Dennis’ Drum Dream

Demo Vídeo: https://youtu.be/CdPDfgKs-dQ

Group members: 111511132 陳冠廷
111511156 葉宗翰
111511217 胡家銓
111550114 趙堉安
111550126 金以凡

A. Introduction

In recent years, with the invention of online music streaming applications, the music market is experiencing an unprecedented surge in growth. For people who are interested in playing instrument, they might wish that there are a sheet music for the instrument they play, for example, sheet music for guitar, sheet music for piano, sheet music for drum, etc.

To meet the need of those people, we used a python program to output the sheet music for drum. For a given audio file, our program can automatically separate the audio into “drum”, “bass”, “vocal” and “other”, four types of audio track. Since only output the output sheet music cannot properly demonstrate the result, we also used Arduino to produce a output system. We used several servo motor to simulate the drum beat, and a light matrix to simulate the melody.

B. Libraries we use

a. Demucs

"Demucs" stands for "Deep Extractor for Music Sources." It is a Python library designed for audio source separation, specifically aimed at isolating individual sound sources from mixed audio signals. It employs deep learning techniques to achieve the separation of different audio sources.

To thoroughly introduce Demucs, I’ll first talk about some techniques used in building the training model of Demucs.

1. Wave-U-Net

Wave-U-Net is a convolutional neural network applicable to audio source separation tasks, it uses a 5-5 encoder-decoder pair to separate the target audio. The original U-Net is used for image analysis, it is able to automatically divide the image into several faces and draw the edge. Wave-U-Net is based on the U-Net architecture and is designed for one-dimensional time domain tasks. the model processes audio signals at different scales by utilizing downsampling and upsampling blocks with one-dimensional convolutions.[1]

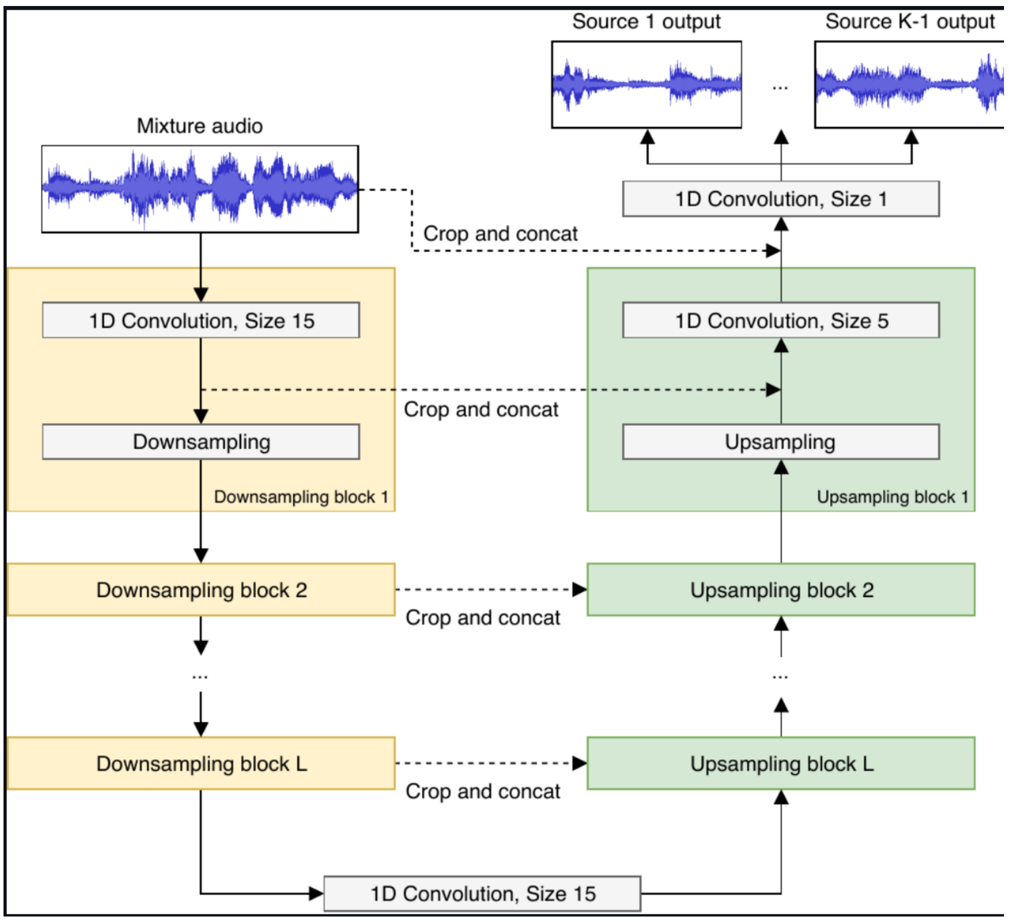


Figure 1: Wave-U-Net structure

2. **bi-LSTM**

"bi-LSTM" stands for "Bi-directional Long Short-Term Memory", it is a kind of RNN(Recurrent neural network). LSTM can proceed a series of data, multiple input and has memory. bi-LSTM connects two LSTM networks, one proceeds the input forward, the other proceed backward. This improve the accuracy of the prediction when the message is relevant to the context before and after.2

[1] Wave-U-Net, https://github.com/f90/Wave-U-Net

[2] 2. bi-LSTM, https://ithelp.ithome.com.tw/articles/10298511?sc=iThelpR

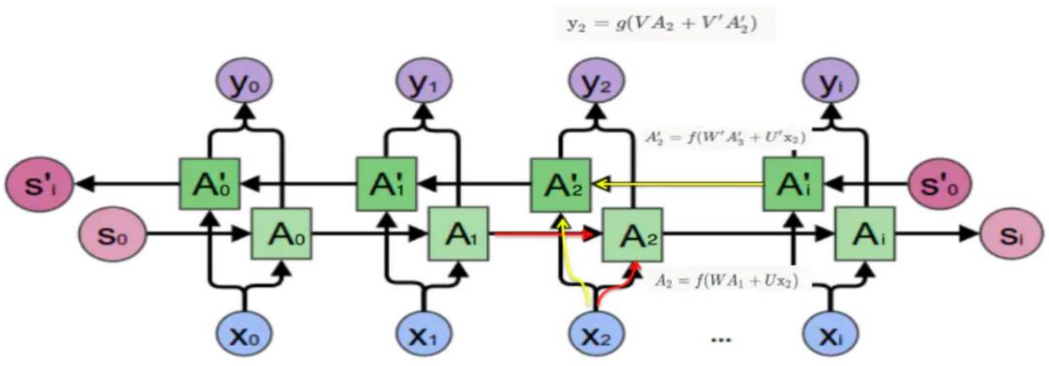


Figure 2: bi-LSTM structure

3. Transformer model

Transformer model is a newly developed neural network by Ashish Vaswani in 2017. Transformer introduced a new concept: “Attention”. Transformer model is originally developed for language analysis, language analysis is usually contextually relevant, so for traditional model such as CNN, RNN, a label for each word is required, which will cost lots of memory and time training. By adding “Multi-headed Attention” and “self-Attention” items in the model, we can find a word’s meaning by other words in the same sentence, since the model is self-attention.

4. STFT and ISTFT

STFT and ISTFT stand for “Short-Time Fourier Transform” and “Inverse Short-Time Fourier Transform” The Short-Time Fourier Transform is a mathematical tool used for analyzing the frequency content of a signal as it changes over time. It is obtained by applying the Fourier Transform to localized segments of a signal. The original Fourier Transform is given by

$$F(z) = \int_{-\infty}^{\infty} f(t)e^{-2\pi itz}\,dz$$

And STFT is given by

$$F(t,f) = \int_{-\infty}^{\infty} f(\tau) \cdot w(\tau - t) \cdot e^{-j2\pi f\tau} d\tau$$

The STFT provides a time-frequency representation of the signal, by applying the STFT at different time points, we can obtain a spectrogram, which illustrates how the frequency content of the signal changes over time. The computational implementation of the STFT involves discretizing the time and frequency domains, resulting in a matrix of complex values. This matrix is often referred to as the time-frequency representation or spectrogram of the signal.

STFT is often be compared to FFT(Fast Fourier transform), since they both are essential tools in signal processing. However they have different targets. STFT focuses on capturing time-varying frequency information by segmenting signals into short intervals, providing a dynamic spectral view. On the other hand, FFT efficiently computes the static frequency content of an entire signal.

In Demucs, it combined the four techniques we mentioned above. For the architecture, Demucs is inspired by the Wave-U-Net but use a transformer model to take place of the fifth encoder-decoder set, it also use bi-LSTM to connect each encoder and decoder. By doing this, Demucs can take advantage of bi-LSTM and transformer, with the use of the self-attention, Demucs can separate the audio file more properly, since an audio file is often autocorrelation.

Besides, Demucs used two U-Net in the model, one is for time domain, the other is for frequency domain. To let the input audio file denote in time and frequency, it does STFT to the input audio file first, after the separation, use ISTFT to transfer the audio file into correct file type.[3]

[3] HYBRID TRANSFORMERS FOR MUSIC SOURCE SEPARATION, Simon Rouard, Francisco Massa, Alexandre D´efossez, Meta AI

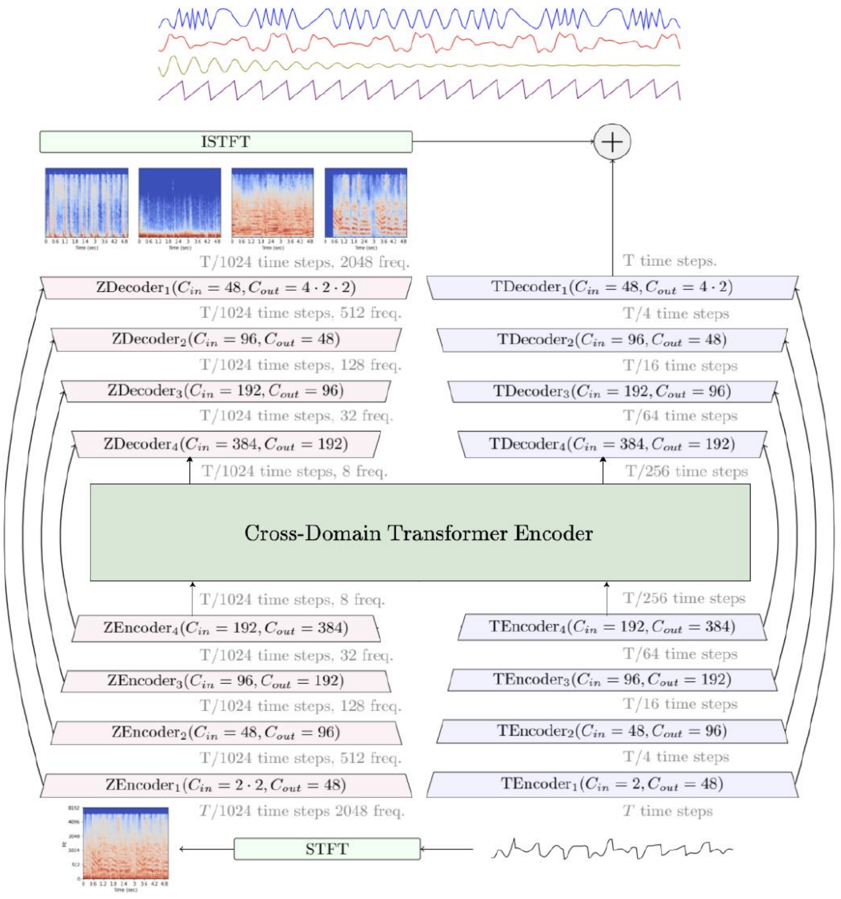

ISTFT
T/1024 time steps, 2048 freq.
T time steps.
$\text{ZDecoder}_1(C_{in} = 48, C_{out} = 4 \cdot 2 \cdot 2)$
$\text{TDecoder}_1(C_{in} = 48, C_{out} = 4 \cdot 2)$
T/1024 time steps, 512 freq.
T/4 time steps
$\text{ZDecoder}_2(C_{in} = 96, C_{out} = 48)$
$\text{TDecoder}_2(C_{in} = 96, C_{out} = 48)$
T/1024 time steps, 128 freq.
T/16 time steps
$\text{ZDecoder}_3(C_{in} = 192, C_{out} = 96)$
$\text{TDecoder}_3(C_{in} = 192, C_{out} = 96)$
T/1024 time steps, 32 freq.
T/64 time steps
$\text{ZDecoder}_4(C_{in} = 384, C_{out} = 192)$
$\text{TDecoder}_4(C_{in} = 384, C_{out} = 192)$
T/1024 time steps, 8 freq.
T/256 time steps
Cross-Domain Transformer Encoder
T/1024 time steps, 8 freq.
T/256 time steps
$\text{ZEncoder}_4(C_{in} = 192, C_{out} = 384)$
$\text{TEncoder}_4(C_{in} = 192, C_{out} = 384)$
T/1024 time steps, 32 freq.
T/64 time steps
$\text{ZEncoder}_3(C_{in} = 96, C_{out} = 192)$
$\text{TEncoder}_3(C_{in} = 96, C_{out} = 192)$
T/1024 time steps, 128 freq.
T/16 time steps
$\text{ZEncoder}_2(C_{in} = 48, C_{out} = 96)$
$\text{TEncoder}_2(C_{in} = 48, C_{out} = 96)$
T/1024 time steps, 512 freq.
T/4 time steps
$\text{ZEncoder}_1(C_{in} = 2 \cdot 2, C_{out} = 48)$
$\text{TEncoder}_1(C_{in} = 2, C_{out} = 48)$
T/1024 time steps 2048 freq.
T time steps
STFT

Figure 3: Demucs

b. Librosa

Librosa is a Python library designed for audio and music analysis. It allows users to easily load audio files, compute various audio features, visualize spectrograms, and perform tasks such as pitch estimation, beat tracking, and more. It offers a convenient and efficient interface for working with audio data in Python.[4]

c. Pytorch

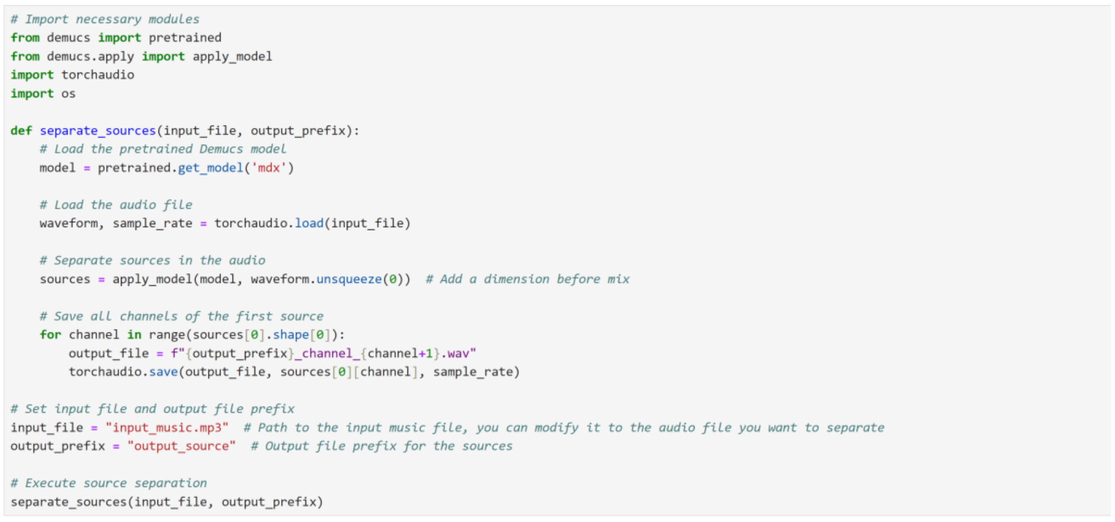
PyTorch is an open-source deep learning framework developed by Facebook. It provides a set of functions and modules to build and train neural networks, which can be used to create models for tasks like speech recognition or music analysis.[5]

[4] McFee, Brian, Colin Raffel, Dawen Liang, Daniel PW Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. "librosa: Audio and music signal analysis in python." In Proceedings of the 14th python in science conference, pp. 18-25. 2015.

[5] Pytorch, https://pytorch.org/audio/main/

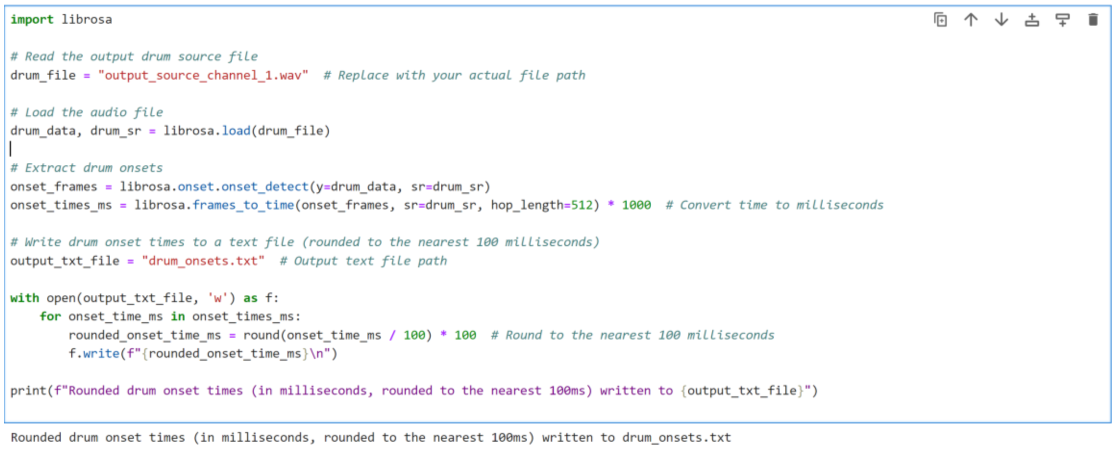
C. Code explanation

a. Audio analysis



```python
# Import necessary modules
from demucs import pretrained
from demucs.apply import apply_model
import torchaudio
import os

def separate_sources(input_file, output_prefix):
    # Load the pretrained Demucs model
    model = pretrained.get_model('mdx')

    # Load the audio file
    waveform, sample_rate = torchaudio.load(input_file)

    # Separate sources in the audio
    sources = apply_model(model, waveform.unsqueeze(0))  # Add a dimension before mix

    # Save all channels of the first source
    for channel in range(sources[0].shape[0]):
        output_file = f"{output_prefix}_channel_{channel+1}.wav"
        torchaudio.save(output_file, sources[0][channel], sample_rate)

# Set input file and output file prefix
input_file = "input_music.mp3"  # Path to the input music file, you can modify it to the audio file you want to separate
output_prefix = "output_source"  # Output file prefix for the sources

# Execute source separation
separate_sources(input_file, output_prefix)
```

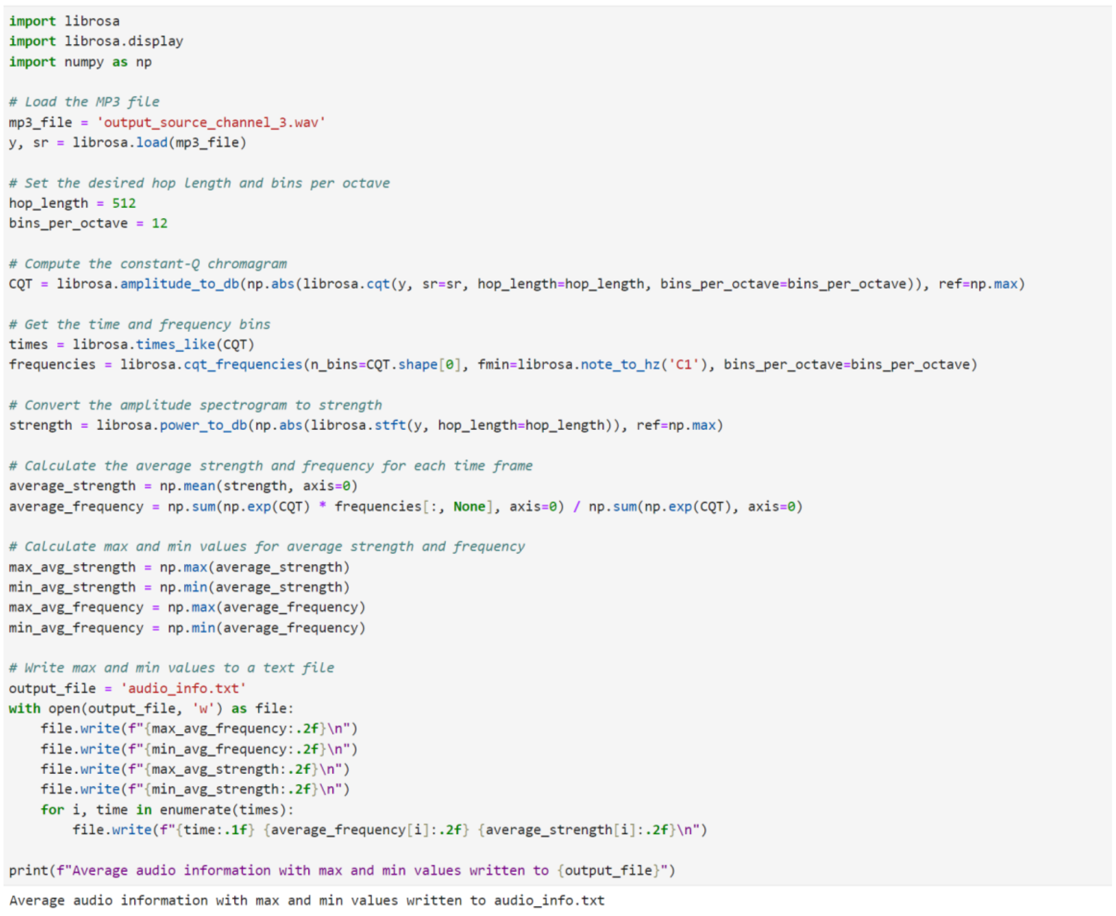
First, we use "Pytorch" library to load the .mp3 input file. Then applied the pre-trained model in Demucs library to the input file. After the model separated the input file, we save the output file prefix.



```python
import librosa

# Read the output drum source file
drum_file = "output_source_channel_1.wav"  # Replace with your actual file path

# Load the audio file
drum_data, drum_sr = librosa.load(drum_file)

# Extract drum onsets
onset_frames = librosa.onset.onset_detect(y=drum_data, sr=drum_sr)
onset_times_ms = librosa.frames_to_time(onset_frames, sr=drum_sr, hop_length=512) * 1000  # Convert time to milliseconds

# Write drum onset times to a text file (rounded to the nearest 100 milliseconds)
output_txt_file = "drum_onsets.txt"  # Output text file path

with open(output_txt_file, 'w') as f:
    for onset_time_ms in onset_times_ms:
        rounded_onset_time_ms = round(onset_time_ms / 100) * 100  # Round to the nearest 100 milliseconds
        f.write(f"{rounded_onset_time_ms}\n")

print(f"Rounded drum onset times (in milliseconds, rounded to the nearest 100ms) written to {output_txt_file}")
```

```
Rounded drum onset times (in milliseconds, rounded to the nearest 100ms) written to drum_onsets.txt
```

The output file contains four channels, which are "drum", "bass", "other" and "vocal" respectively, since our goal of this project is to pick the drum beat out of a mixed audio file, so we use channel 1 as the output. Then use the Librosa library to write the .wav file into a text file which contains a set of data indicate the moment of the drumbeats.

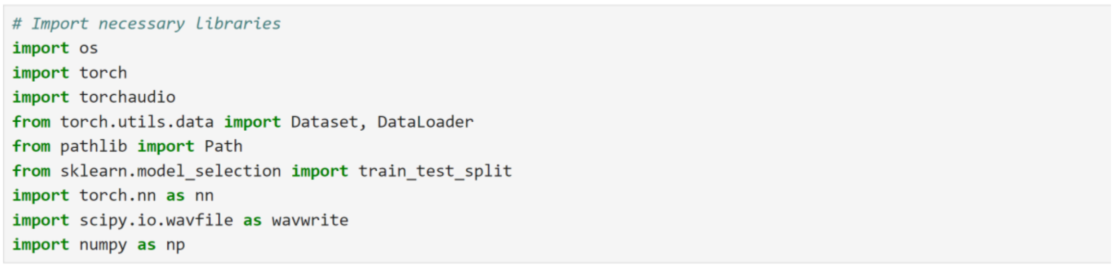


```
import librosa
import librosa.display
import numpy as np

# Load the MP3 file
mp3_file = 'output_source_channel_3.wav'
y, sr = librosa.load(mp3_file)

# Set the desired hop length and bins per octave
hop_length = 512
bins_per_octave = 12

# Compute the constant-Q chromagram
CQT = librosa.amplitude_to_db(np.abs(librosa.cqt(y, sr=sr, hop_length=hop_length, bins_per_octave=bins_per_octave)), ref=np.max)

# Get the time and frequency bins
times = librosa.times_like(CQT)
frequencies = librosa.cqt_frequencies(n_bins=CQT.shape[0], fmin=librosa.note_to_hz('C1'), bins_per_octave=bins_per_octave)

# Convert the amplitude spectrogram to strength
strength = librosa.power_to_db(np.abs(librosa.stft(y, hop_length=hop_length)), ref=np.max)

# Calculate the average strength and frequency for each time frame
average_strength = np.mean(strength, axis=0)
average_frequency = np.sum(np.exp(CQT) * frequencies[:, None], axis=0) / np.sum(np.exp(CQT), axis=0)

# Calculate max and min values for average strength and frequency
max_avg_strength = np.max(average_strength)
min_avg_strength = np.min(average_strength)
max_avg_frequency = np.max(average_frequency)
min_avg_frequency = np.min(average_frequency)

# Write max and min values to a text file
output_file = 'audio_info.txt'
with open(output_file, 'w') as file:
    file.write(f"{max_avg_frequency:.2f}\n")
    file.write(f"{min_avg_frequency:.2f}\n")
    file.write(f"{max_avg_strength:.2f}\n")
    file.write(f"{min_avg_strength:.2f}\n")
    for i, time in enumerate(times):
        file.write(f"{time:.1f} {average_frequency[i]:.2f} {average_strength[i]:.2f}\n")

print(f"Average audio information with max and min values written to {output_file}")
```

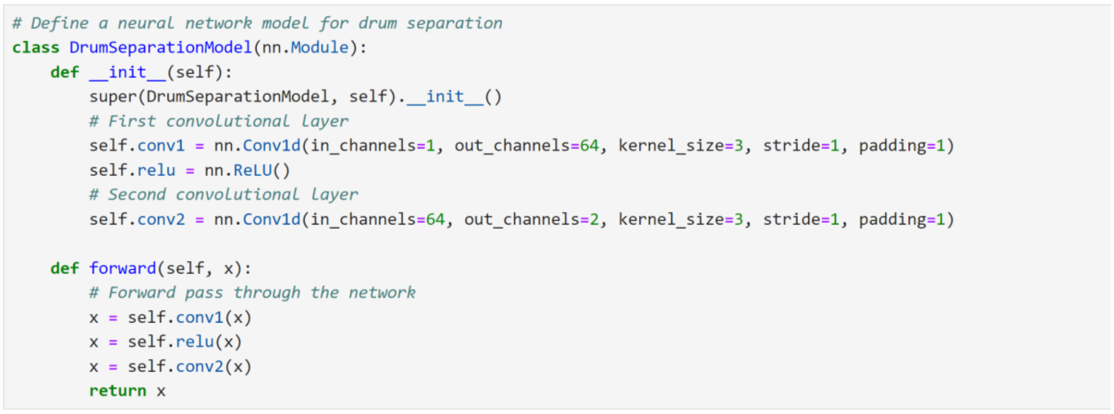
```
Average audio information with max and min values written to audio_info.txt
```

In the final part of this code, we output the melody text file. For every 0.1 second, we write a set of pair of frequency to magnitude. Also, we record the maximum and minimum frequency and magnitude for further analyzation.

b. Model training

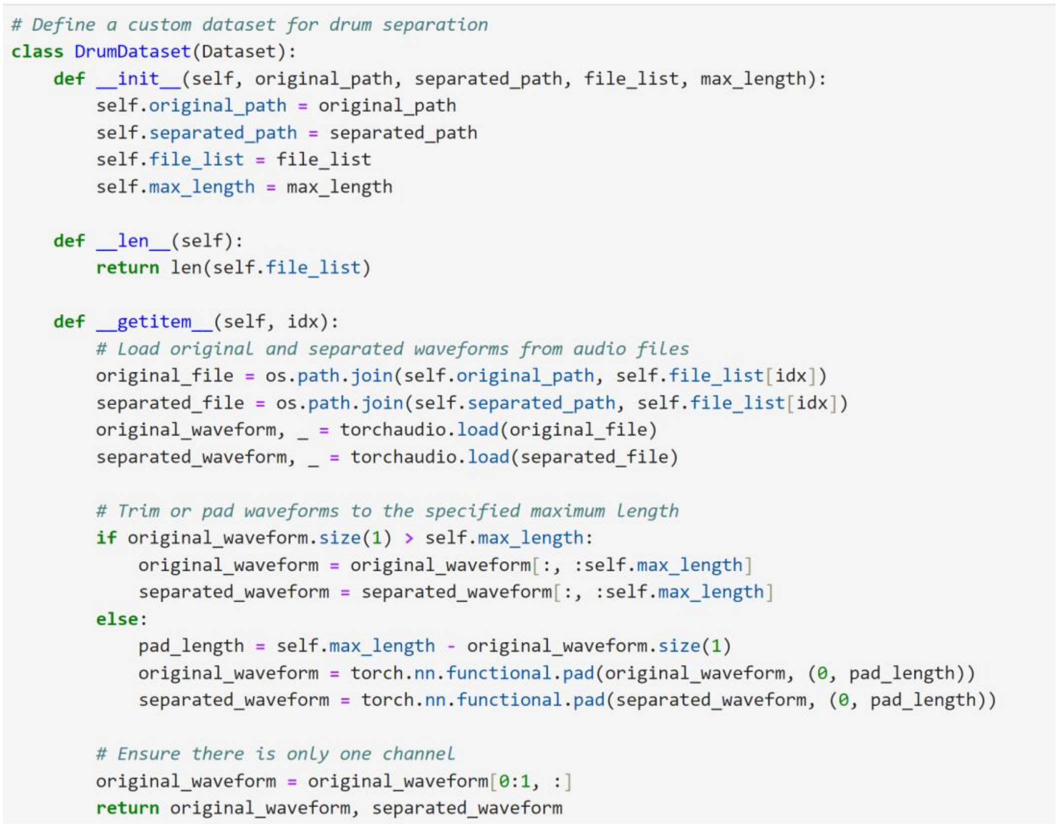


```python
# Import necessary libraries
import os
import torch
import torchaudio
from torch.utils.data import Dataset, DataLoader
from pathlib import Path
from sklearn.model_selection import train_test_split
import torch.nn as nn
import scipy.io.wavfile as wavwrite
import numpy as np
```



```python
# Define a neural network model for drum separation
class DrumSeparationModel(nn.Module):
    def __init__(self):
        super(DrumSeparationModel, self).__init__()
        # First convolutional layer
        self.conv1 = nn.Conv1d(in_channels=1, out_channels=64, kernel_size=3, stride=1, padding=1)
        self.relu = nn.ReLU()
        # Second convolutional layer
        self.conv2 = nn.Conv1d(in_channels=64, out_channels=2, kernel_size=3, stride=1, padding=1)

    def forward(self, x):
        # Forward pass through the network
        x = self.conv1(x)
        x = self.relu(x)
        x = self.conv2(x)
        return x
```

First, we trained our own model. It kicks off by importing all the necessary libraries for audio, PyTorch, and neural networks. Then, we create a simple neural network for drum separation using PyTorch's nn.Module class with just two convolutional layers.



```python
# Define a custom dataset for drum separation
class DrumDataset(Dataset):
    def __init__(self, original_path, separated_path, file_list, max_length):
        self.original_path = original_path
        self.separated_path = separated_path
        self.file_list = file_list
        self.max_length = max_length

    def __len__(self):
        return len(self.file_list)

    def __getitem__(self, idx):
        # Load original and separated waveforms from audio files
        original_file = os.path.join(self.original_path, self.file_list[idx])
        separated_file = os.path.join(self.separated_path, self.file_list[idx])
        original_waveform, _ = torchaudio.load(original_file)
        separated_waveform, _ = torchaudio.load(separated_file)

        # Trim or pad waveforms to the specified maximum length
        if original_waveform.size(1) > self.max_length:
            original_waveform = original_waveform[:, :self.max_length]
            separated_waveform = separated_waveform[:, :self.max_length]
        else:
            pad_length = self.max_length - original_waveform.size(1)
            original_waveform = torch.nn.functional.pad(original_waveform, (0, pad_length))
            separated_waveform = torch.nn.functional.pad(separated_waveform, (0, pad_length))

        # Ensure there is only one channel
        original_waveform = original_waveform[0:1, :]
        return original_waveform, separated_waveform
```

Next, we define a custom dataset class for drum separation, loading and handling the audio data.

```
# Calculate Root Mean Squared Error (RMSE)
def calculate_rmse(predictions, targets):
    rmse = torch.sqrt(torch.mean((predictions - targets)**2))
    return rmse.item()
```

```
# Set paths for training and validation datasets
train_original_path = "C:\\Users\\chaoy\\Downloads\\DSD100\\DSD100\\Mixtures\\Dev"
train_separated_path = "C:\\Users\\chaoy\\Downloads\\DSD100\\DSD100\\Sources\\Dev"
val_original_path = "C:\\Users\\chaoy\\Downloads\\DSD100\\DSD100\\Mixtures\\Test"
val_separated_path = "C:\\Users\\chaoy\\Downloads\\DSD100\\DSD100\\Sources\\Test"

# List all audio file names
train_files = list(Path(train_original_path).rglob('**/mixture.wav'))
val_files = list(Path(val_original_path).rglob('**/mixture.wav'))

# Assume file names have a one-to-one correspondence
assert len(train_files) == len(val_files)

# Split file names into training and validation sets
train_files, val_files = train_files, val_files
```

We also set up a function to calculate the Root Mean Squared Error between predicted and target waveforms.

Moving on, we set paths for training and validation datasets, list all the audio file names, split them into sets, and use DataLoader to load the datasets.

```
# Create instances of training and validation datasets
max_length = 1133393   # Set a maximum length
train_dataset = DrumDataset(train_original_path, train_separated_path, train_files, max_length)
val_dataset = DrumDataset(val_original_path, val_separated_path, val_files, max_length)

# Use DataLoader to load datasets
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)

# Initialize the drum separation model
model = DrumSeparationModel()

# Initialize loss function and optimizer
criterion = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

```
# Train the model
num_epochs = 3
for epoch in range(num_epochs):
    for original_waveform, separated_waveform in train_loader:
        # Forward pass
        predictions = model(original_waveform)
        # Compute loss
        loss = criterion(predictions, separated_waveform)
        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Then, we initialize the drum separation model, the Mean Squared Error loss function, and the Adam optimizer for training.

We jump into training the model using a loop over a specified number of epochs. It does the whole forward pass, loss calculation, backward pass, and optimization steps.

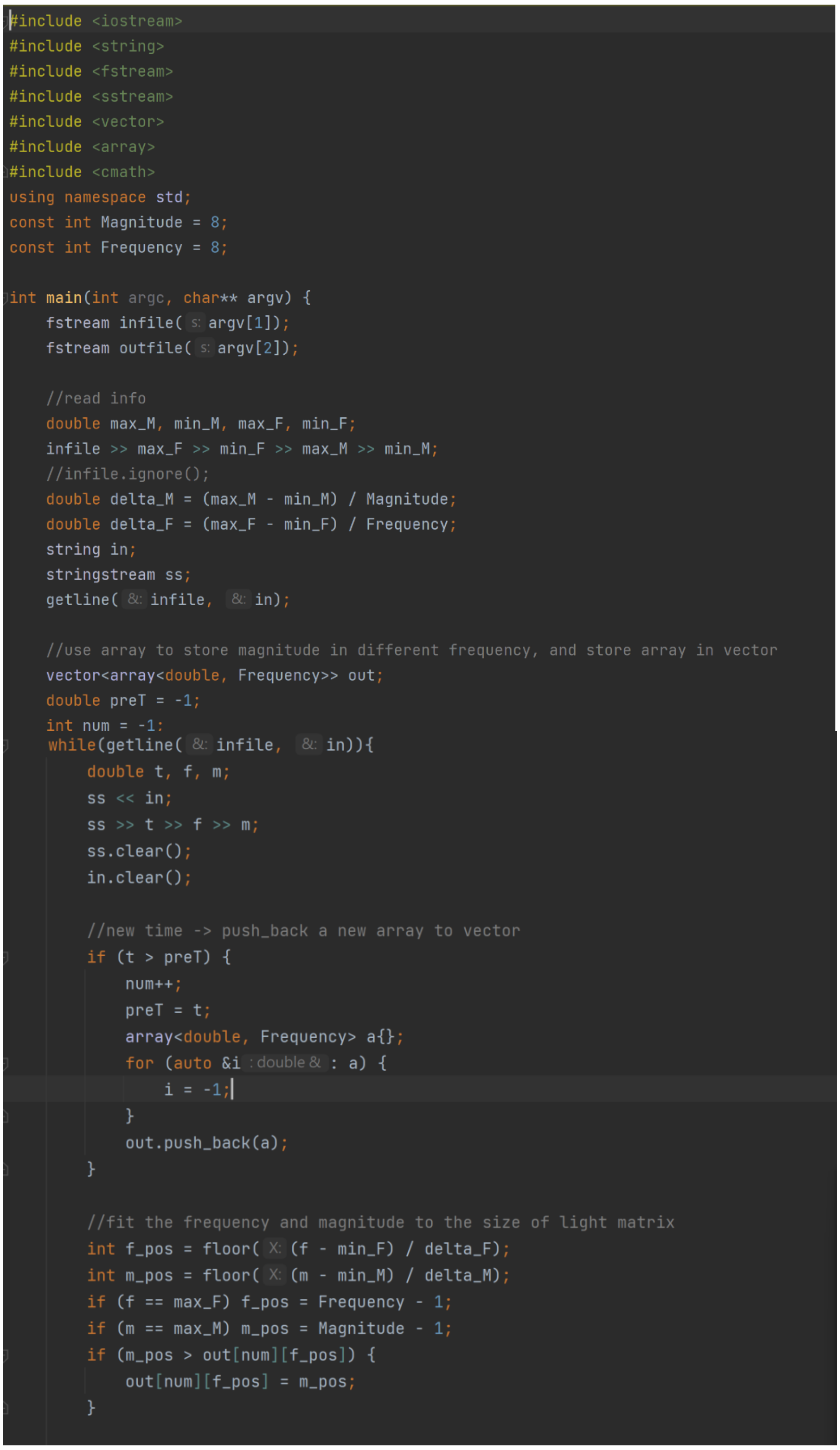
```python
# Evaluate the model on the validation set
model.eval()
with torch.no_grad():
    total_rmse = 0.0
    total_samples = 0
    for original_waveform, separated_waveform in val_loader:
        # Forward pass
        predictions = model(original_waveform)
         # Calculate RMSE
        rmse = calculate_rmse(predictions, separated_waveform)
        total_rmse += rmse
        total_samples += original_waveform.size(0)
    accuracy = total_rmse / total_samples
    print(f'Validation RMSE: {accuracy}')
```

```
Validation RMSE: 0.005262430310249328
```

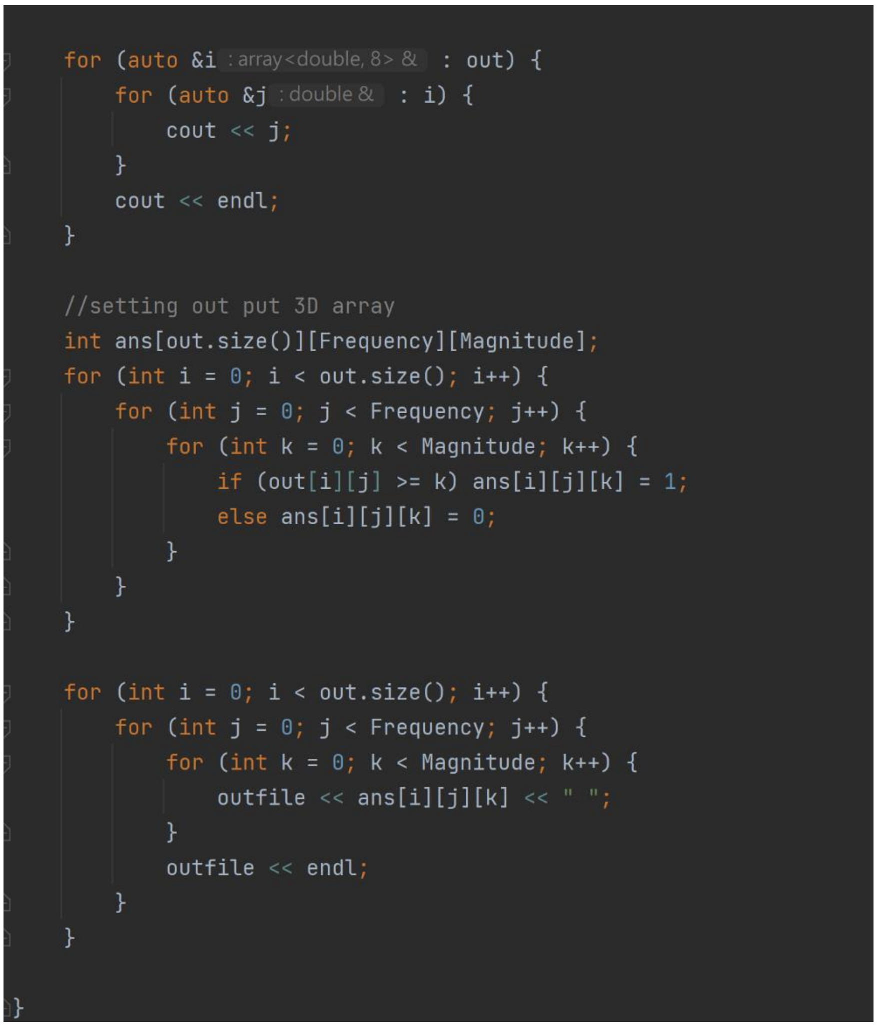
```python
# Load a new audio file for drum separation
new_audio_file = "drums3.wav"
new_waveform, _ = torchaudio.load(new_audio_file)

# Trim or pad the waveform to the specified maximum length
if new_waveform.size(1) > max_length:
    new_waveform = new_waveform[:, :max_length]
else:
    pad_length = max_length - new_waveform.size(1)
    new_waveform = torch.nn.functional.pad(new_waveform, (0, pad_length))

# Ensure there is only one channel
new_waveform = new_waveform[0:1, :]
new_waveform = new_waveform.unsqueeze(0)

# Make predictions using the trained model
model.eval()
with torch.no_grad():
    predicted_waveform = model(new_waveform)
    predicted_waveform = predicted_waveform.squeeze(0)

# Scale the floating-point tensor to the range of 16-bit integers
predicted_waveform_int = (predicted_waveform * 32767).to(torch.int16)

# Save the predicted audio as a WAV file
predicted_audio_file_wav = "drums4.wav"
wavwrite.write(predicted_audio_file_wav, 44100, predicted_waveform_int.numpy().T)
```

After that, we evaluate the model on the validation set, calculate the RMSE, and save the predicted drum audio as a wav file. But, the results weren't as awesome as we hoped. Two main reasons – not enough data and our computational resources reaching their limit. We used the DSD100 dataset with just 100 data points, split 50-50 for training and validation. Increasing the data might improve results, but our GPU and memory were maxed out, making further optimization a challenge.

c. Data pre-processing



```cpp
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
#include <vector>
#include <array>
#include <cmath>
using namespace std;
const int Magnitude = 8;
const int Frequency = 8;

int main(int argc, char** argv) {
    fstream infile(argv[1]);
    fstream outfile(argv[2]);

    //read info
    double max_M, min_M, max_F, min_F;
    infile >> max_F >> min_F >> max_M >> min_M;
    //infile.ignore();
    double delta_M = (max_M - min_M) / Magnitude;
    double delta_F = (max_F - min_F) / Frequency;
    string in;
    stringstream ss;
    getline(infile, in);

    //use array to store magnitude in different frequency, and store array in vector
    vector<array<double, Frequency>> out;
    double preT = -1;
    int num = -1;
    while(getline(infile, in)){
        double t, f, m;
        ss << in;
        ss >> t >> f >> m;
        ss.clear();
        in.clear();

        //new time -> push_back a new array to vector
        if (t > preT) {
            num++;
            preT = t;
            array<double, Frequency> a{};
            for (auto &i : a) {
                i = -1;
            }
            out.push_back(a);
        }

        //fit the frequency and magnitude to the size of light matrix
        int f_pos = floor((f - min_F) / delta_F);
        int m_pos = floor((m - min_M) / delta_M);
        if (f == max_F) f_pos = Frequency - 1;
        if (m == max_M) m_pos = Magnitude - 1;
        if (m_pos > out[num][f_pos]) {
            out[num][f_pos] = m_pos;
        }
```

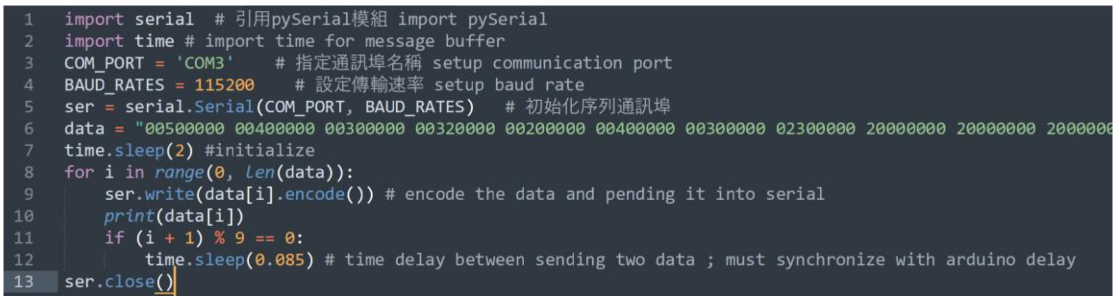


```
    for (auto &i : array<double, 8> & : out) {
        for (auto &j : double & : i) {
            cout << j;
        }
        cout << endl;
    }

    //setting out put 3D array
    int ans[out.size()][Frequency][Magnitude];
    for (int i = 0; i < out.size(); i++) {
        for (int j = 0; j < Frequency; j++) {
            for (int k = 0; k < Magnitude; k++) {
                if (out[i][j] >= k) ans[i][j][k] = 1;
                else ans[i][j][k] = 0;
            }
        }
    }

    for (int i = 0; i < out.size(); i++) {
        for (int j = 0; j < Frequency; j++) {
            for (int k = 0; k < Magnitude; k++) {
                outfile << ans[i][j][k] << " ";
            }
            outfile << endl;
        }
    }

}
```

Here we deal with the melody output file, we hope to visualize the melody into a spectrum which will change by time. Hence we choose to use a 2D LED light matrix to show the output. In this part pf code, we first fit the frequency and magnitude into the size of the light matrix we used, after the process, we output a file which contain a 3D array, i-axis for time, j-axis for frequency and k-axis for magnitude.

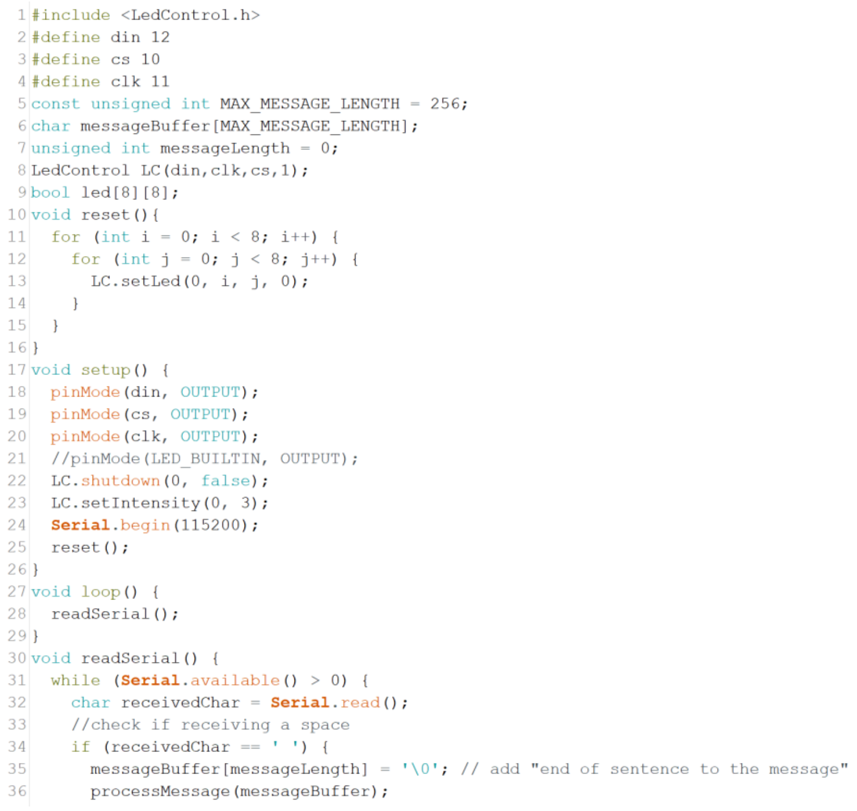
d. Result visualization

1. Light Matrix

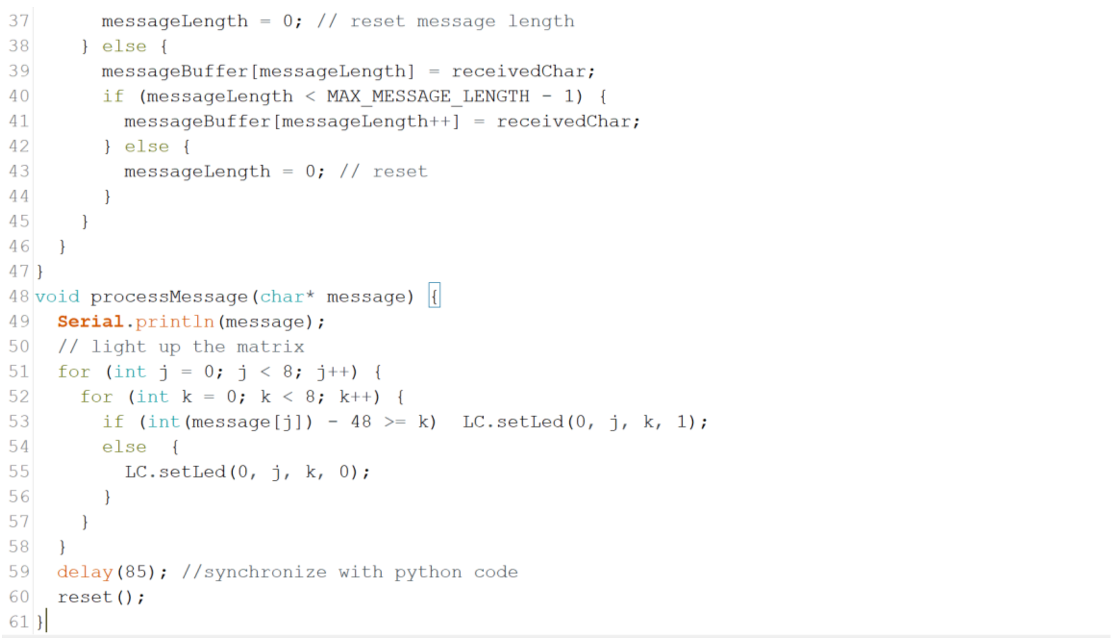


```python
import serial  # 引用pySerial模組 import pySerial
import time # import time for message buffer
COM_PORT = 'COM3'    # 指定通訊埠名稱 setup communication port
BAUD_RATES = 115200    # 設定傳輸速率 setup baud rate
ser = serial.Serial(COM_PORT, BAUD_RATES)   # 初始化序列通訊埠
data = "00500000 00400000 00300000 00320000 00200000 00400000 00300000 02300000 20000000 20000000 2000000
time.sleep(2) #initialize
for i in range(0, len(data)):
    ser.write(data[i].encode()) # encode the data and pending it into serial
    print(data[i])
    if (i + 1) % 9 == 0:
        time.sleep(0.085) # time delay between sending two data ; must synchronize with arduino delay
ser.close()
```

The format of the file for controlling the light matrix is hard for Arduino to read by serial.read(), because serial.read() can read at most 63 characters at a same time, we have at least 64 characters in our input file, though. Thus, we choose to use "pyserial" model, then connect to the Arduino UNO pad and input the required data. We directed copied the output and paste as a string. And use "serial.write" to write the data into Arduino.



```cpp
#include <LedControl.h>
#define din 12
#define cs 10
#define clk 11
const unsigned int MAX_MESSAGE_LENGTH = 256;
char messageBuffer[MAX_MESSAGE_LENGTH];
unsigned int messageLength = 0;
LedControl LC(din,clk,cs,1);
bool led[8][8];
void reset(){
  for (int i = 0; i < 8; i++) {
    for (int j = 0; j < 8; j++) {
      LC.setLed(0, i, j, 0);
    }
  }
}
void setup() {
  pinMode(din, OUTPUT);
  pinMode(cs, OUTPUT);
  pinMode(clk, OUTPUT);
  //pinMode(LED_BUILTIN, OUTPUT);
  LC.shutdown(0, false);
  LC.setIntensity(0, 3);
  Serial.begin(115200);
  reset();
}
void loop() {
  readSerial();
}
void readSerial() {
  while (Serial.available() > 0) {
    char receivedChar = Serial.read();
    //check if receiving a space
    if (receivedChar == ' ') {
      messageBuffer[messageLength] = '\0'; // add "end of sentence to the message"
      processMessage(messageBuffer);
```



```
      messageLength = 0; // reset message length
    } else {
      messageBuffer[messageLength] = receivedChar;
      if (messageLength < MAX_MESSAGE_LENGTH - 1) {
        messageBuffer[messageLength++] = receivedChar;
      } else {
        messageLength = 0; // reset
      }
    }
  }
}
void processMessage(char* message) {
  Serial.println(message);
  // light up the matrix
  for (int j = 0; j < 8; j++) {
    for (int k = 0; k < 8; k++) {
      if (int(message[j]) - 48 >= k)  LC.setLed(0, j, k, 1);
      else  {
        LC.setLed(0, j, k, 0);
      }
    }
  }
  delay(85); //synchronize with python code
  reset();
}
```

After we use “pyserial” to input data to the Arduino UNO pad, we keep reading the input until it reads a space, then we can use “settLED” function to switch each LED on or off. The LED light matrix will be keep updating until the data encounter the end of file.

It is worth-mentioned that the delay of the light matrix needs to be synchronized with the sleep time of “pyserial” input.

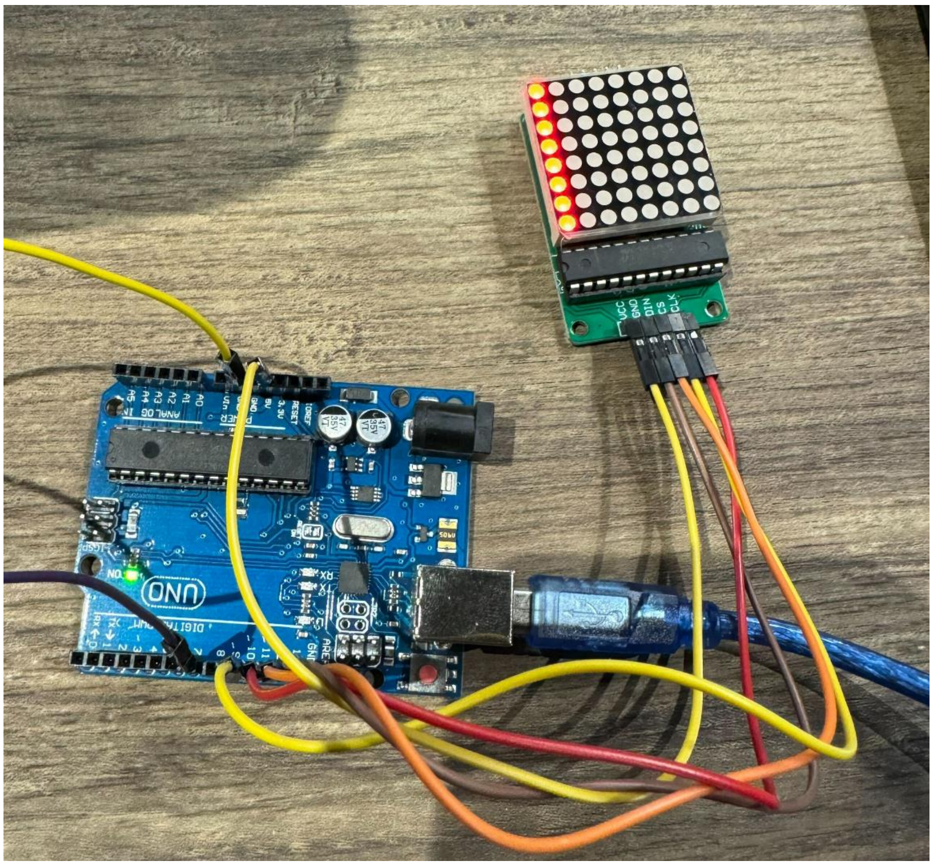
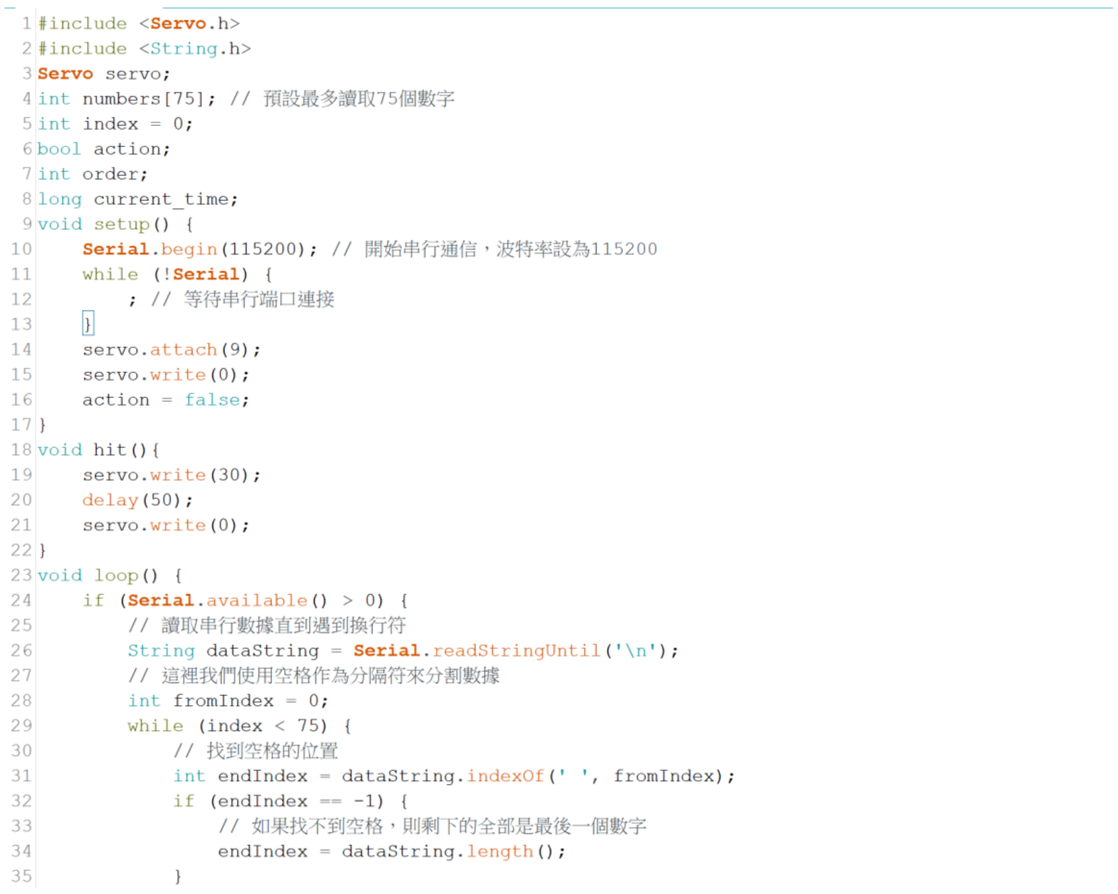
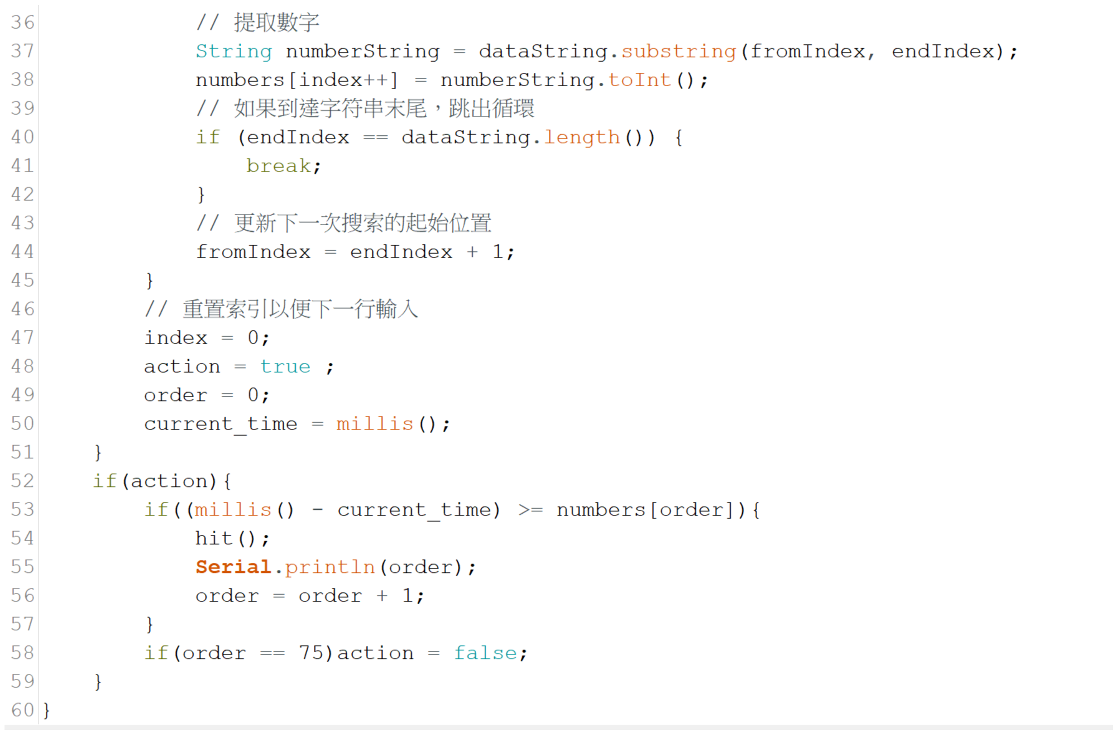


Figure 4: Light Matrix and Arduino UNO pad we used

2. Servo motor



```
#include <Servo.h>
#include <String.h>
Servo servo;
int numbers[75]; // 預設最多讀取75個數字
int index = 0;
bool action;
int order;
long current_time;
void setup() {
    Serial.begin(115200); // 開始串行通信，波特率設為115200
    while (!Serial) {
        ; // 等待串行端口連接
    }
    servo.attach(9);
    servo.write(0);
    action = false;
}
void hit(){
    servo.write(30);
    delay(50);
    servo.write(0);
}
void loop() {
    if (Serial.available() > 0) {
        // 讀取串行數據直到遇到換行符
        String dataString = Serial.readStringUntil('\n');
        // 這裡我們使用空格作為分隔符來分割數據
        int fromIndex = 0;
        while (index < 75) {
            // 找到空格的位置
            int endIndex = dataString.indexOf(' ', fromIndex);
            if (endIndex == -1) {
                // 如果找不到空格，則剩下的全部是最後一個數字
                endIndex = dataString.length();
            }
```



```
            // 提取數字
            String numberString = dataString.substring(fromIndex, endIndex);
            numbers[index++] = numberString.toInt();
            // 如果到達字符串末尾，跳出循環
            if (endIndex == dataString.length()) {
                break;
            }
            // 更新下一次搜索的起始位置
            fromIndex = endIndex + 1;
        }
        // 重置索引以便下一行輸入
        index = 0;
        action = true ;
        order = 0;
        current_time = millis();
    }
    if(action){
        if((millis() - current_time) >= numbers[order]){
            hit();
            Serial.println(order);
            order = order + 1;
        }
        if(order == 75)action = false;
    }
}
```

For the drum system, we use a text file containing a set of time in millisecond. If we process the data while reading the rest of the data, Arduino UNO pad will experience lagging and cause the beat to have error. We read all the data first and store in an array. Then we use a Arduino built-in model “millis()” to record system time, while the system time is at the correct moment of a drum beat, we let the servo motor move. The servo motor moves up and down for 30 degree, by doing this, we are able to control the motor to hit a surface and generate a hitting sound to simulate a single drum beat.

D. Possible Implementation

For further development, based on the “Demucs” library, we can develop a new audio analysis model for more instrument. Enhance the program for real-time audio source separation, allowing musicians to play along with live performances. We can also combine the light matrix and the input audio file directly, then we can generate a spectrum in real-time.

E. Conclusion

In conclusion, our project successfully addresses the growing demand for sheet music extraction from mixed audio files, specifically focusing on drumbeats. The combination of advanced techniques, such as Demucs, Wave-U-Net, bi-LSTM, and the Transformer model, enables accurate audio source separation.

The implementation involves not only the backend processing using Python libraries like Librosa and PyTorch but also a tangible output system with Arduino, including servo motors for drum beat simulation and a light matrix for visualizing the melody. This integration offers a comprehensive solution for musicians who seek both auditory and visual representations of their music.