**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 text, white, tiled

Description automatically generatedA picture containing dark, 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

**MIDI File Specification**

<http://www.somascape.org/midi/tech/mfile.html#:~:text=MIDI%20files%20are%20the%20standard,and%20System%20Exclusive%20messages.>

File is split into chunks, one after another. Chunk format is as follows:

* Chunk header (8 bytes) – 4-byte type sign: either “MThd” or “MTrk” showing header chunk or track chunk. And 4 Byte Chunk Length – size in bytes of the following data. This means that **total chunk size** = 8 chunk bytes + chunk length.

MThd Chunks (Header Chunks)

* Usually 6 Bytes Chunk Length, split into **3x 16-bit** integers.
* Integer one: **format**, describing how track chunks relate to one another. 0 = a single MIDI track chunk. 1 = two or more MIDI track chunks intended to be played simultaneously. 2 = two or more MIDI track chunks intended to be played independently.
* Integer two: **number of tracks** – no limit, apart from with format = 0, where track number must equal 1.
* Integer three: **tickdiv**, with **bit 15** (first bit of first byte) indicates timing format, with **bit 15 = 0** indicating metrical timing (pulses per quarter note, e.g. musical notation representation) and **bit 15 = 1** indicating timecode (frames per second).

MTrk Chunks (Track Chunks)

* **Variable length quantity** **“delta time”** denoting time since last event
* **Event (2+ bytes)** describing event in format MIDI, SysEx, or Meta.

*Delta Time Value in “MTrk Chunks”*

* Variable length using 1, 2, 3, or 4 bytes only.
* Determines number of **tickdiv intervals** (MThd) since last event, meaning that a delta time of 0 will execute simultaneously since last event.
* Variable length is determined with the first bit of each byte, with a **1** determining if another byte follows. This means the last byte of each delta time will have a 0 as the first bit. **Only the last seven bits of each byte are relevant to the delta time value.**

*MIDI Event in “MTrk Chunks”*

* Events involving audio
* First nibble of first byte indicates message type and second nibble of first byte indicates channel number
* Example message types include **note on, note off and pitch bend**

*SysEx Event in “MTrk Chunks”*

* System exclusive events
* Encoded as single, continuous and escape sequence types
* A single SysEx Event is encoded as follows (“F0”, **variable length “length”**, message)
* Escape sequences are encoded as follows (“F7”, **variable length “length”**, bytes), used for commands such as time and system common.

*Metadata Events in “MTrk Chunks”*

* Used for Non-MIDI events such as copyright
* Follows format (“FF”, type, **variable length “length”,** data)
* Type is an integer, such as “02” for copyright

**To-do Plans**

* **Whilst** doing, train neural network with different configurations **documenting** results.

1. Research several new and different neural net training methods
2. Implement researched training methods into library
3. Research MIDI file type
4. Create MIDI C++ saving library
5. Test with AI / Fixed Threshold / Proportional Threshold for filtering notes which will be used for MIDI

**Diagram

Description automatically generatedUI Screen Plan**

**Natural Selection Neural Network Whiteboard Pseudocode**

Text

Description automatically generated

**Alternative Self-Made MIDI Format**

* For each instrument, a **vector<vector<float>>** is used for the entire track. Each sub **vector<float>** is a frame, with the first value being used as a time delay interval until the next track, and other floats representing audible frequencies (ones that should be played).
* Can be saved to a file with the following format: ***piano\_index, piano\_data, guitar\_index, guitar\_data*** *and so on.*
* Good databases for sounds for playback can be found here: <https://ehomerecordingstudio.com/best-virtual-instruments/>
* However, many of these are not free so either record my own or record from other software.

**Filtering Notes from Network Output to be Used for MIDI (Options)**

* Threshold across entire track, finding only loudest frequencies
* Finding highest points in individual sections (such as words) to represent most audible frequencies **– most efficient / effective ratio**
* Neural network to determine if note should be used (1 or 0 output) according to surrounding note volumes

Finding highest points in individual sections

A picture containing chart

Description automatically generated

* Red line is representative of original frequency in spectrogram, other line represents functions applied to the frequency (therefore reference of **cos(x)** is irrelevant).
* **Algorithm** is as follows: temporarily normalise frequencies, apply mathematical function to make large values stay the same and small values get smaller, and then use a threshold to cut out quiet noise.

**Methods of Fast Convergence in Neural Networks (Backpropagation)**

Cyclical Learning Rate and Momentum

Includes specifying minimum and maximum for both learning rate and momentum and causing a **maxima** **for learning rate** in the middle of learning and **minima for momentum** in the middle of the learning, as the following graph represents.

Chart, line chart

Description automatically generated

Weight Decay

After each iteration, **“decay”** weights by multiplying each weight by a factor just less than 1. This prevents weights from growing too large, and for more complex neural networks, smaller decays should be used.

Random Datasets to Reduce Size

Consider **carefully** selecting fewer chunks from each spectrogram that are representative for the entire sample, to drastically reduce time due to less iterations. A random selection method processed each iteration could also be used.

Changing Target Values

Instead of choosing target values in the range 0 to 1 (the peaks of the activation function), consider choosing target values in the range 0.1 to 0.9. This avoids larger weights being adjusted which commonly makes the network unstable.

**Resilient Backpropagation (RPROP) Training Method**

Concept

RPROP does not use the magnitude of the calculated delta weight, only its sign (positive or negative). Alongside this, instead of requiring a hard-to-find learning rate, RPROP maintains separate weight deltas.

Diagram

Description automatically generatedAlgorithm

Side Notes

I am unsure if increase in delta weight should be done by a percentage multiplier or adding a constant value, so try with both. Perhaps try decreasing the multiplier / constant as epochs progress to create more gentle learning once nearest minima. **Easy to implement with an implementation of standard backpropagation.**

**Dropout**

Concept

Randomly dropping out of nodes during **training only** to prevent overfitting. Only parameter required is **probability of node dropping**.

Diagram

Description automatically generatedAlgorithm

Side Notes

Convergence can take twice as long when using dropout.

**Fonts**

Good library for free commercial use fonts found here <https://www.1001fonts.com/free-for-commercial-use-fonts.html?page=1>.

**Algorithm for Comparing Network Experimental Results**

Diagram

Description automatically generated

**Algorithm for Auto Training Several Configurations**

Text

Description automatically generated

**Training Results**

Layers 4-5, Biases 0-1, Epochs 1000 Best Network

Layers: 512, 128, 128, 128, 128, 64

Biases: 1

No Cyclical Learning Rate

Learning Rate = 0.4

Momentum = 0.05

Layers 6-7, Biases 0-1, Epochs 1000 Best Network

Layers: 512, 128, 128, 128, 128, 128, 128, 64

Biases: 1

No Cyclical Learning Rate

Learning Rate = 0.4

Momentum = 0.05

**Comparing Network Optimisation Algorithms**

|  |  |  |
| --- | --- | --- |
| **Name** | **Advantages** | **Disadvantages** |
| Gradient Descent | Minimal computation time, minimal memory usage. | Generally slow convergence time. |
| Newtons Method | Faster Convergence | High memory usage, high computation time due to exact hessian use. |
| Conjugate Gradient | Faster Convergence Than Gradient Descent, Avoids Hessian Calculation | High Memory Usage Due to Computation of Conjugate Parameter |
| Quasi-Newton Method | Approximates inverse hessian to save time, similar convergence speed to newtons method. | High memory usage due to storage of approximate hessian. |
| Levenberg-Marquardt | Very fast convergence normal computation time. | High memory usage due to storage of approximate hessian. |
| Adaptive Linear Momentum | Low memory usage, slightly faster than gradient descent. | Slightly higher computation time than gradient descent. |
| Adagrad | Avoids manually choosing learning parameters | Computationally expensive and slow convergence |
| Adadelta | Faster convergence than Adagrad | Computationally expensive |
| Root Mean Square Propagation | Faster convergence than Adagrad | Computationally expensive |
| Nesterov Accelerated Gradient | Faster convergence than gradient descent | Below average convergence speed |

**Adaptive Linear Momentum Algorithm is favoured by most websites**

**Alternative Algorithms for Network Training Explained**

<https://www.neuraldesigner.com/blog/5_algorithms_to_train_a_neural_network>

Levenberg-Marquardt Algorithm (Fastest Convergence, Highest Memory)

1. Set initial damping parameter = 0.001
2. Calculate error with current parameters
3. Setting Jacobian Matrix to be partial derivatives of a **single output network error (combine all into one)** with respect to each weight. Add to a matrix on one row, with many columns.
4. Set approximate hessian to , where **λ** is the damping parameter and **I** is the identity matrix.
5. Retrieve a delta of weights in order of their appearance in the Jacobian matrix with , where **e** is the final network error.
6. If error with new parameters increased, increase damping parameter (x10 usually), otherwise decrease damping parameter (x0.1 usually).

It is notable that the identity matrix **must** be added, so if the damping parameter becomes rounded to 0 from becoming too small, the algorithm will break.

Stochastic Gradient Descent (Very good!)

Selects a few random examples from dataset each epoch instead of the entire dataset. Saves time and the same (or better) convergence.

Adaptive Linear Momentum (“ADAM”)

A method that avoids choosing a constant learning rate.

Where variable **gt** is the derivative of a weight at a time, **vt**is the exponential average gradient, **st** is the exponential average square gradient and **B1** and **B2** are hyperparameters. **The vt and st** **used in weight updates are calculated to be (v / (1 – B1)) and (s / (1 – B2)) respectively.**

Common Values:

Root Mean Square Propagation (RMSProp)

Averages moving exponential gradients to produce more gradual and constant convergence, less likely to increase error back up and converges faster in general than gradient descent. The cost function **must** be always positive, such as a squared function.

Default values:

The algorithm is as follows:

1. Initiate **meanSquare** = **[0, …, 0]** and initiate **meanGradients = [0, …, 0]**
2. Iterate over batchSize of examples and average weight derivatives with respect to cost function. The meanSquare can be calculated procedurally with the formula:
3. Calculate new weights with the following formula below:

**‘K Nearest Neighbours’ in Audio Splitting**

Finds **K** nearest points classified with frequencies and takes the majority verdict. For example, if 3 out of 5 closest points are vocals, the data point is chosen as vocals. Counted as a supervised learning technique as dataset is labelled.

Requirements: low samples per chunk for accuracy (avoids having to include context), low frequency resolution for efficiency.

Dimensions of each datapoint will be equal to frequency resolution, and points will be labelled vocals or backing.

Nearest points can be found by calculating distance using the Euclidean distance formula:

Where **a** and **b** are data points and **i** is an integer index from 1 to dimension count.

**Concept Algorithm For Constant Training**

Text

Description automatically generated

**Step Function Possible for Backpropagation**

Gradients of original Heaviside step function are 0, so therefore cannot be used in backpropagation as the delta will result to be 0 due to the chain rule.

A high slope sigmoid function can be used to replace this, with the formula below, using **a** as a high value:

**Alternative Methods to Classify Audio Frequencies Avoiding Neural Network**

1. K Nearest Neighbours
2. Naive Bayes Classifier (Probability Approach)
3. Regression Methods
4. Decision Trees

**Improved Natural Selection Algorithm for Neural Networks**

1. Create initial population of Neural Networks, all with the same structure but all with different and random weight parameters.
2. Measure each neural networks fitness. For regression cases, using the mean squared error is good enough. However, in situations where an error is not clear, things like points collected in a game / time alive works well.
3. Reproduce a new population using ‘k’ parents. When k = 1, this is asexual reproduction and when k > 1, this is sexual reproduction. The probabilities of a parent being selected for reproduction can be found by performing the SoftMax function on each fitness score of the network.
4. Mutate each neural network by adding ‘Gaussian noise’ to each weight in the network.
5. Repeat from step 2.

**Batch Gradient Descent**

* Updates after iterating over entire dataset, instead of each data point.
* Less jumpy than stochastic descent, should descent gradually and consistently.
* Works by averaging out derivatives from each data point.

**Frontend Platform**

* Unity cannot work to load audio files from mobile device
* For iOS, use XCode and implement feature called ‘MusicKit’ that apple provides. This allows audio files to be loaded into runtime memory from Files, URLs, and the device Music Library. This requires XCode Version 12.
* For Android, use Android Studio as this allows audio files to be loaded from SD cards and URLs using the ‘MediaPlayer’ feature listed in documentation.

**Concept Theorised To Train Over All Songs in Dataset at Once without Stacks of Memory**

* Use a stochastic dataset with sections of each song
* Create a function that returns the random ‘all song dataset’
* Call at the start of each training epoch to get a new dataset each training iteration
* This function will return a small enough dataset each time which still includes a good enough section of each song.