Audio Splitter App

Split an input audio file into different channels of instruments to be used for backing tracks. Solves having to use bad quality YouTube videos as only source for backing.

Diagram

Description automatically generated

Plan to make it work by changing input audio to a set bitrate and passing through a specific instrument neural network as a spectrogram. Run through entire spectrogram column by column and using current, before and next spectrogram values as neural network input, providing which current spectrogram values are the instrument as the output.

Could be acquired by audio streaming service or I could run ads and initial payment for revenue.

Rough Plans for the Project

1. Create working audio to spectrogram (or useable input) algorithm prototype.
2. Implement working audio to spectrogram in lower-level language like C++.
3. Create working neural network output to audio algorithm prototype.
4. Implement working neural network output to audio algorithm in lower-level language like C++.
5. Find good neural network configuration and train with datasets to a low error.
6. Create front end user interface with audio input and audio player output.

If ‘Escape Velocity’ Revenue is Earned

1. Add optional online method to pass-through high-speed servers, quicker response.
2. Create pre-processed audio library from commonly used songs, quicker response (will involve data collection to see most popular songs – as I will not be able to do every song ever, only the most popular).

Spectrograms



A 3-dimensional graph used to represent audio. The x-axis is used to represent how far through the audio each column of the graph is. The y-axis is used to represent what frequency each row is. The brightness of each section represents the volume or amplitude of each section in dBFS (decibels full scale).

Neural Network Inputs



Use volumes of green row alongside volumes of previous x rows (blue) alongside volumes of next x rows (yellow).

Training Neural Network

Use gradient descent, calculating errors from sample data found here: <https://sigsep.github.io/>. Includes 9 sources to a total of 1733 samples which include: total track, bass, drums, other, and vocals.

Sub Tasks

* C++ implementation of audio files (preferably MP3) to spectrogram data, which can be saved to a spectrogram.
* Processing a full song’s spectrogram data to a specific track’s spectrogram data.
* Processing output spectrogram data to audio.
* Method of downloading audio into program by user.
* User interface.

WAV File Format Breakdown

<http://soundfile.sapp.org/doc/WaveFormat/>

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Size in Bytes** | **Example (hex)** |
| Chunk Identifier | “RIFF” | 4 | 52 49 46 46 “RIFF” |
| Chunk Size | Size of chunk after this | 4 |  |
| Format | “WAVE” | 4 | 57 41 56 45 “WAVE” |
| Sub Chunk 1 Identifier | “fmt “ | 4 | 66 6d 74 20 “fmt “ |
| Sub Chunk 1 Size | 16 | 4 | 10 00 00 00 |
| Audio Format | PCM = 1, any other values show compression | 2 | 01 00 |
| Channel Count | 1 – Mono 2 – Stereo | 2 | 02 00 |
| Sample Rate | Samples per second | 4 | 22 56 00 00 |
| Byte Rate | Sample rate \* channel count \* (bits per sample / 8) | 4 | 88 58 01 00 |
| Block Align | Bytes per sample for all channels | 2 | 04 00 |
| Bits Per Sample | Bits in a sample | 2 | 10 00 |
| Sub Chunk 2 Identifier | “data” | 4 | 64 61 74 61 “data” |
| Sub Chunk 2 Size | Samples \* channels \* (bits per sample / 8) | 4 | 00 08 00 00 |
| Data | The sound data | “Sub Chunk 2 Size” | Below |

Audio data samples – bits in order of their channels (left then right if stereo) representing amplitude at a certain time.

Text

Description automatically generated with medium confidence

APIs for file types stored at <https://github.com/mrbean26/audiosplitter>

MP3 File Format

<https://github.com/lieff/minimp3>

Using “minimp3.h” (base level) and “minimp3\_ex.h” (high level API) to read in MP3 files.

Text

Description automatically generated

Math

The time **t** at sample **s** can be written as .

The wavelengths between two amplitude peak times, **t0** and **t1** can be calculated using , thus making the frequency at time equal to or .

Discrete Fourier Transform

A mathematical function which turns a finite sequence of equally spaced samples (in this case amplitudes in an audio file) into a frequency domain (frequency and amplitude on the graph). This can be represented over time by doing individual functions over sections of samples.

This equation explains the volume ***v***for frequency ***f***. Variables are:

* Volume ***v*** for frequency ***f***
* Sample count ***N***
* Amplitude ***a*** (or sample) number ***s***
* Imaginary number ***j***
* Euler’s constant ***e***

The value of ***v(f)*** will return a value in the form of real and imaginary numbers. The result is the sum of the absolute values of these numbers.



Fast Fourier Transform – The Cooley Tukey Algorithm

Just a faster way of calculating the Discrete Fourier Transform.

And using:

So that a more efficient algorithm can be made like this…



Spectrogram Algorithm

1. Load samples from an audio file.
2. Split samples into chunks of equal sizes, sizes must be a power of 2.
3. Apply a window function to each chunk of audio samples – Hanning window is common.
4. Run fast Fourier transform on each chunk.
5. Apply the log mathematical function to each value in each chunk.
6. Downsize each chunk by taking an average per certain number of values.
7. For each average value, apply either the function **x6** or **1.5x**. The latter is most likely more efficient.
8. Find the maximum value across all chunks and divide each value by this to create a range from 0 to 1.
9. Multiply each value by 255 to receive a colour value and set each pixel to each value.



1.5x and x6 respectively

A picture containing dark, tiled

Description automatically generatedA picture containing text, white, tiled

Description automatically generated

FFTW Implementation

Using the DLL & LIB files found in the compressed files below, attach the LIB to the linker with project. Include the only header “fftw3.h” into the project and follow documentation (<http://www.fftw.org/index.html>) for usage. Examples with 64 sample chunks are shown, one dimensional and two dimensional respectively.



Text

Description automatically generated

Neural Network Output to Audio Algorithm

Inputs: *samplesPerChunk* (must be equal to that used in training and creating spectrogram data - *integer),* *fullTrackSpectrogram*(the spectrogram created with the full track of the song, using *samplesPerChunk* – **make sure this is not frequency downsized and scaled to the range 1 –** *vector<vector<float>>*), and *fullNeuralNetworkOutput* (with a frequency downsized track as input, using *samplesPerChunk* – *vector<vector<float>>*).

Outputs: *audioSamples* (*vector<integer>).*

1. Take the inputs into a function.
2. Take integer *frequencySampleCount* *=* *len(fullTrackSpectrogram[0]) / len(fullNeuralNetworkOutput[0]).*
3. Multiply each float in *fullNeuralNetworkOutput[i]* by each corresponding *frequencySampleCount* floats in *fullTrackSpectrogram[i]*, writing to *fullTrackSpectrogram[i].*
4. Run IFFT for each chunk in *fullTrackSpectrogram*.
5. Combine each chunk of *fullTrackSpectrogram* into a single *vector<float>* consecutively.
6. Return

Neural Network Structure Tests

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Activation** | **Hidden**  **Layer Count** | **Average Hidden Layer Size**  **(x \* inputSize)** | **Average Bias Count** | **Learning Rate** | **Momentum** | **Max error in final 10 items (1 epoch)** | **Time for 1 epoch (3800)** |
| reLu | 1 | 1.5 | 1 | 0.25 | 0 | 6.2 | 49 |
| sigmoid | 1 | 1.5 | 1 | 0.25 | 0 | 6.2 | 51 |
| tanh | 1 | 1.5 | 1 | 0.25 | 0 | 6.2 | 50 |
|  | | | | | | | |
| tanh | 0 | 1.5 | 1 | 0.25 | 0 | 3.2 | 4 |
| tanh | 1 | 1.5 | 1 | 0.25 | 0 | 6.2 | 50 |
| tanh | 2 | 1.5 | 1 | 0.25 | 0 | 6.7 | 93 |
|  | | | | | | | |
| tanh | 1 | 1 | 1 | 0.25 | 0 | 3.8 | 28 |
| tanh | 1 | 1.5 | 1 | 0.25 | 0 | 6.2 | 50 |
| tanh | 1 | 2 | 1 | 0.25 | 0 | 6.2 | 49 |
|  | | | | | | | |
| tanh | 1 | 1.5 | 0 | 0.25 | 0 | 4.5 | 38 |
| tanh | 1 | 1.5 | 1 | 0.25 | 0 | 6.2 | 50 |
| tanh | 1 | 1.5 | 2 | 0.25 | 0 | 5.1 | 39 |
|  | | | | | | | |
| tanh | 1 | 1.5 | 1 | 0.12 | 0 | 7.8 | 40 |
| tanh | 1 | 1.5 | 1 | 0.25 | 0 | 6.2 | 50 |
| tanh | 1 | 1.5 | 1 | 0.37 | 0 | 3.8 | 39 |
|  | | | | | | | |
| tanh | 1 | 1.5 | 1 | 0.25 | -0.2 | 7.8 | 39 |
| tanh | 1 | 1.5 | 1 | 0.25 | 0 | 6.2 | 50 |
| tanh | 1 | 1.5 | 1 | 0.25 | 0.2 | 7.8 | 40 |

From separate experiments, it seems that layer count is more significant than layer size in the reduction of errors.

Neural Network Timing Tests

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Layer count** | **Average Layer Size** | **Average Bias Count** | **Learning Rate** | **Momentum** | **Time for 100 epochs** |
| 5 | inputSize | 1 | 0.5 | 0.2 | 23.95 |
| 5 | inputSize \* 2 | 1 | 0.5 | 0.2 | 76 |
| 5 | inputSize \* 4 | 1 | 0.5 | 0.2 | 248.75 |
|  |  |  |  |  |  |
| 3 | inputSize \* 2 | 1 | 0.5 | 0.2 | 16.08 |
| 5 | inputSize \* 2 | 1 | 0.5 | 0.2 | 76 |
| 7 | inputSize \* 2 | 1 | 0.5 | 0.2 | 137.55 |
|  |  |  |  |  |  |
| 5 | inputSize \* 2 | 0 | 0.5 | 0.2 | 74.06 |
| 5 | inputSize \* 2 | 1 | 0.5 | 0.2 | 76 |
| 5 | inputSize \* 2 | 2 | 0.5 | 0.2 | 76.08 |
|  |  |  |  |  |  |
| 5 | inputSize \* 2 | 1 | 0.0 | 0.2 | 70 |
| 5 | inputSize \* 2 | 1 | 0.5 | 0.2 | 76 |
| 5 | inputSize \* 2 | 1 | 1 | 0.2 | 75.7 |
|  |  |  |  |  |  |
| 5 | inputSize \* 2 | 1 | 0.5 | 0.0 | 75.79 |
| 5 | inputSize \* 2 | 1 | 0.5 | 0.5 | 76 |
| 5 | inputSize \* 2 | 1 | 0.5 | 0.2 | 76.2 |

Legal

1. The most important thing to know is you can edit music if you do not intend to use it for commercial use.
2. Downloading videos from YouTube is in breach of YouTube's Terms of Service