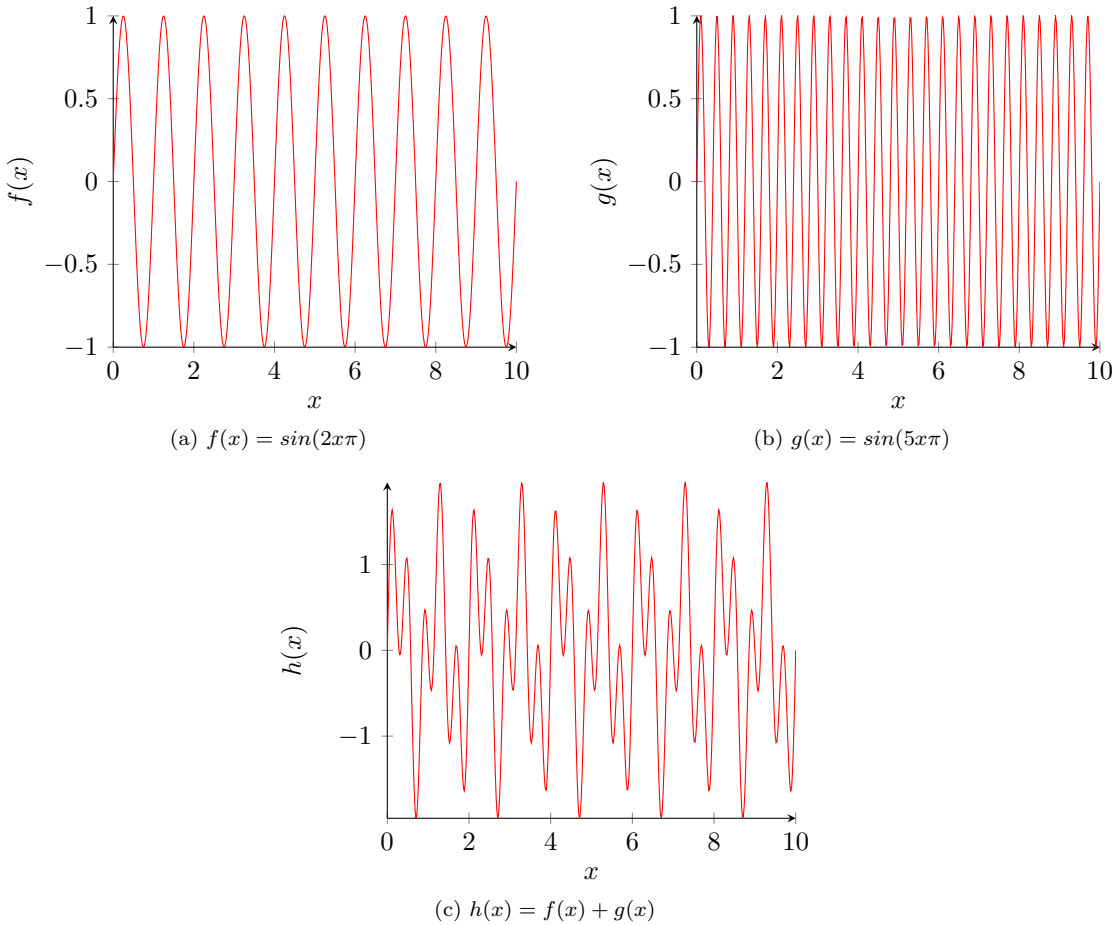
Different notes used in music have different pitches and hence frequencies as well. The *International Organisation for Standardisation (ISO)* have standardised that the note of A above middle C, or A_4, is to be set to 440 Hz[4]. Due to this, a note can be sometimes detected by noting its

[4] `https://www.iso.org/standard/3601.html`

frequency. For example, were one to measure a sound wave as 440 Hz, they'd be able to deduce that, assuming ISO standards apply, the note is an A_4.

Frequency can be visualised by plotting a change in sound pressure over time (see fig. 1). The distance between one peak and the next is called a wavelength, and this denotes one complete wave. The frequency is the number of complete waves that occur over time. This can be calculated very simply, $f = \frac{n_w}{t}$ where f is frequency, n_w is the number of waves, and t is time. In theory one could use this to deduce pitch. However, in praxis, this becomes a lot more complex. Sound is almost always composed of a fundamental pitch, which is usually the most prominent one that is heard, and a series of *overtones*, other pitches that are higher than the fundamental. Any pitch except for a true sine wave will contain overtones [7], and, since music is most usually not composed of just one single sine wave, detecting pitch becomes a lot more complicated due to the constant presence of these overtones.

2.1.2 The Fourier Transform



(a) $f(x) = sin(2x\pi)$

(b) $g(x) = sin(5x\pi)$

(c) $h(x) = f(x) + g(x)$

Figure 2: Adding two functions, $f(x)$ and $g(x)$, to produce a new one, $h(x)$. x can be used to represent time, or any other domain.

A single sine wave can be represented as a function, such as $f(x) = sin(x)$. Therefore, a sound

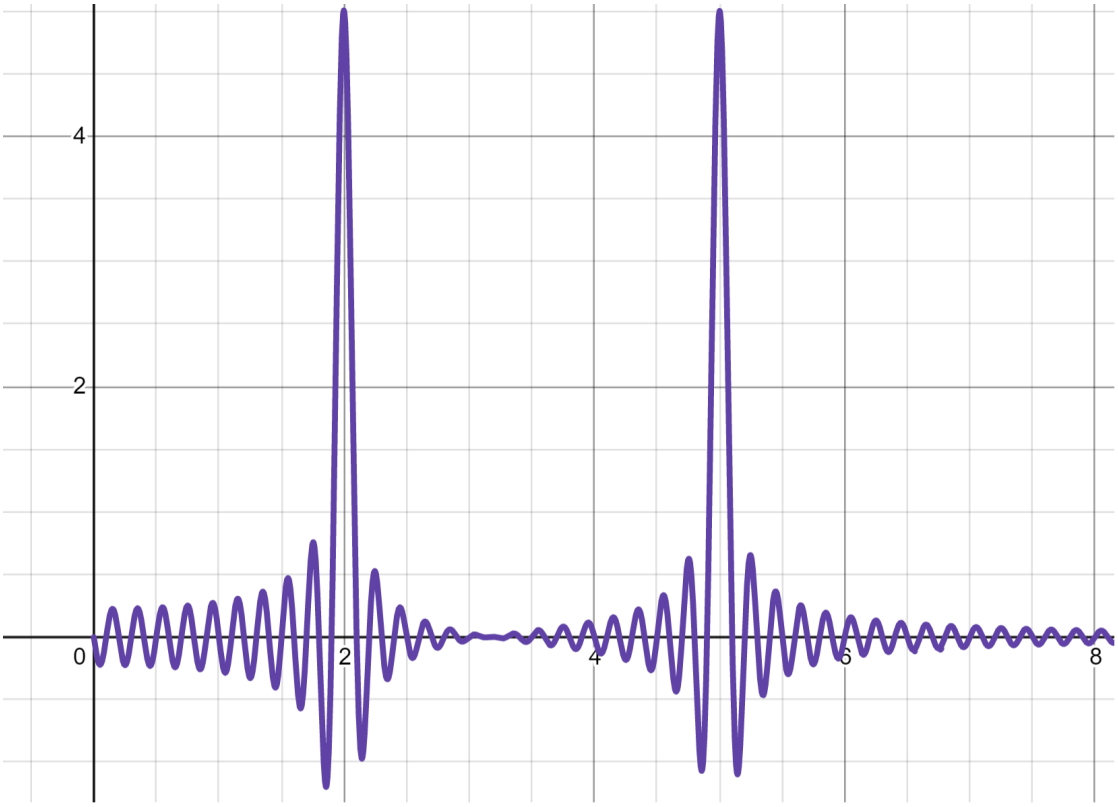


Figure 3: Fourier Transform perform on $h(x)$ from fig. 2. Peaks can be seen at 2 and 5, the frequencies of constituent functions$f(x)$ and $g(x)$, respectively. This was obtained via $\int_0^{10} h(x)sin(2\pi x\xi)dx$. Graph is different from the previous due to the need of using separate software for calculating integrals.

that includes overtones can be said to be composed of a composition of various different functions (see fig. 2 for a visualisation of adding two sine waves together), representing the fundamental pitch and the individual overtones. Thus, finding the fundamental pitch and overtones becomes a problem of decomposing the composite function of multiple pitches into the functions of its constituent pitches. This can be done by applying a *Fourier Transform (FT)*.

The FT of a function $f(x)$ will result in a new function $\hat{f}(x)$, the peaks of which correspond to the frequency of its constituent functions. In effect, the function represents the quantity of each frequency present in the original function. For example, in fig. 2, the two functions composing $f(x)$ and $g(x)$ have frequencies of 2 and 5 respectively. Plotting the FT $\hat{h}(x)$ will result in the graph shown in fig. 3, with clear peaks at 2 and 5.

There are two ways one can compute a FT, one using the complex plane:

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x)e^{-2\pi ix\xi}dx \tag{1}$$

The other using either sines or cosines:

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) sin(2\pi x\xi) dx \tag{2}$$

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) cos(2\pi x\xi) dx \tag{3}$$

A variant of the FT is the *Short-Time Fourier Transformation (STFT)*, which essentially splits up a function into chunks and computes the FT on those chunks individually. This variant is one usually used in order to produce a spectrogram of certain particular sounds. This is expressed as follows:

$$X(\tau, \omega) = \int_{-\infty}^{\infty} x(t) w(t - \tau) e^{-i\omega t} dt \tag{4}$$

Where $w(\tau)$ is a window function, used to separate the whole equation into chunks.

2.1.3 Spectrograms

Using the Fourier Transform, we are now capable of finding the frequencies within a given sound at a particular time. By computing how much of each frequency is present per interval of time, a *spectrogram* can be made to display this visually (see fig. 4 and fig. 5 for examples). Spectrograms are laid out as follows:

- The x axis represents time.
- The y axis represents frequency in Hz. Since the frequency of notes in music goes increases exponentially (increasing a note's pitch by an octave doubles its frequency), the y axis is often logarithmic.
- Areas in the spectrogram are shaded in whenever a specific frequency occurs at a specific time. Different shades represent a higher or lower presence of a particular frequency (i.e. how loud that particular frequency is). This is measured in dB.

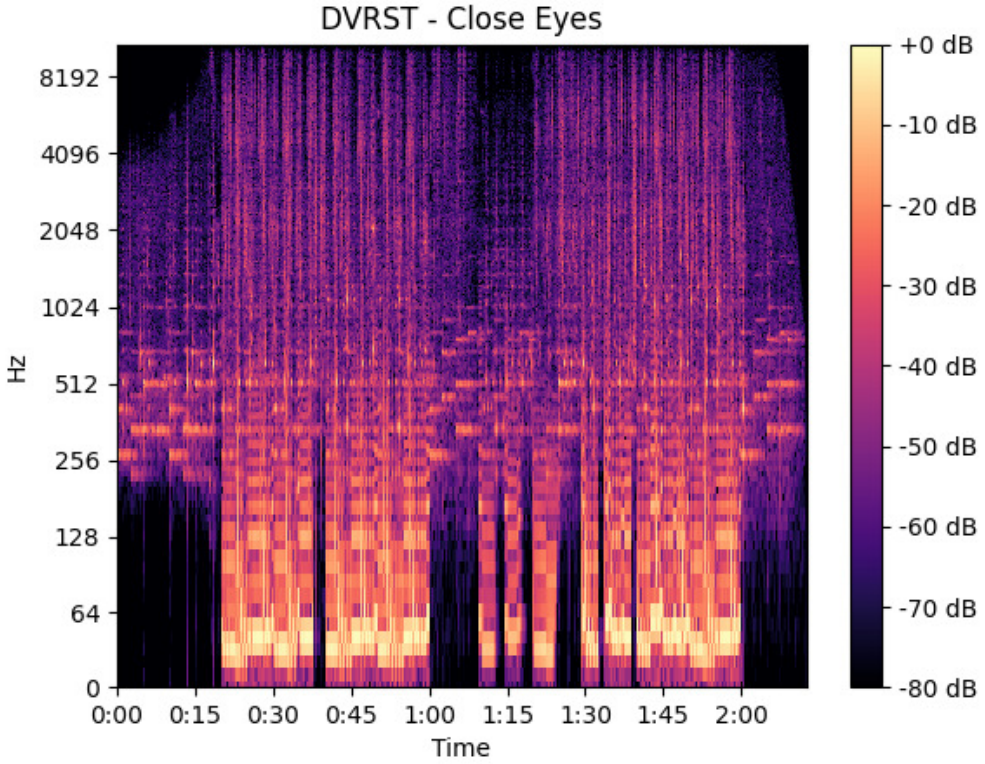


Figure 4: Spectrogram of the song Close Eyes by DVRST, a simple, repetitive 2 minute song. It is possible to make some educated guesses on the song's structure purely from the visualisation alone. For example, there are few frequencies present initially in the song's introduction. More frequencies appear at around 0:20, disappear at around 0:38, then reappear at around 0:42. Both sets seem similar, so this may perhaps hint at the presence of one verse which is repeated.

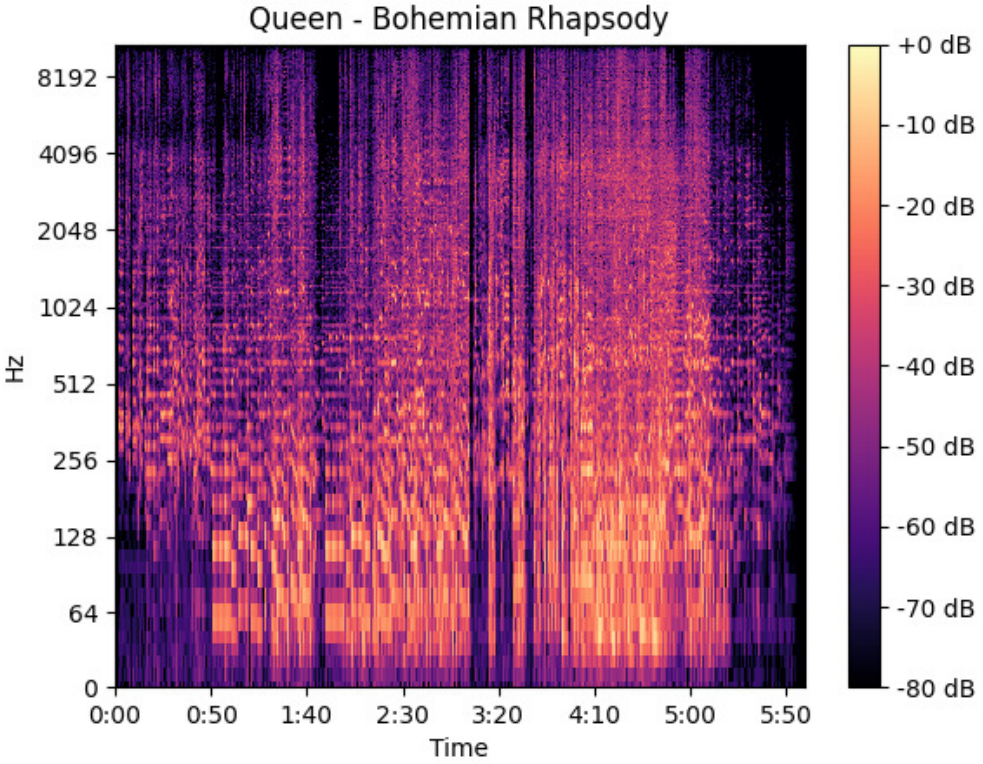
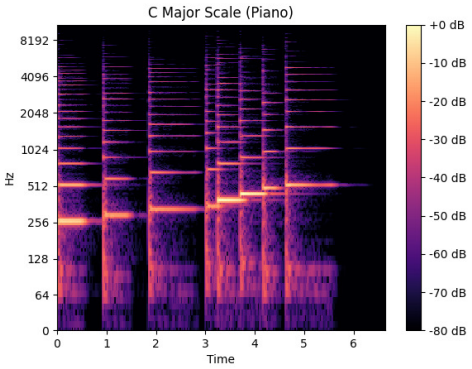
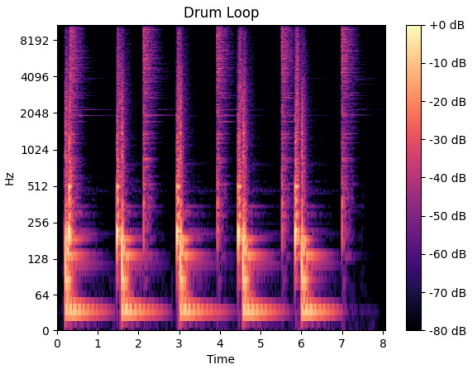


Figure 5: Spectrogram of Bohemian Rhapsody by Queen, a song well-known for its various changes in instrumentation as it progresses. We can see a relatively simple introduction up until what is likely a verse at 0:50, where more instruments come in. The part beginning at 3:10 is sung a cappella, with the absence of accompanying instrumentation being visible on the spectrogram as far fewer frequencies being present. Immediately after this, at around 4:00, we can see a crescendo where a lot of instrumentation overlaps, since this area of the spectrogram is quite heavily populated at all frequencies.

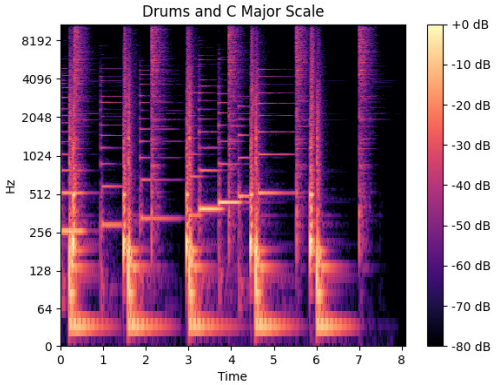
2.1.4 Source Separation



(a) Spectrogram of an ascending C Major scale played on a piano, with notes played at an irregular rhythm. Note how the lines above the fundamental pitch appear at regular intervals - these are its harmonics, overtones which appear at integer multiples of the fundamental pitch.



(b) Spectrogram of a drum pattern.



(c) Spectrogram of the above two sounds played on top of each other.

Figure 6: Two spectrograms composed into a third.

Music is usually composed of various instruments, overlapping with each other. Much like how a particular sound's fundamental pitch and overtones can add together to create a more complicated function, which is then decomposed to find its constituents, so can spectrograms of individual instruments be added together to create a more complicated spectrogram (see fig. 6), the decomposition of which has its own dedicated field of research known as *source separation*, itself a subfield of *signal separation*. There are other applications for source separation besides just separating the different instruments used in a song. For example, in the cocktail party problem, there are multiple speakers having different conversations simultaneously at a cocktail party, and source separation would aim to isolate the conversations from each other.

Due to the fact that spectrograms are represented as an image, approaches used for decomposing images into constituents can be applied to audio source separation, leading to an overlap between

the fields. Since source separation is the approach this dissertation is going to take in order to achieve its aim for drums transcription, source separation will be discussed in more detail in the discussion of this dissertation's approach.

2.1.5 Music Information Retrieval and Automatic Drum Transcription

There is a dedicated field of research in MIR in which *Automatic Music Transcription (AMT)* is considered one of the most important and complex problems [2]. Most of it focuses on detecting things such as pitch, onset time (the time at which notes are played), and duration of notes played by specific melodic instruments (such as guitar or piano) or performed by a human voice [22]. A subsection of it, *Automatic Drum Transcription (ADT)*, focuses specifically on transcribing drums.

Since most instruments consist of a pitch, detecting and classifying non-pitched sounds is considered its own problem [13]. Note the difference in spectrograms between a piano and a drum kit, shown in fig. 6. This usually takes the form of *Automatic Drum Transcription (ADT)* since drums are the most commonly used non-pitched instrument found in Western music. A review of the current literature in ADT was done by Wu et al. [28], who suggested that source separation methods could be used as a preprocessing technique to enhance the accuracy of current ADT models.

Early works in *Automatic Drum Transcription (ADT)* [8] [16] proposed the following four types of approaches:

- *Segment and Classify*
- *Separate and Detect*
- *Match and Adapt*
- *Hidden-Markov-Model-based Recognition*

Of these, a source separation approach can be considered as *separate and detect*, since the two main steps of the approach are literally to separate sources and then detect their onsets (more on onsets is discussed later). More recently, the literature overview done by Wu et al. [28] used the different steps used in various other attempted approaches to propose a new set of four classifications:

- *Segmentation-Based Methods*
- *Classification-Based Methods*
- *Language-Model-Based Methods*
- *Activation-Based Methods*

2.2 The Drum Kit

Drums are a percussive instrument, sometimes also called membranophones [24]. A single drum is composed of a membrane stretched over a shell that is then struck either by the player's hands or with various dedicated objects such as drumsticks or brushes. There are various kinds of drums, and these are used in varying numbers depending on what music is being played. For example, in a marching band, drummers typically have one drum which they wear with a harness to allow for them to move while playing, and they typically use drumsticks. In contrast, bongo drums usually come in pairs and the drummer plays them directly with their hands. Timpani

come in sets, can be tuned to different pitches and are struck with a mallet. Finally, the main subject of this dissertation is the modern day drum kit, which is very commonly used in Western music and is usually what is referred to whenever drums are mentioned in everyday parlance. This will be examined in further detail (see creffig:drumsdiagram for a diagram).

(a) A marching band drum, with harness and drumsticks.

(b) A pair of bongo drums.

(c) A timpani set.

(d) A modern day drum kit.

Figure 7: Various drums.

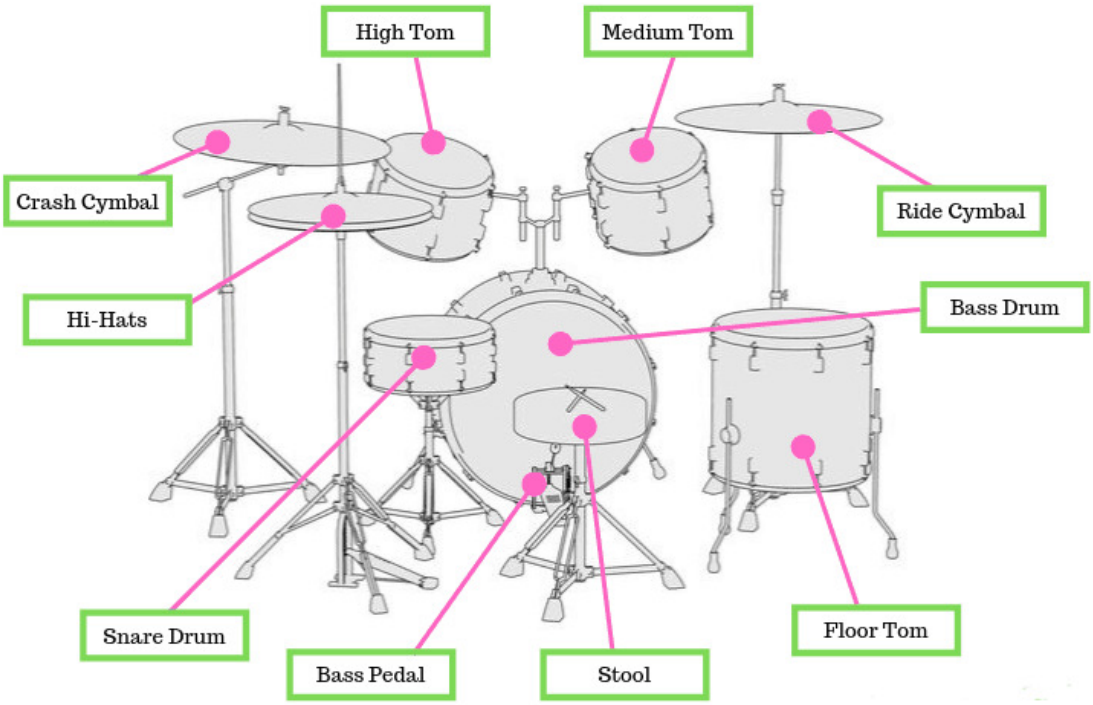


Figure 8: Diagram of the modern drum kit.

It should be noted before the parts of a drum kit are discussed that drum kits can greatly vary depending on the drummer, as there is a large variety of different instruments that can be added or removed in order to add greater sound variety or simplify playing and moving the kit. With that said, most modern drum kits are composed of a mixture of drums and cymbals (round metal plates), which are usually struck by some variety of drumsticks. The most important parts are listed as follows:

- *Snare Drum (SD).*
- *Bass Drum (KD).* Also called a Kick Drum (hence KD). These are struck by a mallet attached to a pedal which is activated by one of the drummer's feet.
- *Hi-Hats (HH).* A pair of cymbals stacked on top of each other. The top cymbal can be moved up (and off of the bottom cymbal) via a pedal accessed with one of the drummer's feet. Usually a drummer will have one foot on the HH pedal and one on the KD pedal. Depending on whether the two cymbals touch, the Hi-Hats can be considered open (not touching) or closed (touching), each producing different types of sounds. One can also strike an open Hi-Hat then immediately close it, or use the pedal to lift and drop the top Hi-Hat onto the bottom one.

HH, KD and SD form the foundation of the modern drum kit, and are always included in every kit. Certain minimalist kits contain only these three. The most basic pattern these three are played with to denote a rhythm (also called a groove) involves striking the HH at regular intervals (or quarter notes), while alternating between KD and SD. This is colloquially known as the basic

rock groove, due to its prevalence in rock music. The other parts of the instrument, which are used to support the main three of HH, KD and SD, are as follows:

- *Tom-Tom Drums (HT, MT, FT).* Usually in sets of three, High, Medium and Floor Toms, though occasionally more of them are added to a kit. These have a different sound than the SD, and are most often used in drum fills which mark the end or beginning of new sections of a song.
- Crash Cymbal (CC). These have a strong, sustained sound that stands out from the rest of the kit. Usually used to mark new sections of a song, or change the feel to a louder, more energetic one.
- Ride Cymbal (RC). These are usually used for a purpose similar to the HH, being struck at regular intervals to create a steady groove. Usually replace the HH to change up the sound of the drums and create a different "feel".

There are two major categories of modern drum kits - traditional (or acoustic) and electronic. Acoustic drums can allow a drummer to create more specific and unique sounds than electronic drums (since all sounds an electronic set makes must be programmed into it), while electronic drums can be directly connected into computer software which can write down which parts are played and when. Electronic drums can also be programmed to produce different sounds when played. This makes electronic kits, compared to acoustic ones, paradoxically more flexible (due to the large variety of samples they can play from) and more restrictive (due to not being able to exactly replicate every single intricacy of the physics of an acoustic kit).

2.3 MIDI

Electronic drums are considered a type of *MIDI* instrument, similar to electronic keyboards, due to their digital nature. MIDI stands for *Musical Instrument Digital Interface*, and includes most of the aspects that are involved in any music that is produced digitally. One MIDI link through a MIDI cable can carry up to 16 different channels of information (channel 10 is usually reserved for percussion - this will play into the transcription part of this dissertation). MIDI is based around the transmission of event messages, such as a note's pitch, duration or velocity (usually correlated with loudness), or tempo (how fast the rhythm plays).

Whenever a MIDI instrument is played, all of the interactions a musician has with the instrument (such as pressing keys on a keyboard or striking a specific part of an electronic drum kit) are converted into MIDI. These can then be passed on to a sound module, which generates sounds based on what was played (e.g. playing a crash cymbal sound when an electronic crash cymbal is struck), or transferred into a computer, which can save the MIDI data as a file (in effect automatically transcribing the music). Due to the ability to save MIDI data, electronic drummers can already access and alter music they themselves have played, which is another advantage electronic drums have over acoustic kits.

While notes for MIDI keyboards simply correspond to playing the same sound at a higher pitch, drums have to map notes to the different instruments in a kit. Notes moving through channel 10, the percussion channel, are mapped in fig. 9.

2.3.1 Digital Audio Workstations

As mentioned earlier in this section, a MIDI instrument's output can either be connected to a sound generator or to a computer. If connected to a computer and saved as a MIDI file (using

Key	Note	Sound
35	B0	Acoustic Bass Drum
36	C1	Bass Drum 1
37	C#1	Side Stick
38	D1	Acoustic Snare
39	Eb1	Hand Clap
40	E1	Electric Snare
41	F1	Low Floor Tom
42	F#1	Closed Hi-Hat
43	G1	High Floor Tom
44	Ab1	Pedal Hi-Hat
45	A1	Low Tom
46	Bb1	Open Hi-Hat
47	B1	Low-Mid Tom
48	C2	Hi-Mid Tom
49	C#2	Crash Cymbal 1
50	D2	High Tom
51	Eb2	Ride Cymbal 1
52	E2	Chinese Cymbal
53	F2	Ride Bell
54	F#2	Tambourine
55	G2	Splash Cymbal
56	Ab2	Cowbell
57	A2	Crash Cymbal 2
58	Bb2	Vibraslap

Key	Note	Sound
59	B2	Ride Cymbal 2
60	C3	High Bongo
61	C#3	Low Bongo
62	D3	Mute High Conga
63	Eb3	Open High Conga
64	E3	Low Conga
65	F3	High Timbale
66	F#3	Low Timbale
67	G3	High Agogo
68	Ab3	Low Agogo
69	A3	Cabasa
70	Bb3	Maracas
71	B3	Short Whistle
72	C4	Long Whistle
73	C#4	Short Guiro
74	D4	Long Guiro
75	Eb4	Claves
76	E4	High Wood Block
77	F4	Low Wood Block
78	F#4	Mute Cuica
79	G4	Open Cuica
80	Ab4	Mute Triangle
81	A4	Open Triangle

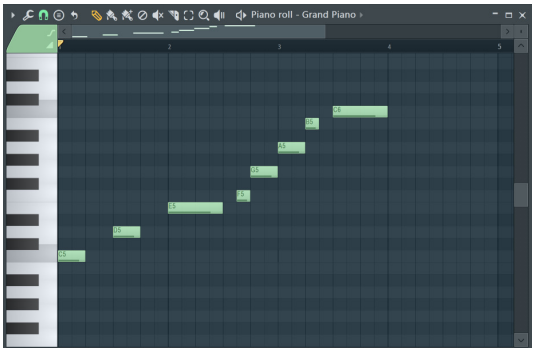
Figure 9: Percussion mappings for MIDI notes.

file format .mid), these can then be placed into a *Digital Audio Workstation (DAW)*, in which different MIDI files can be combined, alongside samples and recordings of acoustic instruments to create music. The introduction to this dissertation listed a few common DAWs.

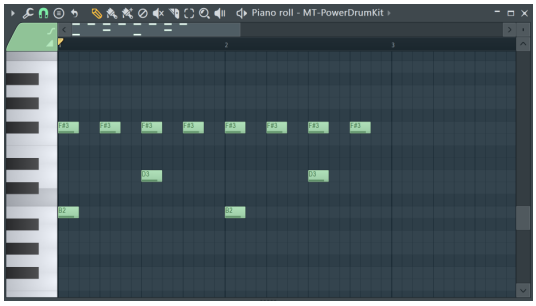
Within DAWs, MIDI notes are most often visualised via notes along a grid, placed next to a virtual piano. It is usually here where either MIDI instruments are played or MIDI files are imported, allowing the DAW to allocate sounds to the notes that have been played. As an example, fig. 10 shows FL Studio's MIDI notation of the piano scale shown in fig. 6 and a basic rock groove.

2.4 Automatic Transcription: Pitched vs Non-Pitched

Unlike pitched instruments, which exhibit fairly structured spectra focusing on a base pitch and a structured harmonic series, the sounds produced by drum instruments are more noise-like. Due to this, some existing AMT algorithms used for transcribing pitched instruments that are based on frequency cannot be applied in the same way to ADT. The lack of pitch is further compounded by the fact that very often multiple parts of a drum kit are struck simultaneously, creating an overlap in their sound and makind certain other parts of the instrument harder to detect. This makes approaches that work for single melodic cadences such as a singing voice even less suitable. Recall the difference between spectrograms of a piano and drums shown in fig. 6.



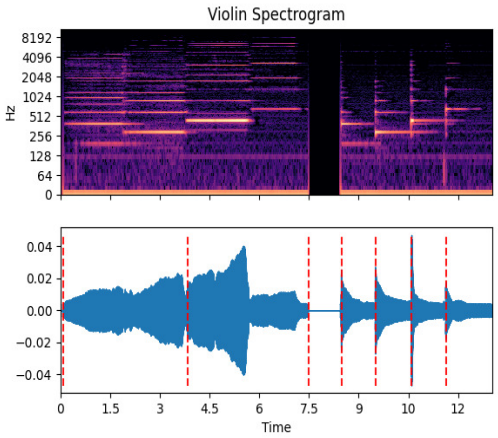
(a) MIDI notation of a C major scale, with notes played at an irregular rhythm.



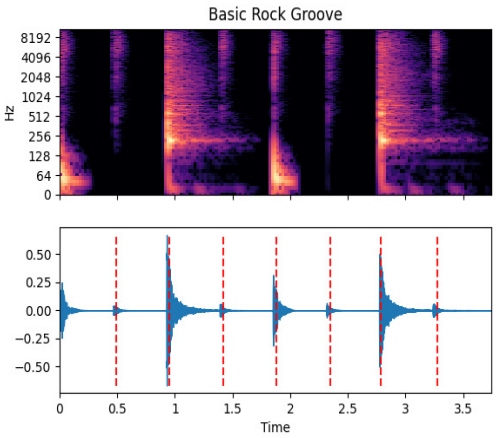
(b) MIDI notation of a basic rock groove.

Figure 10: C major scale and basic rock groove, as seen in MIDI format in FL Studio.

While frequency may not work for drums very reliably, *onset detection* does. The onset of a note is simply the time at which it begins. Being percussive instruments, drums largely exhibit transient sounds - they begin loudly (i.e. with a high amplitude) and abruptly, and then decay a lot more quietly. This is also described as drum notes having a high *attack*. Thus, by detecting large and abrupt changes in amplitude, one could find the onset of a drum note. A comparison between high and low attack is shown in fig. 11, showing the spectrogram and strength of sound produced by a violin played initially with a bow (which has a very low attack) and then plucked (which has a much higher attack). This is also compared to the basic MIDI rock groove shown in fig. 10. A simple onset detection algorithm was done on both, showing that it is more effective at finding notes when they exhibit a high attack.



(a) Ascending violin scale. First half is played with a bow, second is plucked. Note how the onset detection algorithm fails to find the notes at 2 and 6 seconds.



(b) The same rock groove shown in fig. 10.

Figure 11: Comparison between onsets of a violin, bowed and plucked, and drums.

It can be seen in fig. 11 that drums have very prominent and easily detectable onsets. In fact,

the only issue with using onsets for transcribing drums from audio is that different parts of the drum are not distinguished from each other. For example, the notes played by the HH at 0.5s, $\tilde{1}$.5s, etc. and the notes played by the SD at 1s and $\tilde{2}$.75s are both detected as onsets. The distinction between notes becomes further compounded by the fact that even in the most basic of grooves a drum kit is often playing multiple notes at the same time. Whenever KD and SD are struck in the rock groove, the HH is also struck simultaneously. This is detected as only a single onset, when in reality it is two different notes.

However, if an audio file were to contain only a part of the instrument in isolation, (e.g. only the HH hits in the rock groove), then transcribing it would be relatively trivial due to the transient nature of drums, since all one would need to do is detect the onsets. Hence, it can be hypothesised that, if all parts of a drum kit could be perfectly isolated from each other, ADT can be reduced to simple onset detection per isolated instrument. Then, once all onsets are found for all instruments, they can be combined to result in a complete transcription.

This hypothetical approach is what this dissertation attempts to implement in practice.

3 Approach

This dissertation will attempt to create a program which, given an audio input of drums being played (with no other instruments involved save for those included in a drum kit), will provide a MIDI transcription of the drums as output. This transcription could then be inserted into a DAW and immediately provide the user with a drum beat they can use in their music. As mentioned in the previous section, the approach that is going to be taken is one of source separation. The general idea is that the program would work as follows:

1. Drum audio file is taken as input.
2. Drums are decomposed into their constituent instruments (e.g. HH, SD, KD), and saved as individual audio files of their own.
3. Onsets of the separated drum parts are detected, mapping each part to the times at which it is played.
4. The above onset mapping is used to create a MIDI transcription of the drum beat. This is saved as a MIDI (.mid) file, ready for use.

Source separation is its own dedicated field of study. As such there are numerous approaches that have been used to attempt it, such as *Non-negative Matrix Factorisation (NMF)* [23] [25] [26], Bayesian Models [15], and analysing repeating patterns [19]. However, this dissertation will instead attempt to use *deep learning* in order to train a model which could, ideally, predict which parts of a drum's spectrogram belong to which parts of the drum itself and use this to separate the audio file into its constituent drum parts. Fortunately, instead of having to implement a neural network architecture from scratch, there already exists an Python library which can be adapted to perform the requisite source separation.

3.1 Spleeter

Spleeter[5] is a tool for source separation initially presented by Hennequin et al. [10] which implements source separation for separating the different instruments in a song (specifically vocals, drums, bass, guitar, piano and "other"). It includes pre-trained models for this kind of song decomposition which compete with state-of-the-art when used to decompose songs in the musdb18 dataset [20], a common benchmark dataset for source separation as applied to different instruments in songs. Of course, this is a wholly different application than the one in this dissertation. However, it can also be adapted to perform separation of custom sources given a dataset on which it can train and a variety of configuration settings. Hence, if provided with an appropriate dataset of drum sounds, it can be trained to separate the different parts of a drum kit from each other.

An added benefit of using Spleeter is that the pre-trained models can actually be practically made use of in order to expand the scope of the program. Since Spleeter has competes with state-of-the-art in separating parts of a music track, this may be used to isolate the drums of any song. This would allow the program to not only transcribe isolated drums, but the drum patterns of any song that includes other instruments playing alongside the drums as well. Of course, since even state-of-the-art source separation is not perfect, obtaining a perfect transcription would likely be very optimistic. However, it may still be possible to obtain an approximate transcription, which still greatly expands the potential use cases of the program. Due to this, the general idea of the program outlined in the beginning of this section can be updated thus:

[5]Source code can be found at `https://github.com/deezer/spleeter`

1. **Full song is taken as input.**
2. **Drums are separated from the rest of the song using a pretrained model, then passed on to the rest of the program.**
3. Drum audio file is taken as input.
4. Drums are decomposed into their constituent instruments (e.g. HH, SD, KD), and saved as individual audio files of their own.
5. Onsets of the separated drum parts are detected, mapping each part to the times at which it is played.
6. The above onset mapping is used to create a MIDI transcription of the drum beat. This is saved as a MIDI (.mid) file, ready for use.

For actually performing source separation, Spleeter uses a *U-Net* architecture.

3.1.1 The U-Net

The U-Net is an architecture initially used for medical imaging [21], a field which requires fine accuracy in image recreation. Coincidentally, source separation also requires fine accuracy in image recreation; a stray pixel within a spectrogram can have a very large difference in the sound it describes. Since frequency is on a logarithmic scale, a pixel being slightly higher may completely change an instrument's pitch, add enough noise to make the fundamental incomprehensible, or have a variety of similar undesirable effects. Due to the U-Net's capability of recreating fine detail, it has already been used in source separation before, such as for separating a song's vocals from the instrumentation [11].

At its core, the U-Net is a type of *Convolutional Neural Network (CNN)*. These have historically used for classifying images (e.g. an input image is classed as either a cat or a dog), and utilise *convolutions* in order to extract features from an image. An example of a convolution, with a short explanation of how they work, can be seen in fig. 12.

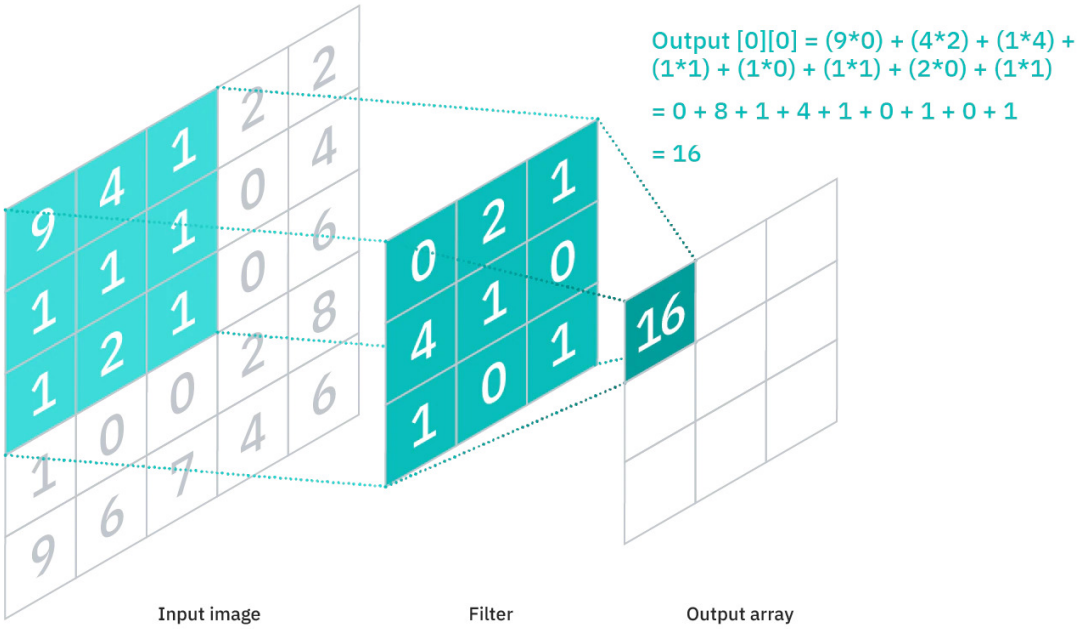


Figure 12: The larger matrix in this figure can be considered a representation of an image, where a number represents how much a colour (e.g. green) is present in a given pixel and the number's position represents the pixel's position in the image. This is then fed through a convolution where a number is selected and it and its surrounding numbers are fed through a filter (which is used to find features in the image) in order to produce a new number. This is sometimes combined with *pooling*, which reduces the size of the image after it is convolved, though sometimes an image can be convolved multiple times before being pooled. An activation function is applied to this (for example, ReLU) and the result is a *feature map*, seen as the smaller matrix on the right, representing how much of a feature is present. After (usually) many convolutions, the resulting feature map can be used to classify an image using a variety of classification techniques (such as, in the example of classifying cats and dogs, providing probabilities that a given image is a cat or a dog).

Traditional CNNs generate relatively few predictions, used to classify the image. U-Nets, on the other hand, provide an architecture which is used to classify each pixel within an image, thus recreating the image, which is done by combining the usual CNN's convolution layers with an equivalent number of *deconvolutional layers*, where the pooling is replaced by upsampling. Furthermore, while the image size is halved, the number of feature channels (each channel containing a feature map) is doubled every convolution, resulting in a small but deep representation of the original image. Finally, in order to make the upsampling more accurate, features from the convolutional layers are combined with the deconvolutional layers, which allows the resulting model to retain a lot of the lower-level accuracy that could otherwise be lost by the convolutions. A visual representation of the U-Net is seen in fig. 13.

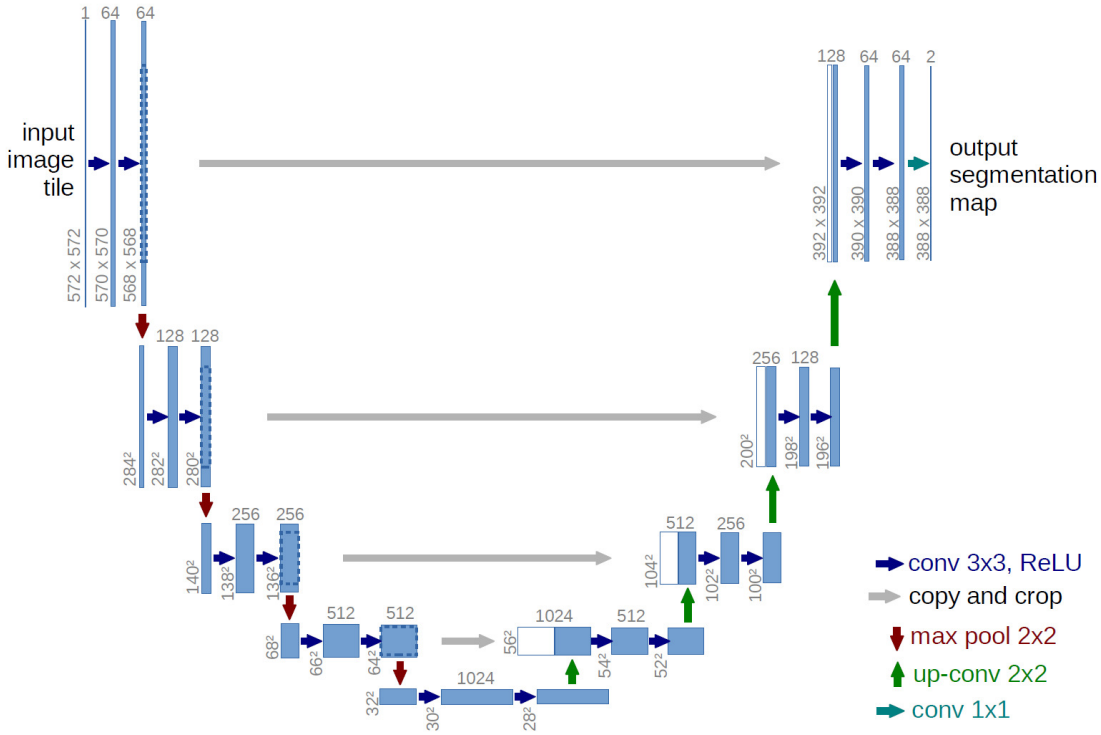


Figure 13: Diagram representation of an example U-Net, with operations on each layer labelled on the bottom right. It begins with an input image of 572x572, which is convolved with a ReLU activation function twice, obtaining 64 feature maps from it (shown above the layer). This is then pooled via 2x2 max pooling (i.e. the image is split into 2x2 matrices and each is simplified down to a single number by selecting the largest), therefore halving the size of the image. The feature channels are doubled every time pooling occurs up until the image is a mere 28x28 with 1024 feature channels. This is then repeatedly deconvolved back up until the image is approximately the same size as the original. Each deconvolution layer is combined with its respective convolution layer, denoted by the gray arrows, so that any features lost during convolution are recovered. Finally, the result is passed through a Wiener filter, which estimates the output image.

For the purposes of source separation, Spleeter uses one U-Net per source. For example, its pre-trained 4-stem model, which separates a song into vocals, drums, bass and other, maps a U-Net to each instrument. Each U-Net is then dedicated to predicting the spectrogram of its particular instrument in isolation. For example, the U-Net for vocals takes the initial spectrogram as an input and its output is a prediction of what the vocals in the spectrogram are, with everything not considered vocals omitted.

3.2 Datasets

As with any deep learning approach, it is important to obtain a large and relevant dataset. Since the objective of this program is to take an input of an isolated drum track and produce an output of each drum part within the track in isolation, the dataset must consist of drum patterns in audio form, alongside each part of the drum playing that pattern in complete isolation from the others. Fortunately, there is a dataset that is very commonly used in ADT literature [28] - the

IDMT-SMT dataset.

The dataset contains acoustic, sampled and synthesized drum loops composed purely of *Hi-Hat*, *Snare Drum (SD)* and *Kick Drum (KD)*, which were described as the three fundamental parts of any drum kit in section 2.2. It also contains the onsets for each track. The isolated tracks were combined to produce mixes using all three parts of the kit, resulting in 64 mixes in total. It should be noted that the dataset actually had the potential for a total of 192 mixes by shuffling the different parts around, but this had not been considered during this pre-processing (one of the many limitations of the approach that was taken).This dataset was then split up into training/validation/testing datasets with a ratio of 80/10/10 % (i.e. 51/6/7), which leads to another limitation, which is that cross-validation could have been used but were also not considered (though implementing such an approach within Spleeter may have required to alter Spleeter's own code).

Additional pre-processing that was performed was changing files from mono (1 channel) to stereo (2 channels, usually used for left/right speakers/headphones), due to an initial assumption that Spleeter failed if given a mono input. This was done by duplicating the mono channel. A reversal function, setting the channels back to mono, was also created, since it was then discovered that Spleeter actually worked better with mono channels despite mainly being programmed for stereo. This was one of a fair few challenges that arose from implementing Spleeter, the specifics of which will be discussed later. The mono/stereo functions are shown in listing 1.

```
import wave
import array
from pydub import AudioSegment


def make_stereo(file1):
    """ Converts a file from mono to stereo."""

    ifile = wave.open(file1)
    # print(ifile.getparams())
    (nchannels, sampwidth, framerate, nframes, comptype, compname) = ifile.
    getparams()
    assert comptype == 'NONE'
    array_type = {1:'B', 2: 'h', 4: 'l'}[sampwidth]
    left_channel = array.array(array_type, ifile.readframes(nframes))[::nchannels]
    ifile.close()

    stereo = 2 * left_channel
    stereo[0::2] = stereo[1::2] = left_channel

    ofile = wave.open(file1, 'w')
    ofile.setparams((2, sampwidth, framerate, nframes, comptype, compname))
    ofile.writeframes(stereo.tostring())
    ofile.close()


def make_mono(ifile):
    """ Converts a file from stereo to mono. """

    sound = AudioSegment.from_wav(ifile)
    sound = sound.set_channels(1)
    sound.export(ifile, format="wav")
```

Listing 1: Functions for changing files from mono to stereo and vice versa.

3.2.1 Setting up Spleeter

In order to train a model, Spleeter requires a configuration to be specified in the form of a .json file. The specific configuration file used for training the drum separation model is as follows, with comments edited in and marked with a #:

```
{
    "train_csv": "csv_files/drum_train_new.csv", # Path to a csv file containing
    paths for training data
    "validation_csv": "csv_files/drum_validation_new.csv",  # Path to a csv file
    containing paths for validation data
    "model_dir": "drum_model_new",  # Path to store the model
    "mix_name": "mix",  # Name of mix files (containing all sources)
    "instrument_list": ["HH", "SD", "KD"],  # List of different sources to
    separate

    # Audio parameters
    "sample_rate":44100,  # Sample rate for loading audio files
    "frame_length":4096,  # Frame length of short time Fourier transform (STFT)
    "frame_step":1024,  # Frame step of STFT
    "T":512,  # Time length of input spectrogram segment in STFT frames
    "F":1024,  # Number of frequency bins to be processed
    "n_channels":2,  # Number of channels of input audio (1 for mono, 2 for stereo
    )
    # Oddly enough training only completed when this was set to 2 and the training
     data was in mono

    "separation_exponent":2,  # Used for computing ratio masks. 2 is equal to a
    Wiener filter
    "mask_extension":"zeros",  # Sets anything above frequency bin F to either
    zero or the average below F.
    "learning_rate": 1e-4,
    "batch_size":4,  # Size of mini-batches used in optimisation
    "training_cache":"cache/training",
    "validation_cache":"cache/validation",
    "train_max_steps": 100000,
    "throttle_secs":1800,
    "random_seed":2,  # Used for randomness
    "save_checkpoints_steps":1000,
    "save_summary_steps":5,
    "model":{
        "type":"unet.unet",  # Uses a U-Net model
        "params":{  # Activation functions for convolution/deconvolution
            "conv_activation":"ELU",
            "deconv_activation":"ELU"
        }
    }
}
```

Listing 2: .json configuration file.

The above parameters were chosen via repeated trial and error, since training very frequently produced errors at startup (sometimes even when the parameters and all other factors, e.g. dependencies, remained constant after successfully training the model).

Of note is the requirement of .csv files for paths of each mix, KD, SD and HH audio files, as well as their duration (fig. 14 displays their structure. Organisation of these, as well as the csv files' generation, was automated via utility scripts .

```
""" Used to write csv files for training/validation directories given their path.
    """
```

mix_path	HH_path	SD_path	KD_path	duration

Figure 14: Format of the csv files for training and validation sets. The full csvs are not included due to size.

```

import librosa
import os
import csv

path = 'validation_new'
with open('drum_' + path + '.csv', 'w', newline='') as file:
    w = csv.writer(file)
    w.writerow(['mix_path', 'HH_path', 'SD_path', 'KD_path', 'duration'])
    current_row = ['', '', '', '', '']
    for filename in os.listdir('Training\\' + path):
        if "HH" in filename and "#train" not in filename:
            current_row[1] = path + '\\' + filename
        elif "KD" in filename and "#train" not in filename:
            current_row[3] = path + '\\' + filename
        elif "MIX" in filename and "#train" not in filename:
            current_row[0] = path + '\\' + filename
            current_row[4] = librosa.get_duration(filename='Training\\' + path + '\\' + filename)
        elif "SD" in filename and "#train" not in filename:
            current_row[2] = path + '\\' + filename
        else:
            print("Skipped " + filename)
        if current_row[0] and current_row[1] and current_row[2] and current_row[3]:
            print(current_row)
            w.writerow(current_row)
            current_row = ['', '', '', '', '']

    for filename in os.listdir('Training\\' + path):
        if "HH#train" in filename:
            current_row[1] = path + '\\' + filename
        elif "KD#train" in filename:
            current_row[3] = path + '\\' + filename
        elif "MIX#train" in filename:
            current_row[0] = path + '\\' + filename
            current_row[4] = librosa.get_duration(filename='Training\\' + path + '\\' + filename)
        elif "SD#train" in filename:
            current_row[2] = path + '\\' + filename
        else:
            print("Skipped " + filename)
        if current_row[0] and current_row[1] and current_row[2] and current_row[3]:
            print(current_row)
            w.writerow(current_row)
            current_row = ['', '', '', '', '']
```

Listing 3: Script written to generate a csv file from a folder full of dataset audio.

3.3 Implementation

Implementation will be examined backwards, beginning with the completed program then analysing each part of it in sequence, from the beginning to the end of the algorithm.

3.3.1 Main Function

The program was implemented as a simple Python (.py) file that can be run in the command line. In order for the program to run, certain requirements need to be met:

- Spleeter and its dependencies must be installed. Dependencies must be of a specific version lest compatibility issues occur (more on this in section 3.3.3. To use GPU acceleration, CUDA v10.1 must be installed.
- A pretrained model and its configuration file are required. These are already provided. The configuration file can be seen in listing 2.
- The pretrained model must be in a folder named "pretrained_models" and in the same directory the program is run via command line, otherwise Spleeter will be unable to access it.
- If transcribing from a song, the song should ideally only have drums that use the KD, SD and HH, due to these being the only instruments the model was trained on (otherwise other parts of the kit will be forcibly counted as KD, SD or HH). This could of course be extended with a bigger dataset involving more drum parts.

Of note is that there are two variations of the program. song_to_drums.py will provide a MIDI transcription of the drums in a full song that includes other instruments besides drums as well. A second version, wav_to_midi.py, will transcribe an isolated drum track to MIDI. These work in the exact same way, except song_to_drums.py has an extra step of first isolating the drums from the song. Hence, while this dissertation will be following the programming behind song_to_drums.py, everything that is applied to it besides the drum isolation applies exactly the same to wav_to_midi.py.

That being said, if the requirements are satisfied, the program can be run via the following command on the command line:

```
python -m song_to_drums.py -i [input file name] -o [output file name] -c [path to config file]
```

The program itself is structured as follows:

```
from spleeter.separator import Separator
from transcriber import transcribe
from argparse import ArgumentParser
import librosa


def song_to_drums():
    '''Takes any song as input, produces a MIDI transcription of the drums in it.
    '''

    # Parse command line input
    parser = ArgumentParser()
    parser.add_argument('-i', '--ifile')  # Input file
    parser.add_argument('-o', '--ofile')  # Output file
    parser.add_argument('-c', '--config')  # Config file for model (.json).
    Currently drum_config_new.json
    args = parser.parse_args()

    # Prepare spleeter's separator
    separator_song = Separator('spleeter:4stems')

    # Calculate BPM
```

```
    y, sr = librosa.load(args.ifile)
    onset_env = librosa.onset.onset_strength(y, sr=sr)
    bpm = librosa.beat.tempo(onset_envelope=onset_env, sr=sr)

    # Separate stems.
    separator_song.separate_to_file(args.ifile, 'stems')

    # Separate and transcribe the drums.
    separator_drums = Separator(args.config)
    separator_drums.separate_to_file('stems/drums.wav', 'drumparts')
    transcribe('drumparts/HH.wav', 'drumparts/KD.wav', 'drumparts/SD.wav', args.
    ofile, bpm)


# The following is a script safeguard to prevent scripts from behaving strangely.
def main():
    song_to_drums()


if __name__ == '__main__':
    main()
```

Listing 4: The program's main function.

The program can be split up into the following main steps:

1. Parses the command line's input.
2. Estimates a song's *Beats Per Minute (BPM)*. This is done purely via a preset set of functions from the librosa library. This involves performing some onset detection, and using the frequency of onsets to estimate the BPM. The BPM is later used for transcription purposes.
3. Separates the song into drums and the rest of the instruments using Spleeter's pretrained model.
4. Separates the drums into KD, SD and HH using the drums model.
5. Uses the KD, SD, HH and BPM to create a MIDI transcription.

3.3.2 Transcription

While the actual separation model has been already discussed in Spleeter's section, the approach to transcription has yet to be discussed. Transcription is done using the Python library mido[6], which contains functions for reading and writing MIDI files.

The first step in transcribing is converting the obtained onsets, which are in seconds, to MIDI time. MIDI does not set notes to seconds, but rather tempo-dependent ticks. Thus, for transcription purposes, two values need to be found first - *Beats Per Minute (BPM)* and *Pulses Per Quarter-note (PPQ)*[7]. BPM can be found from the original pre-separation drum pattern using librosa via a dedicated BPM detection function. From the BPM, each beat is further subdivided into PPQ ticks, a resolution of which can vary depending on what software is processing the MIDI files. For example, FL Studio's default PPQ is 96. Selection of this is therefore somewhat arbitrary. Since transcription was tested on FL Studio, the PPQ used in this dissertation was 96.

[6]Source code can be found at `https://github.com/mido/mido/`
[7]defined as "ticks per beat" in mido.

Once source separation is complete, predicted onsets are mapped to their respective instrument, thus creating a timeline of which instrument is hit and when, such as in the following example, using one measure of the basic rock groove shown in fig. 6 with simplified timings:

Instrument	HH, KD	HH	HH, SD	HH
Time (seconds)	0.0	0.5	1.0	1.5

With BPM and PPQ in mind and be converted from seconds to MIDI ticks. This involves use of the following equation:

$$ticks = \frac{onset}{60} * PPQ * BPM \tag{5}$$

Applied to the previous example, if we assume the loop's BPM is 130 and we are using FL Studio, with a PPQ of 96, we get the following:

Instrument	HH, KD	HH	HH, SD	HH
Time (ticks)	0	104	208	312

Once the onsets are mapped to their instruments and converted to ticks, the MIDI file can finally be created. MIDI files are based on an ordered message system, where turning notes on and off (alongside other things, such as changing the tempo) are stored as messages, in sequence. Each message contains the message itself (e.g. "note_on"), and the time since the last message was sent. Since the currently mapped tick values are in absolute time, the difference between ticks (delta time) is used instead.

Percussion in MIDI still uses notes, with each being mapped to a particular type of drum sound. There is a large quantity of available MIDI drum notes, but due to the dataset used this dissertation will focus on KD, SD and HH, which are 35, 38 and 42, respectively.

As stated before, messages in MIDI can be used to turn on a note, or to turn it off. Accurate on/off times are important for synthesized notes, or those that can be sustained indefinitely. However, since by nature drums do not generate a sustained sound, the time at which a note is set to turn off is relatively arbitrary compared to the time a note is set to turn on. While turning a note off too soon would cause the drum sound to be cut short, there is no penalty in extending the note's duration further than what a sample would be able to sustain. Hence, notes are set to turn off only if the same one is about to be turned on again, entirely avoiding the issue of cutting notes short unless many of the same note are played in quick succession, which would have resulted in them being cut short anyway to allow for the new ones to be played.

Thus, the complete process of transcribing to a MIDI file given the onsets is as follows:

1. Obtain/decide on a PPQ used in the software the MIDI transcription is going to be used in. (this must be done manually, but every following step is automated)
2. Obtain an estimate of the song/pattern's BPM.
3. If the song/pattern contains non-drum instruments, perform source separation using Spleeter's pre-trained 4-stem model to separate the drums from them.
4. Use trained drum source separation model to separate isolated drum song/pattern into the drum's constituent parts.

5. Detect onsets in each isolated part.
6. Map onsets to their instrument and save in one list.
7. Convert onset time from seconds to ticks.
8. Create a new MIDI file by sending delta time and note on/off messages for each instrument to MIDI channel 10, which is usually reserved for percussion.

```
# Transcribes a set of HH, KD and SD into a midi file.
from onsets import get_onsets, get_delta_times
from mido import Message, MidiFile, MidiTrack, MetaMessage, bpm2tempo
import operator


def calculate_ticks(note_time_ms, bpm, ppq):
    return int(note_time_ms/60000*ppq*bpm)


def onsets_to_tuples(note, onsets, bpm, ppq):
    r_onsets = [calculate_ticks(round(num*1000), bpm, ppq) for num in onsets]
    return [(note, onset) for onset in r_onsets]


def transcribe(hh, kd, sd, output, bpm, ppq=96):
    '''
    Transcribes a set of hh, kd, sd wav files into a midi file.
    '''
    # Initialise midi file
    mid = MidiFile()
    track = MidiTrack()
    tempo_track = MidiTrack()
    mid.tracks.append(tempo_track)
    mid.tracks.append(track)

    # FL Studio uses 96 ticks per beat, hence the default ppq is 96.
    mid.ticks_per_beat = ppq

    # Generate an index of what instrument plays at what time.
    timeline = []

    # HH = 42, KD = 35, SD = 38
    # get_onsets() is a simple function which returns the onsets of a file using
    librosa.
    timeline += onsets_to_tuples(42, get_onsets(hh), bpm, ppq)
    timeline += onsets_to_tuples(35, get_onsets(kd), bpm, ppq)
    timeline += onsets_to_tuples(38, get_onsets(sd), bpm, ppq)

    # We now have a list of tuples, describing what not plays at what time.
    # However, these are not ordered.
    # Since MIDI messages proceed linearly through time, we must first sort them
    timeline.sort(key=operator.itemgetter(1))

    # Since MIDI is based on delta time (time since last message, we must convert
    the timeline to one of delta times.
    delta_timeline = [timeline[0]]
    delta_timeline += [(timeline[n][0], timeline[n][1]-timeline[n-1][1]) for n in
    range(1, len(timeline))]

    # Set the tempo
    tempo_track.append(MetaMessage('set_tempo', tempo=bpm2tempo(bpm)))

```

```
    # Add the MIDI messages in sequence
    for n in delta_timeline:
        track.append(Message('note_off', note=n[0], channel=10, time=n[1]))
        track.append(Message('note_on', note=n[0], channel=10, time=0))

    mid.save(output)


def main():
    path = "midi_groove/rock_groove"
    transcribe(path + '/HH.wav', path + '/KD.wav', path + '/SD.wav', 'trial.mid',
    130)


if __name__ == "__main__":
    main()
```

Listing 5: Code which accomplishes the above

3.3.3 Challenges of Implementing Spleeter

Spleeter as a tool resulted in being somewhat inconsistent and unreliable. Documentation was not very detailed, so many issues that arose when attempting to train a model had to be solved via pure trial and error, sometimes never discovering the reasons behind an error. This unreliability even extended to reproducibility - for example, after a small test model successfully trained using very few steps, increasing the steps and changing nothing else resulted in the program failing to train. When this change was undone, hence returning to the parameters which allowed the model to train, and training was attempted again, the training still failed (even when the previous model was removed).

Spleeter's interplay with mono and stereo files was also hardly documented at all. Training Spleeter with mono inputs did not work when the number of channels in the configuration file was set to 1, (thus representing mono), but did work when it was set to 2.

Another issue caused by the lack of documentation is the requirement for very specific dependencies. While some dependencies are listed via pip when installing Spleeter, others, such as the requirement of CUDA 10.1 in order to enable GPU acceleration, were much more difficult to find. Even more difficult to find were some dependencies that, despite being set to automatically install alongside Spleeter, were of a version that was incompatible with Spleeter. For example, installing the gpu version of Spleeter made it install an incompatible version of Tensorflow. Due to the lack of documentation, this had to be manually found and corrected by installing the correct version of Tensorflow, which was also difficult to find due to the lack of documentation.

Overall, Spleeter had a few idiosyncrasies, some of which were not very clearly documented, resulting in added challenge when attempting to use it.

4 Results

Certain metrics had to be found in order to evaluate the results objectively. It was found that there were two different ways in which the program could be evaluated:

- The quality of source separation could be evaluated using the standard source separation metrics [5] of *Signal to Distortion Ratio (SDR),Source to Spatial Distortion Image (ISR),*

Source to Interference Ratio (SIR) and *Source to Artifacts Ratio (SAR)*. The higher each number, the closer to the source (and therefore the more accurate) a prediction is.

- The accuracy of the onsets could be evaluated via using a confusion matrix of *True Positives (TP)*, *False Positives (FP)* and *False Negatives (FN)* to calculate an *F-Measure*, which would give the accuracy a rating between 0 and 1 (1 being perfectly accurate).

4.1 Evaluation: Source Separation Metrics

Evaluation using source separation metrics was done using the museval Python package [6]. Code used to derive these results can be seen in listing 6.

```
""" Evaluate a directory using museval, given a ground truth directory and an
    estimate directory.
    Files in both directories must have matching names. """

import museval
import os

reference_path = "../Training/test_new"
results_path = "../Results_new"

results = museval.EvalStore(frames_agg='median', tracks_agg='median')

for file in os.listdir(reference_path):
    print("Evaluating " + file)
    ref_dir = os.path.join(reference_path, file)
    res_dir = os.path.join(results_path, file)
    results.add_track(museval.eval_dir(ref_dir, res_dir))

print(results)
results.save('train.pandas')
```

Listing 6: Code used to evaluate source separation.

The results obtained from running the evaluation are as follows:

HH	ISR	6.632452
	SAR	6.868449
	SDR	4.577312
	SIR	12.924094
KD	ISR	29.984024
	SAR	24.659818
	SDR	23.707093
	SIR	34.720290
SD	ISR	15.036854
	SAR	11.730660
	SDR	10.298638
	SIR	16.220415

As a comparison, Spleeter's pre-trained models for separating vocals, bass, drums and other in songs show SDRs of 4-7. Given the relative simplicity of this task, the higher SDR for KD and SD is not completely unprecedented. However, this does show that source separation for KD and SD are very effective, while HH separation is relatively weak. This results in a decent portion of KD and SD hits being included in the HH's isolated file.

4.2 Evaluation: Confusion Matrix

More important than source separation metrics is the accuracy of the model's transcription. Evaluation of the model's effectiveness at actually transcribing notes can be done via a simple confusion matrix to see whether notes were correctly or incorrectly detected. This was done by detecting the onsets in an original isolated KD, SD or HH (used as ground truth), comparing them to onsets detected in the resulting separated KD, SD or HH (the predictions), and finding the following:

- Predicted onsets were considered *True Positives (TP)* if they were within a selected tolerance window of ground truth onsets.
- Predicted onsets were considered *False Positives (FP)* if they did not correspond to any ground truth onsets.
- Ground truth onsets that were not predicted at all were considered *False Negatives*.
- An attempt was made to categorise *True Negatives (TN)* as well, best defined by an example: when looking at the onsets of a KD, if there is no KD onset at a time t but it is known that there is a SD onset at time t a TN is when no KD onset is detected at time t. In effect, TNs occur if no other instruments are "bleeding" in when it is known that they could.

From these, *Precision*, *Recall* and *F-Measure* can be calculated thus:

- Precision: $\frac{TP}{TP+FP}$
- Recall: $\frac{TP}{TP+FN}$
- F-measure: $\frac{2*TP}{2*TP+FP+FN}$
- Accuracy (considered experimental): $\frac{TP+TN}{TP+TN+FP+FN}$

Before all of these can be found, however, a *tolerance window* must be selected. Onset detection can result in highly specific numbers down to multiple significant figures. However, there is hardly any practical difference between, for example, an onset being at 2.5386s and one at 2.5385s. Despite this difference being completely negligible, this would still be evaluated as an onset being calculated incorrectly due to the two not being exactly identical. The tolerance window is used to combat this; onsets close enough to each other by a particular amount denoted by the tolerance window are considerd to be identical. Finding a suitable tolerance window is quite important, as too high a tolerance window would mean onsets that are very clearly different from each other are still considered identical, while too low a tolerance window would greatly underestimate the accuracy of the model by considering functionally identical onsets as different.

Three tolerance windows were selected for these onsets in order to evaluate the model's performance for different applications:

For the purposes of basic transcription, adaptation, and remixing within quantized MIDI software, a tolerance of 100ms (aka rounding the onsets to the nearest 0.1s) was used. This is due to the fact that, to the human ear, inter-onset intervals lower than 100ms are virtually indistinguishable from each other [17]. Since, in this case, the model is being applied towards obtaining drum patterns and re-applying or modifying them to create new music, it is reasonable to assume that any further accuracy beyond that of what a human would be capable of distinguishing is not necessary, especially since these onsets are likely to be quantized within MIDI software, removing

any possible inaccuracies. It can be said that this tolerance window was selected on a musical basis.

The second tolerance window selected was 50ms. This was for any applications that would require a higher accuracy (such as accurate real-time transcription for the purposes of educational MIDI-based software), as well as to follow previous ADT literature [3] [18] [27] which was based around extrapolating from the lowest interval at which humans can distinguish two notes as separate, which is around 8-10ms [14]. It can be said that this tolerance window was selected on a more physiological basis.

The final tolerance window selected was 20ms. This was purely for the sake of comparison with the others, as, while it is suggested that it is the ideal tolerance window for precise reproduction of a human ear [28], such a high level of accuracy is beyond necessary for the applications of this model.

Code used to obtain the results can be seen in listing 7.

```
import librosa
import librosa.display
import matplotlib.pyplot as plt
import os
import math


def sensitivity(tp, fn):
    return tp/(tp + fn)


def precision(tp, fp):
    return tp/(tp + fp)


def accuracy(tp, tn, fp, fn):
    return (tp + tn)/(tp + tn + fp + fn)


def f_measure(tp, fp, fn):
    return 2*tp/(2*tp + fp + fn)


# Rounds x to the nearest a, used as a tolerance window.
def round_nearest(x, a):
    return round(round(x / a) * a, -int(math.floor(math.log10(a))))


tp = 0      # Is the note there in the ground truth and there in the prediction?
tn = 0      # Is a note correctly not bleeding in from a different instrument when
     it could be?
fp = 0      # Is a note detected when it shouldn't be here?
fn = 0      # Is a note supposed to be detected here, but isn't?

pred_files = "../Results_new"
truth_files = "../Training/test_new"

for file in os.listdir(pred_files):

    # Store all onsets so that true negatives can be calculated by seeing if
    onsets that could cause bleeding don't

    hh_pred = []
```

```
    hh_truth = []
    kd_pred = []
    kd_truth = []
    sd_pred = []
    sd_truth = []

    for subfile in os.listdir(os.path.join(pred_files, file)):

        print("Calculating " + subfile + "...")

        # Find onsets
        x, sr = librosa.load(os.path.join(pred_files, file, subfile))
        y, sr2 = librosa.load(os.path.join(truth_files, file, subfile))

        onset_frames_x = librosa.onset.onset_detect(x, sr=sr, wait=1, pre_avg=1, post_avg=1, pre_max=1, post_max=1)
        onset_times_x = librosa.frames_to_time(onset_frames_x)

        onset_frames_y = librosa.onset.onset_detect(y, sr=sr2, wait=1, pre_avg=1, post_avg=1, pre_max=1, post_max=1)
        onset_times_y = librosa.frames_to_time(onset_frames_y)

        # Tolerance window
        round_to = 0.02

        # Store predictions and ground truth for each instrument.
        if "HH" in subfile:
            hh_pred = [round_nearest(num, round_to) for num in onset_times_x]
            hh_truth = [round_nearest(num, round_to) for num in onset_times_y]
        elif "KD" in subfile:
            kd_pred = [round_nearest(num, round_to) for num in onset_times_x]
            kd_truth = [round_nearest(num, round_to) for num in onset_times_y]
        elif "SD" in subfile:
            sd_pred = [round_nearest(num, round_to) for num in onset_times_x]
            sd_truth = [round_nearest(num, round_to) for num in onset_times_y]
        else:
            print("Whoops, seems like something was neither HH, KD nor SD!")
            break

    # Calculate true positive rate, false positive rate
    def positives(predictions, truth):
        global tp
        global fp
        for num in predictions:
            if num in truth:
                tp += 1

            elif num not in truth:
                fp += 1

            else:
                print("Ooopsie there's some kinda error here")
                break

    positives(hh_pred, hh_truth)
    positives(kd_pred, kd_truth)
    positives(sd_pred, sd_truth)

    # Calculate false negative rate
    def f_negatives(predictions, truth):
        global fn
        for num in truth:
```

```
            if num not in predictions:
                fn += 1

    f_negatives(hh_pred, hh_truth)
    f_negatives(kd_pred, kd_truth)
    f_negatives(sd_pred, sd_truth)

    # Calculate true negative rate
    def t_negatives(predictions, truth, bleed1, bleed2):
        # If an onset is in bleed, but not in predictions nor truth, then it can be considered a true negative
        # That'd mean the source separation was successful enough here
        global tn

        # First, combine bleed onsets into one list
        # Adapted from https://stackoverflow.com/questions/1319338/combining-two-lists-and-removing-duplicates-without-removing-duplicates-in-orig
        bleed = list(bleed1)
        bleed.extend(b for b in bleed2 if b not in bleed)

        # If something is in bleed but not in truth nor predictions, then it has been correctly filtered out and is a true negative
        for onset in bleed:
            if (onset not in truth) and (onset not in predictions):
                tn += 1

    t_negatives(hh_pred, hh_truth, sd_truth, kd_truth)
    t_negatives(sd_pred, sd_truth, hh_truth, kd_truth)
    t_negatives(kd_pred, kd_truth, hh_truth, sd_truth)

# Print results
print("Total (sans TN): " + str(tp + fp + fn))
print("TP: " + str(tp))
print("TN: " + str(tn))
print("FP: " + str(fp))
print("FN: " + str(fn))

print("Sensitivity: " + str(sensitivity(tp, fn)))
print("Precision: " + str(precision(tp, fp)))
print("Accuracy: " + str(accuracy(tp, tn, fp, fn)))
print("F-measure: " + str(f_measure(tp, fp, fn)))

# Plot onsets on a graph
```

Listing 7: Code used to evaluate drum transcription.

The evaluated results, for all onsets, was as follows:

	100ms	50ms	20ms
TP	383	312	265
FP	166	237	284
FN	38	93	139
TN	330	455	496
Precision	0.70	0.56	0.48
Recall	0.91	0.77	0.66
F-measure	0.79	0.65	0.56
Accuracy	0.78	0.70	0.64

For very low tolerance window transcriptions, the model is not very accurate. However, for the purposes of simple transcription, it performs reasonably well. Of note is that Recall is much

higher than Precision for all tolerance windows, and it can be seen that false positives are far more likely than false negatives. This is likely due to the different parts of the drum bleeding into each other during source separation. For example, the HH file may end up containing some of the KD's hits as well, the onsets of which are detected as belonging to the HH instead of the KD. Given the SIR values computed earlier, it can be safe to assume that the majority of false positives result due to KD and SD bleeding into HH, since there was very little bleeding into the KD and SD channels.

It is also interesting to note the increase in TN results as the tolerance window is lower. This shows the effect the tolerance window has in reducing the total amount of onsets by equating them as identical.

5 Limitations

As mentioned in the results, the main limiting factor for obtaining a high F-measure in this specific case was KD and SD drum bleeding into the HH channel. This is due to the other two having a very high source-to-interference ratio, hence experiencing less bleed. This is likely due to the fact that, as seen in the drum spectrogram in fig. 6, KD and SD are much more visible than HH and therefore easier to be detected when struck alongside other instruments. In comparison, HH are more difficult to distinguish, especially when other parts of the drum are played in tandem with them.

There are a number of other limitations and challenges with the approach that was taken. The first, and most common limitation with most of ADT (and activation-based approaches in general) is the strong reliance on good data to train from. In this case, the IDMT-SMT dataset can be considered relatively small, as it has a total duration of approximately 2:10 hours and focuses on just kick drum, snare drum and hi-hat. The lack of proper mixing of KD, SD and HH even further reduced size of the dataset and the lack of k-fold cross validation further decreased the possible accuracy of the model. The limitations of this dataset became apparent during the evaluation of onset detection, where false positives far outweighed false negatives as an area of error likely due to imperfect source separation resulting in instruments bleeding into each other.

Bigger and more varied datasets could allow for far stronger results. While there had not been many datasets available until recently, TensorFlow's Magenta project[8] has produced the Groove MIDI dataset [12], composed of 13.6 hours of MIDI-annotated drum recordings, and the Expanded Groove MIDI dataset [4], composed of 444 hours of MIDI-annotated drum recordings. Both include more parts of a drum kit as well, such as multiple toms and crash cymbals. Of course, the limitation that then arises with larger datasets is one of storage. While the Groove MIDI dataset is a reasonable 4.76GB in size, the Expanded Groove MIDI dataset is a far less convenient 132GB.

Another limitation of this approach is that it combines different processes (source separation and onset detection), each having its own accuracies/inaccuracies which will invariably affect the other in some way. Inaccuracies in source separation will negatively impact onset detection, as different instruments will bleed into each others' tracks and possibly have their onsets detected, while inaccurate onset detection can lower the usability of even the best source separation models by not detecting relatively obvious onsets. Furthermore, there are many parameters to be used for Spleeter's source separation as well as librosa's onset detection, so finding an optimum for both

[8] `https://magenta.tensorflow.org/`

is a non-trivial task. This results in the overall effectiveness being limited by the effectiveness of the source separation as well as onset detection.

This is compounded by the fact that, for songs that use instruments besides drums, source separation must be performed to first isolate the drums and then performed again as part of the transcription process, making the proposed pipeline dependent on the effectiveness of the onset detection and two different source separation models. Even the current, competing with state-of-the-art pre-trained 4-stem model Spleeter has will still exhibit a few other instruments occasionally bleeding into the drum track, which is likely enough to reduce the transcription's accuracy.

Limitations are also present in the onset detection as well. Instead of using the onsets included with the dataset as ground truth, the onsets for ground truth wav files were estimated using librosa's onset detection features. Hence, it may well be that the onsets considered ground truth are actually not ground truth at all, since the onset detection may have been inaccurate. Furthermore, the onset detection process itself could have perhaps used a more sophisticated implementation. As seen by the fact that the effectiveness of onset detection could be evaluated via a confusion matrix, onset detection can be reduced to a classification problem. Hence, there are various machine-learning-based approaches that could have been used to tackle the onset detection problem, some of which may have been more accurate at detecting onsets.

The last limitation is in transcription. The conversion from absolute time, denoting the onsets, to MIDI time is an approximation based on tempo. The accuracy of the transcription is therefore dependent on the accuracy of BPM estimation, as estimating a different tempo can result in some inaccuracies in the final MIDI file's timings. While manually adjustable in the application of adapting and remixing patterns in a DAW to create music, this can be insufficient for other applications, such as MIDI timing-based education.

6 Conclusions & Future Work

This project has devised a program for *Automatic Drum Transcription (ADT)* consisting of separating a drum track from the rest of the instruments in a song/sample and then using another source separation-based approach to transcribe the drum track into MIDI format. While the transcription works to a certain degree of accuracy, with an optimistic F-measure of 0.79, the usage of multiple layers of source separation combined with source detection and having to estimate BPM leads to inaccuracies compounding themselves, revealing this to likely not be the most efficient approach. However, with a large enough dataset such as the one provided by the Groove and Expanded Groove MIDI datasets, there is certainly potential for source separation to at the very least assist in enhancing the accuracy of ADT, as the parts composing a drum kit can certainly be differentiated from each other using current source separation techniques.

Future work in ADT is encouraged to utilise the Expanded Groove MIDI dataset for training purposes, and look to combining source separation with other methods of ADT to perhaps produce more accurate onset predictions. Moreover, while the Groove datasets are definitely a step above IDMT-SMT in their size, an area that datasets are still lacking in is drum transcriptions alongside full tracks of music containing other instruments, since it may be worth testing whether an activation-based network can be trained to directly separate different parts of a drum kit not just from each other but from the rest of the instruments in a song, instead of relying on two layers of source separation.

7 Reflection

Much has been learned over the course of this dissertation. Most importantly, one must select the tools they use wisely - enough issues with Spleeter and its unreliability arose that a large amount of time was spent debugging issues that sometimes had already been fixed. Even at the time of writing, Spleeter was experiencing some dependency issue that prevented its use, despite the same settings having been used to successfully fully train a model.

Another issue that arose which could have been dealt with more effectively was that of time management. Most of the work on this dissertation was done in large, somewhat spontaneous bursts instead of small constant sessions interspersed with sizeable periods of procrastination, which led to a lot of falling behind on schedule.

Were time to have been better managed, this dissertation could have improved upon many of its limitations, such as implementation of k-fold cross-validation; usage of other datasets such as the Groove MIDI datasets; inclusion of more parts of a drum kit than just HH, KD, SD; a more sophisticated approach for onset detection; and more depth in the writing of this dissertation.

Besids the time management issue, there were also a few careless mistakes which would likely not have happened had more detailed attention been given to the methods used. This mainly concerns a proper preprocessing of the dataset that mixes HH, KD and SD in more ways leading to a training dataset three times larger; and usage of the dataset's own onset annotations instead of using a fallible onset detection algorithm provided by a library.

However, some diligence was practiced, to an extent. Work sessions were always successfully tracked, the time and results of which were all noted down after every single session without fail. This became extremely useful when debugging the constant Spleeter issues, as previous approaches had all been written down leading to a reliable documentation that could be drawn from.

Furthermore, a lot was learned on the subject matter of signal processing, MIR, ADT, and a whole variety of other topics, and the creation of the program resulted in a lot of practice of known programming concepts and the learning of new concepts and libraries.

Overall, the dissertation has been a very positive learning experience and a powerful reminder of the importance of time management.

References

[1] Mette Pedersen Anne B Alexius. "Aspects of Adolescence and Voice: Girls versus Boys ? A Review". In: *Journal of Child and Adolescent Behaviour* 03.03 (2015). DOI: 10.4172/2375-4494.1000211.

[2] Emmanouil Benetos et al. "Automatic Music Transcription: An Overview". In: *IEEE Signal Processing Magazine* 36.1 (2019), pp. 20–30. ISSN: 1053-5888. DOI: 10.1109/MSP.2018.2869928.

[3] C. Dittmar and Daniel Gärtner. "Real-Time Transcription and Separation of Drum Recordings Based on NMF Decomposition". In: *Proc. Intl. Conf. on Digital Audio Effects (DAFx)*. 2014, pp. 187–194.

[4] Lee Callender, Curtis Hawthorne, and Jesse Engel. *Improving Perceptual Quality of Drum Transcription with the Expanded Groove MIDI Dataset*. Examples available at https://goo.gl/magenta/e-gmd-examples. 2020-04-01. URL: https://arxiv.org/pdf/2004.00188.

[5] Emmanuel Vincent, Rémi Gribonval, and Cédric Févotte. "Performance measurement in blind audio source separation". In: *IEEE Transactions on Audio, Speech and Language Processing* (2006), pp. 1462–1469.

[6] Fabian-Robert Stöter et al. *sigsep/sigsep-mus-eval: museval 0.4.0*. Zenodo, 2021. DOI: 10.5281/ZENODO.4486535.

[7] Joshua Fineberg. "Guide to the basic concepts and techniques of spectral music". In: *Contemporary Music Review* 19.2 (2000), pp. 81–113. ISSN: 0749-4467. DOI: 10.1080/07494460000640271.

[8] O. Gillet and G. Richard. "Transcription and Separation of Drum Signals From Polyphonic Music". In: *IEEE Transactions on Audio, Speech, and Language Processing* 16.3 (2008), pp. 529–540. ISSN: 1558-7916. DOI: 10.1109/TASL.2007.914120.

[9] Henry Heffner and Rickye Heffner. "Hearing Ranges of laboratory animals". In: *Journal of the American Association for Laboratory Animal Science : JAALAS* 46 (2007). 02, pp. 20–22.

[10] Romain Hennequin et al. "Spleeter: a fast and efficient music source separation tool with pre-trained models". In: *Journal of Open Source Software* 5.50 (2020), p. 2154. DOI: 10.21105/joss.02154.

[11] A. Jansson et al. "Singing voice separation with deep U-Net convolutional networks". en. In: *Proceedings of the International Society for Music* (2017), pp. 323–332. URL: https://openaccess.city.ac.uk/id/eprint/19289/.

[12] Jon Gillick et al. "Learning to Groove with Inverse Sequence Transformations". In: *International Conference on Machine Learning (ICML)*. 2019.

[13] Anssi Klapuri and Manuel Davy. *Signal processing methods for music transcription*. Springer, 2006.

[14] R. Y. Litovsky et al. "The precedence effect". eng. In: *The Journal of the Acoustical Society of America* 106.4 Pt 1 (1999). Journal Article Research Support, U.S. Gov't, Non-P.H.S. Research Support, U.S. Gov't, P.H.S., pp. 1633–1654. ISSN: 0001-4966. DOI: 10.1121/1.427914. eprint: 10530009.

[15] Alexey Ozerov et al. "Adaptation of Bayesian Models for Single-Channel Source Separation and its Application to Voice/Music Separation in Popular Songs". In: *IEEE Transactions on Audio, Speech, and Language Processing* 15.5 (2007), pp. 1564–1578. ISSN: 1558-7916. DOI: 10.1109/TASL.2007.899291.

[16] Jouni Paulus. "Signal Processing Methods for Drum Transcription and Music Structure Analysis". en. PhD Thesis. Tampere University of Technology, 2009. URL: https://trepo.tuni.fi/handle/10024/115014.

[17] D. J. Povel. "A theoretical framework for rhythm perception". eng. In: *Psychological Research* 45.4 (1984). PII: BF00309709 Journal Article, pp. 315–337. ISSN: 0340-0727. DOI: 10.1007/BF00309709. eprint: 6728975.

[18] R. Stables, J. Hockmann, and C. Southall. *Automatic Drum Transcription using Bi-directional Recurrent Neural Networks*. 2016. URL: http://www.open-access.bcu.ac.uk/4101/1/automatic%20drum%20transcription.pdf.

[19] Z. Rafii and B. Pardo. "REpeating Pattern Extraction Technique (REPET): A Simple Method for Music/Voice Separation". In: *IEEE Transactions on Audio, Speech, and Language Processing* 21.1 (2013), pp. 73–84. ISSN: 1558-7916. DOI: 10.1109/TASL.2012.2213249.

[20] Zafar Rafii et al. *MUSDB18 - a corpus for music separation*. 2017. DOI: 10.5281/ZENODO.1117372.

[21] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. conditionally accepted at MICCAI 2015. 2015-05-18. URL: http://arxiv.org/pdf/1505.04597v1.

[22] J. Salamon et al. "Melody Extraction from Polyphonic Music Signals: Approaches, applications, and challenges". In: *IEEE Signal Processing Magazine* February 2014 (2014), pp. 118–134. ISSN: 1053-5888.

[23] Paris Smaragdis et al. "Static and Dynamic Source Separation Using Nonnegative Factorizations: A unified view". In: *IEEE Signal Processing Magazine* 31.3 (2014), pp. 66–75. ISSN: 1053-5888. DOI: 10.1109/MSP.2013.2297715.

[24] John Tyrrell and Stanley Sadie. *The new Grove dictionary of music and musicians*. 2nd ed. London: Macmillan, 2001. ISBN: 0333608003.

[25] Shankar Vembu and Stephan Baumann. "Separation of Vocals from Polyphonic Audio Recordings". In: 01. 2005, pp. 337–344.

[26] Tuomas Virtanen. "Monaural Sound Source Separation by Nonnegative Matrix Factorization With Temporal Continuity and Sparseness Criteria". In: *IEEE Transactions on Audio, Speech, and Language Processing* 15.3 (2007), pp. 1066–1074. ISSN: 1558-7916. DOI: 10.1109/TASL.2006.885253.

[27] Chih-Wei Wu and Alexander Lerch. "Drum Transcription using Partially Fixed Non-Negative Matrix Factorization with Template Adaptation". In: 10. 2015.

[28] Chih-Wei Wu et al. "A Review of Automatic Drum Transcription". In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 26.9 (2018), pp. 1457–1483. ISSN: 2329-9290. DOI: 10.1109/TASLP.2018.2830113.