UNIVERSITY OF CRAIOVA
FACULTY OF AUTOMATION, COMPUTERS AND ELECTRONICS

DEPARTMENT OF COMPUTERS AND INFORMATION TECHNOLOGY

DIPLOMA PROJECT

CRISTIAN BULEANDRĂ

SCIENTIFIC COORDINATOR

Prof. Univ. Dr. Ing. COSTIN BĂDICĂ

JULY 2016

CRAIOVA

UNIVERSITY OF CRAIOVA
FACULTY OF AUTOMATION, COMPUTERS AND ELECTRONICS

DEPARTMENT OF COMPUTERS AND INFORMATION TECHNOLOGY

Using Machine Learning for Automated Recognition of Sounds Generated by Musical Instruments

CRISTIAN BULEANDRĂ

SCIENTIFIC COORDINATOR

Prof. Univ. Dr. Ing. COSTIN BĂDICĂ

JULY 2016

CRAIOVA

“For the things we have to learn before we can do them, we learn by doing them.”

Aristotle, The Nicomachean Ethics

DECLARAȚIE DE ORIGINALITATE

Subsemnatul *CRISTIAN BULEANDRĂ*, student la specializarea *CALCULATOARE CU PREDARE ÎN LIMBA ENGLEZĂ* din cadrul Facultății de Automatică, Calculatoare și Electronică a Universității din Craiova, certific prin prezenta că am luat la cunoştinţă de cele prezentate mai jos şi că îmi asum, în acest context, originalitatea proiectului meu de licenţă:

- cu titlul „*USING MACHINE LEARNING FOR AUTOMATED RECOGNITION OF SOUNDS GENERATED BY MUSICAL INSTRUMENTS*"
- coordonată de *PROF. UNIV. DR. ING. COSTIN BĂDICĂ,*
- prezentată în sesiunea *IULIE 2016*

La elaborarea proiectului de licenţă, se consideră plagiat una dintre următoarele acţiuni:

- reproducerea exactă a cuvintelor unui alt autor, dintr-o altă lucrare, în limba română sau prin traducere dintr-o altă limbă, dacă se omit ghilimele şi referinţa precisă,
- redarea cu alte cuvinte, reformularea prin cuvinte proprii sau rezumarea ideilor din alte lucrări, dacă nu se indică sursa bibliografică,
- prezentarea unor date experimentale obţinute sau a unor aplicaţii realizate de alţi autori fără menţionarea corectă a acestor surse,
- însuşirea totală sau parţială a unei lucrări în care regulile de mai sus sunt respectate, dar care are alt autor.

Pentru evitarea acestor situaţii neplăcute se recomandă:

- plasarea între ghilimele a citatelor directe şi indicarea referinţei într-o listă corespunzătoare la sfărşitul lucrării,
- indicarea în text a reformulării unei idei, opinii sau teorii şi corespunzător în lista de referinţe a sursei originale de la care s-a făcut preluarea,
- precizarea sursei de la care s-au preluat date experimentale, descrieri tehnice, figuri, imagini, statistici, tabele et caetera,
- precizarea referinţelor poate fi omisă dacă se folosesc informaţii sau teorii arhicunoscute, a căror paternitate este unanim cunoscută și acceptată.

Data, Semnătura candidatului,

UNIVERSITY OF CRAIOVA
Faculty of Automation, Computers and Electronics

Department of Computers and Information Technology

Aprobat la data de
……
Şef de departament,
Prof. dr. ing.
Marius BREZOVAN/
Emil PETRE/
Dorian COJOCARU

PROIECTUL DE DIPLOMĂ

Numele și prenumele studentului/-ei:	BULEANDRĂ CRISTIAN
Enunțul temei:	USING MACHINE LEARNING FOR AUTOMATED RECOGNITION OF SOUNDS GENERATED BY MUSICAL INSTRUMENTS
Datele de pornire:	The starting point of this project was the idea to create an educational software that helps users improve their guitar playing skill by recognizing the notes being played and giving feedback on whether the right notes were played or not.
Conținutul proiectului:	The content of this project includes twelve main chapters that describe step-by-step how the final application was created and what libraries and techniques were used along the way. Emphasis was put on the ideas and problems encountered during each step, and their solutions, rather than on explaining the codebase and the reason why it was structured like this.
Material grafic obligatoriu:	Most graphic content presented in this paper is represented by screenshots of the application and diagrams that describe specific data structures or algorithms.
Consultații:	
Conducătorul științific (titlul, nume și prenume, semnătura):	PROF. UNIV. DR. ING. COSTIN BĂDICĂ
Data eliberării temei:	01.12.2015
Termenul estimat de predare a proiectului:	14.07.2016
Data predării proiectului de către student și semnătura acestuia:	

UNIVERSITY OF CRAIOVA
Faculty of Automation, Computers and Electronics

Department of Computers and Information Technology

REFERATUL CONDUCĂTORULUI ȘTIINȚIFIC

Numele și prenumele candidatului/-ei:
Specializarea:
Titlul proiectului:

Locația în care s-a realizat practica de documentare (se bifează una sau mai multe din opțiunile din dreapta):

- În facultate ☐
- În producție ☐
- În cercetare ☐
- Altă locație: [*se detaliează]*

În urma analizei lucrării candidatului au fost constatate următoarele:

Nivelul documentării		Insuficient ☐	Satisfăcător ☐	Bine ☐	Foarte bine ☐
Tipul proiectului		Cercetare ☐	Proiectare ☐	Realizare practică ☐	Altul [*se detaliază*]
Aparatul matematic utilizat		Simplu ☐	Mediu ☐	Complex ☐	Absent ☐
Utilitate		Contract de cercetare ☐	Cercetare internă ☐	Utilare ☐	Altul [*se detaliază*]
Redactarea lucrării		Insuficient ☐	Satisfăcător ☐	Bine ☐	Foarte bine ☐
Partea grafică, desene		Insuficientă ☐	Satisfăcătoare ☐	Bună ☐	Foarte bună ☐
Realizarea practică	Contribuția autorului	Insuficientă ☐	Satisfăcătoare ☐	Mare ☐	Foarte mare ☐
	Complexitatea temei	Simplă ☐	Medie ☐	Mare ☐	Complexă ☐
	Analiza cerințelor	Insuficient ☐	Satisfăcător ☐	Bine ☐	Foarte bine ☐
	Arhitectura	Simplă ☐	Medie ☐	Mare ☐	Complexă ☐
	Întocmirea specificațiilor funcționale	Insuficientă ☐	Satisfăcătoare ☐	Bună ☐	Foarte bună ☐

	Implementarea	Insuficientă ☐	Satisfăcătoare ☐	Bună ☐	Foarte bună ☐
	Testarea	Insuficientă ☐	Satisfăcătoare ☐	Bună ☐	Foarte bună ☐
	Funcționarea	Da ☐	Parțială ☐	Nu ☐	
Rezultate experimentale		Experiment propriu ☐		Preluare din bibliografie ☐	
Bibliografie		Cărți	Reviste	Articole	Referințe web
Comentarii și observații					

În concluzie, se propune:

ADMITEREA PROIECTULUI	RESPINGEREA PROIECTULUI
☐	☐

Data, Semnătura conducătorului științific,

ABSTRACT

Sound recognition is being used in a large variety of applications, mostly in the form of speech recognition, but other sounds can be classified too. This piece discusses the use of sound recognition for implementing an educational game that helps its users improve their guitar playing skills. A few chapters are dedicated to explaining how the computer ”hears” a sound, how it can be processed and how it can be used as input data for training a neural network to classify different sounds produced by the guitar. The steps that have been taken to achieve the final game demo are explained, in chronological order, and the problems encountered and solutions found during each of them is explained. This document does not contain an in-depth description of how a neural network works, but it shows how different network architectures were used for better solving a specific problem. This paper does not contain many explained code samples as it is more of a story of how a *game that recognizes guitar notes* was “born”.

Keywords: sound recognition, machine learning, neural network, guitar, sound processing, web application, educational game, data labeling, development process, mental process

TABLE OF CONTENTS

FIGURE LIST

1 INTRODUCTION

1.1 Project's goal

The project's goal was to create a sound recognition software that can be used in various applications that require the classification of sounds generated by musical instruments. The software should correctly classify the sounds given as input, even in noisy environments or when a low-quality microphone was used to record the sound samples. In order to demonstrate the usefulness of the recognition software a demo had to also be implemented. The demo is in the form of a video game, played using a guitar, where *notes* fall from the top and the user has to play the correct note in a limited amount of time. Because of the noisiness that had to be accounted for while classifying the sounds and difference in recording equipment quality and/or guitar types, machine learning looked like a valid option for the recognizer's core functionality.

While there are already several educational video games useful in learning to play the guitar (such as Ubisoft's RockSmith®, or Yousician) they are either not free, or require extra hardware for sound detection or the accuracy is not good enough so it happens pretty often for you to play the correct note, without it being recognized by the software. After publishing the training game created for this project, a new, free, high-accuracy training software will be available on different devices (PC, tablets, smartphones and even Smart TVs). In other words, the goal of this project also includes creating a usable educational video game from which others can benefit.

1.2 Motivation

There were several reasons for which I have chosen to develop this software, first of which is that there is no other software publicly available that offers the functionalities needed in order to recognize sounds contained within specific category (e.g. guitar sounds, types of alarms, specific sound pitches, etc.). The second reason was a personal one, more exactly because of my desire to learn more about machine learning and how automated learning can be used to classify sound. The project's development was aimed more towards recognizing the sounds of a guitar because the guitar is one of the most popular instruments around the globe and because it also an instrument that I know how to play.

2 CHAPTERS SUMMARY

The project's development included several separate steps that had to be taken in order to obtain the final deliverables. The different processes taken will be briefly described in this chapter and then, more detailed, in the following chapters.

2.1 The learning process

In this chapter I will be describe the resources and thought-process used while learning the basics of machine learning, artificial neural networks, digital sound representation and sound processing

2.2 The auxiliary software

In order to ease all the future developing steps I have decided to take some time to firstly create several tools that aid in the visualization of the recorded sound, automatically sound training samples recording and sound labeling. Libraries used are also enumerated in this chapter.

2.3 The neural network

The choice of machine learning technique and libraries used will be detailed in this chapter. The evolution of the network architecture will be explained step by step, motivating why each change was required.

2.4 Recording labeled training data

In this step it will be explained how the auxiliary tools previously created have been used in order to record the training data required and how training the neural network was done.

2.5 The game demo

In order to prove the usefulness of the sound recognizer and also demonstrate that it can be integrated in other applications a demo has been created. In this chapter the game rules, design, functionality and implementation will be detailed.

2.6 Putting it all together

While integrating the sound recognizer into the demo several problems were encountered. How they were resolved and how accurate the resulted demo recognizes the notes being played will be explained in this chapter.

2.7 Possible improvements

In this chapter several improvements that could increase the recognizing accuracy will be described, such as using multiple neural networks together (boosting) or grouping samples based on the type of the guitar or microphone that was used for recording.

2.8 Software application structure

The project's folder structure and what each file was used for is described in this section.

2.9 The Development environment

All the integrated development environments (IDEs) and hardware used while developing this project will be listed in this section. Why and how a certain software or hardware was used will also be detailed.

3 THE LEARNING PROCESS

3.1 Digital sound representation

Before diving into machine-learning techniques it is first required to have a good understanding of what sound is and how it is represented, recorded and stored inside a digital computer. First of all, the sound in the "real" world, is a continuous vibration in the form of waves. Being continuous it means that at any given time it has a specific value, given by the amplitude of the waveform at that point in time. In order to be able to store and process sound data using a computer, the **analog sound** has to be converted into a **digital sound** using the processes called **discretization** and quantization.

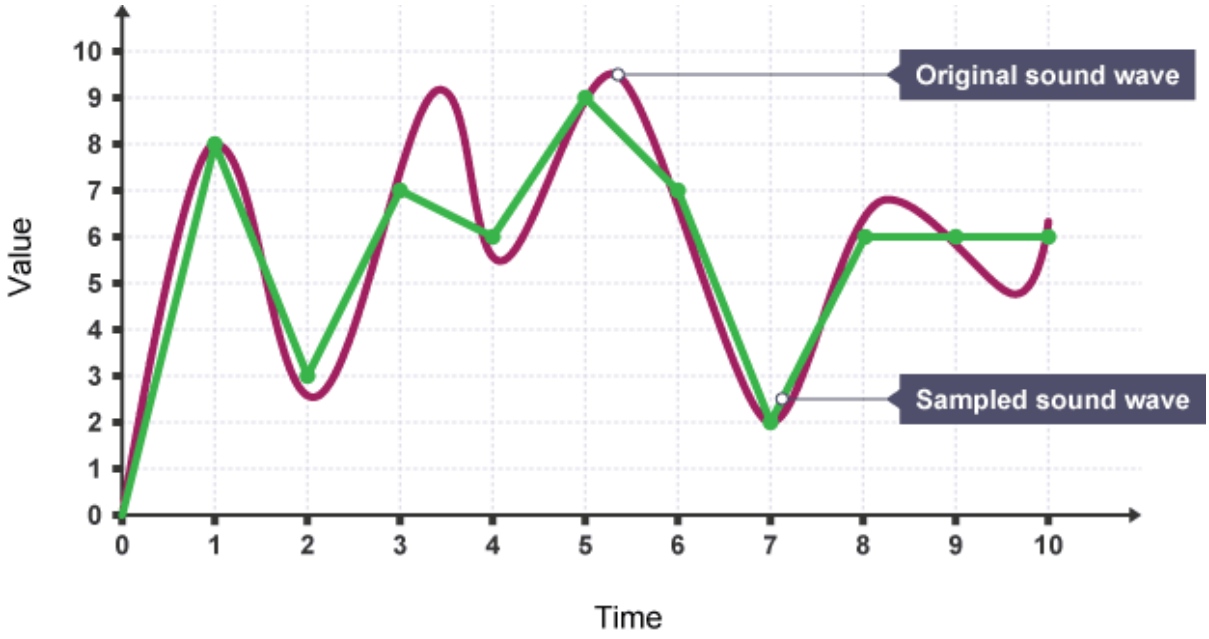


Figure 1: Analog vs sampled sound wave. (source: [BBCEdu])

From the Wikipedia definitions: "In mathematics, discretization concerns the process of transferring continuous functions, models, and equations into discrete counterparts. This process is usually carried out as a first step toward making them suitable for numerical evaluation and implementation on digital computers.". So, discretization can be obtained by *sampling* the analog sound at a fixed sampling rate, which, in the case of the microphone software used for this project, was set to **44.8kHz**, i.e. 44800 samples per second which approximatively equals to one sample every 0.02 milliseconds.

The most important thing to remember is that when input data is received from the microphone it can actually be represented as an *Array* of floats with values in the range [-1, 1]. If the values are plotted, then a graph showing a wave-form representation of the sound can be generated:

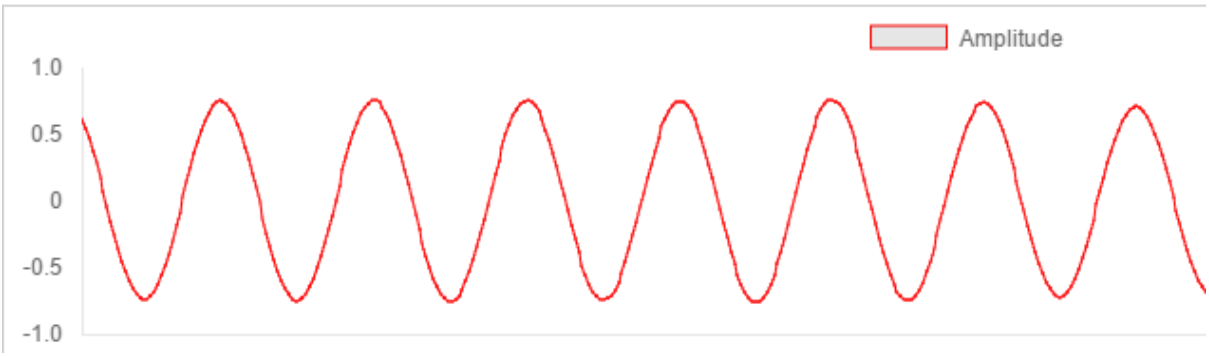


Figure 2: Graph showing sampled data while a whistling sound is recorded. Adjacent points are connected by lines.

3.2 Digital sound processing

While getting the sound from the microphone is easily done using the APIs implemented by the web browsers, making use of it is a bit trickier. Using the values given directly by the waveform data cannot lead to accurate results as the microphone input is very noisy and can be affected by many factors.

3.2.1 Noise reduction

Reducing the noise is a good way to improve the data relevance and it can be easily done by applying some of the filters already implemented in the browser audio filters. One such filter, and the filter that was used for recording the training data required for this project, is the *biquadFilter* set to behave like a band pass filter. A band pass filter means that only sounds that are not in the desired frequency range are muted or have their volume reduced.

The following figure graphically shows a band-pass filter of width ***B***, centered at frequency *f0:*

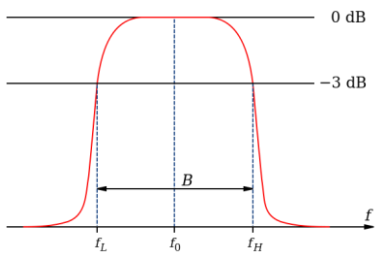


Figure 3: Band pass filter (source: [WikBaPa])

3.2.2 Fourier analysis

Having the sound represented in the time domain is helpful for understanding how our signal looks like but not that useful when trying to understanding what frequencies are most important for

recognizing a specific sound. Using *Fourier analysis,* we can convert a signal from its time domain representation to a frequency domain one, which allows to quantify the frequency intervals of the input sound, recorded at the default rate of 44800 samples per second. The *fftSize* of the *Analyser* Web Audio API node was set to 1024; in other words, using *fast Fourier transform* the input data is processed in chunks of 1024 values, this means that at any point in time the FFT data stored corresponds to the latest 21ms (1024/44800 s) of the sound recorded. Because smoothing is also enabled, the sound recorded before this interval also affects the output FFT data to some degree. The method that is used by the Web Audio API to compute the *Fourier transform* is presented below:

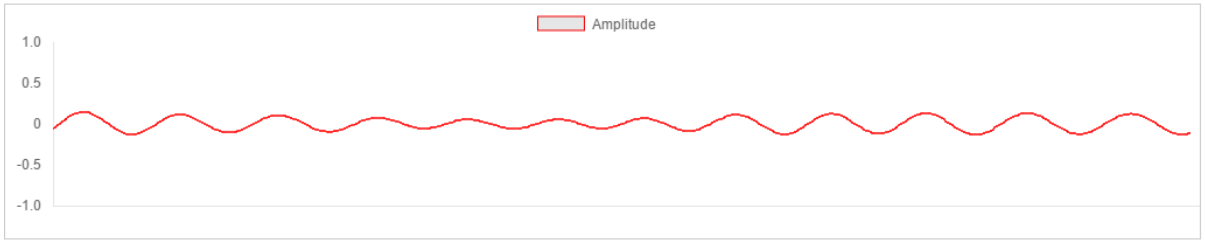
"When the current frequency data are computed, the following operations are to be performed:

1. **Down-mix** all channels of the time domain input data to mono assuming a channelCount of 1, channelCountMode of "max" and channelInterpetation of "speakers". This is independent of the settings for the AnalyserNode itself.
2. **Apply a Blackman window** to the time domain input data
3. **Apply a Fourier transform** to the windowed time domain input data to get imaginary and real frequency data
4. **Smooth over time** the frequency domain data
5. **Conversion to dB**." (source [WebAud] #widl-AnalyserNode-fftSize)

The *getByteFrequencyData* of the *AnalyserNode* returns an array of only 512 values (as the FFT data array is symmetric, only half of the 1024 values are relevant), where each value X of the array represents how similar is the input sound (latest, smoothed, ~21ms recorded data) with a sinusoidal wave frequency $F_{x.}$

The *fast Fourier transform* implementation of the *AnalyserNode* and, is computed continuously, in an efficient way, so the discrete Fourier transform representation of the input sound can be accessed multiple times a second with no performance issues. This proved to be useful, as in the demo game presented later in this paper the FFT data is requested at a rate of over 30 times per second.

Waveform:



FFT Bar chart:

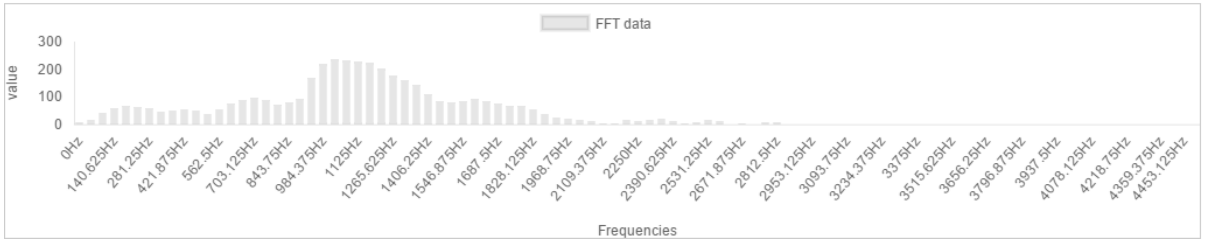


Figure 4: FFT transform bar chart of the waveform in red.

3.3 Machine learning

After learning about the sound representation and what data can be obtained from a microphone the next step was to learn more about machine learning and what type of machine learning technique would have good results in sound recognition. Machine learning algorithms can be split into two different categories: supervised and unsupervised. The main difference between those two is that supervised learning requires the output dataset to be provided when training the network (e.g. the data is labeled) whereas in unsupervised learning the input data is enough to train the network. Because of this, supervised learning is usually used for classifying the input data and unsupervised learning is used for the clustering the input data (group the input data based such that inputs that are similar to each other are most likely to be part of the same group). As the number of classes (the guitar sounds played) to be recognized is pretty large and the inputs for different classes are, in some cases, very similar, supervised learning was used in this project.

3.3.1 Artificial Neural Networks

While the name may sound complicated, artificial neural networks are really easy to implement and understand. They are said to mimic to a certain degree the activity of the brain because their components are *neurons* with *synapses* between them, similar to the brain. Their digital representation can be expressed in the form of a graph, where each node represents a neuron and each edge between two nodes a synapse. Most simple and popular neural network architecture is the perceptron, in which neurons are grouped in different layers and the neurons in each layer only has direct connections with the ones in the next layer.

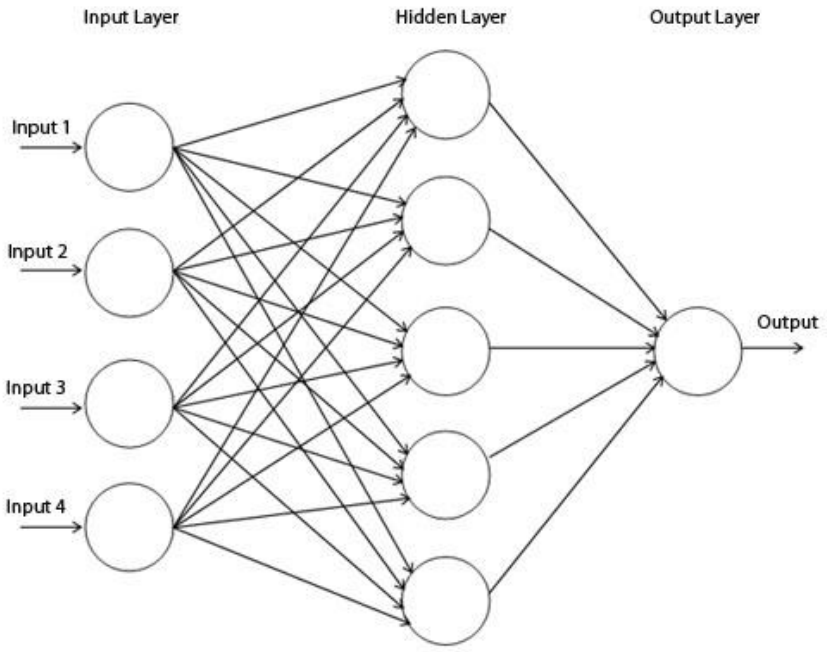


Figure 5: Example of a perceptron with a single hidden layer and one output neuron. (source: [GitSyn])

For this project the input is sound, more exactly the normalized (values are mapped to the [0, 1] interval) fast Fourier transform data of the sound, and the output will be the class of the sound. For guitar I have chosen to have 6 * 13 = 48 classes to be recognized; here 6 is the number of strings the guitar has and 13 is given by the number of distinct notes to recognize for each string, where 13 comes from the 11 fretted notes classified (when the finger is placed on one of the frets numbered from one to eleven), one class for the open string (when the string is played openly, with no fingers placed on the fretboard) and one class for the muted string (when the string is not played at all). Only the first eleven fretted notes are recognized for reducing the complexity of the project and also because starting with the twelfth fret the next octave starts for each of the strings (first string 12^{th} fret is the same note as first string open, but one octave higher).

The following screenshot shows the different note names for the first twelve frets of a standard-tuned (EADGBE) guitar. Note that instead of recognizing the note names I have decided to recognize the exact string and string fret position (this is because, as it can be seen in the screenshot below, the same note name appears on different string; even though the note is the same, the soundwave form differs as the strings have different materials and thickness) in order to better recognize where the player actually had his fingers when the sound was played.

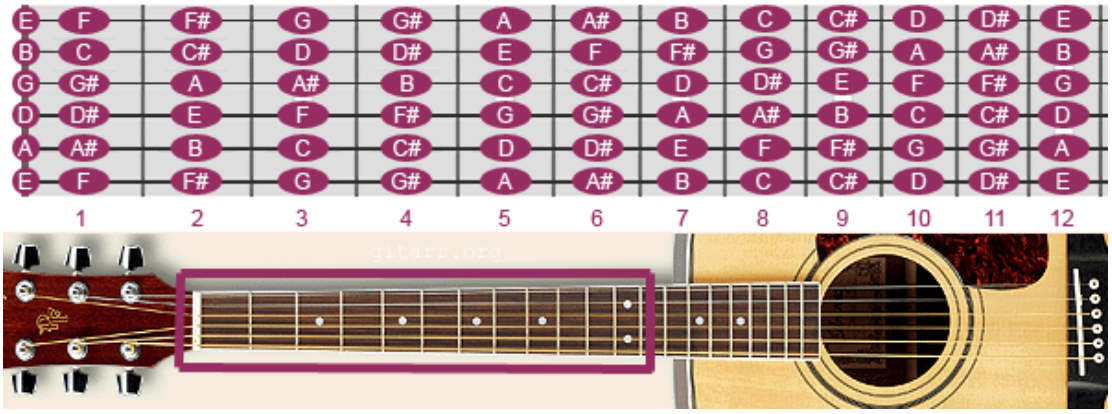


Figure 6: Note names on the guitar fretboard (source: http://www.guitar-chord.org/images/fretboard.png)

3.3.2 Training process

In order for the neural network to be useful it has to be trained with labeled data, also known as classified examples. Labeled data consists of the input data and the label (class), where input data in this project is the sound and the label is the guitar string and fret being played. For training the network so that it recognizes sounds labeled data has to be acquired and fed into to the network, which will "learn" to correctly classify the data using backpropagation. The learning process is usually automated using backpropagation, a training method that, given the current network structure (weights and biases), tries to minimize the error of a given cost function, usually using the gradient descent method.

After each iteration of the backpropagation algorithm the network's weights and biases are automatically updated. This process is repeated until the error on the given test cases is under a specified value. The error, or how well the network performs on the test cases, is computed by activating the network on input data for which the output is known and seeing how big is the difference between the real, known, output and the output generated by the network. As soon as the outputs are similar enough the training can be stopped. There's also the case when the generated output never reaches the desired accuracy and, in order to avoid an infinite loop, the training process is also stopped if the desired accuracy is not achieved after a given number of maximum iterations.

More details about how the labeled training data was recorded are shown in chapter 6.

4 THE AUXILIARY SOFTWARE

Once the algorithmic foundation of this project has been thoroughly researched it was time to create auxiliary tools that, after being completed, will drastically reduce the time it takes to do repetitive

actions or to visualize data. Several open-source libraries and browser build-in APIs have been used to develop those auxiliary tools.

4.1 Libraries and APIs used

Although libraries have been used for both the recognition software and the game demo, in this section only libraries related to the recognition software will be enumerated.

4.1.1 Chart.js

The first library usage that the user encounters is the one of *Chart.js*, a JavaScript library used to display graphical charts of the sound waveform data and fast Fourier transform bar charts.

For each of the three charts a few optional settings had to be passed to the Chart.js library in order for them to be correctly displayed:

- For the *waveform* chart the minimum and maximum values displayed have been to **-1** and **1**. The fill was also removed and plot point radius set to zero so that the chart looks like simple 1px line, with the equation *y=0* when no sound is being recorded.

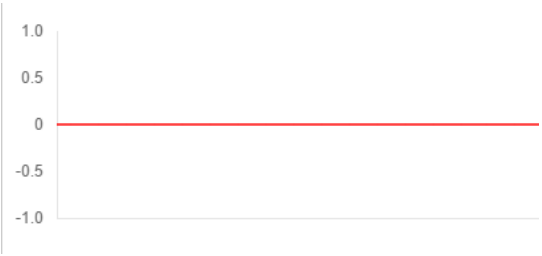


Figure 7: The waveform chart when no sound is being played looks like a line with y=0.

- For the fast Fourier transform chart the type was set to *bar* and minimum and maximum values set to 0 and 256 respectively. Those values were set because the FFT data is populated from the Web Audio API using the *getByteFrequencyData* method of the AnalyserNode that sets the data in *Uint8Array*, array that can only store 1-byte unsigned integers, thus only values between 0 and 256. Another change that had to be done for the chart to provide meaningful data was to compute and display labels that correspond to the frequencies of the displayed bar charts.

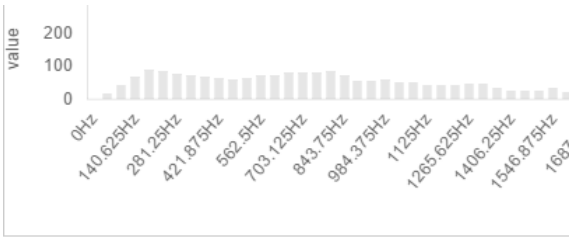


Figure 8: Labels showing the frequency in Hz on the X axis and the value [0-256] on the Y axis.

- For the final chart, the one that displays the normalized FFT data, the output interval has been set to [0,1] (because values are normalized between minimum and maximum of the initial FFT data). The rest of the settings are the same as for the initial FFT bar chart.

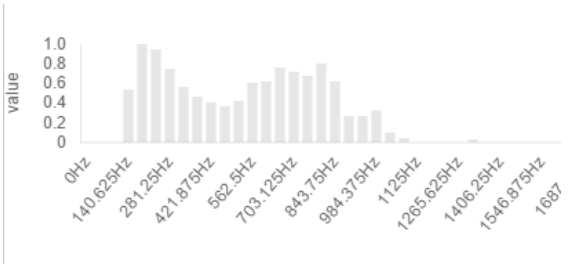


Figure 9: Same data as in the figure above but normalized between minimum and maximum values.

4.1.2 jQuery

jQuery, one of the most popular JavaScript libraries, has been used in this project for both ensuring cross-browser compatibility and for keeping the code cleaner when adding event listeners or when selecting and altering DOM elements.

4.1.3 Synaptic.js

Because this project involved the creation of a usable neural network (not only finding an architecture that works in theory, but also implementing a network which outputs relevant data when activated on the input dataset) I have decided to use an open-source library instead of writing all the code to create, train, test and a neural network. From their GitHub ReadMe description: "Synaptic is a JavaScript neural network library for node.js and the browser, its generalized algorithm is architecture-free, so you can build and train basically any type of […] neural network architectures."

While this library made it trivial to create networks for multi-class recognition and to train them, it had a few issues that were slowing the development of this project. One of the biggest issues was training performance. Synaptic.js has two training functions, one that trains the network in the main browser thread, so locking the entire page while the training phase is being run and another one that

trains the network in a web worker thread. After a few trials it turned out that using the worker train function was slower than the main-thread training function which was a bit odd, as the worker thread should be faster than the main thread as there is no UI or DOM to worry about. After debugging the code, I have noticed that the web training method was being executing like so:

1. Copy the activation and propagate functions to the worker source code.
2. Initialize the worker (create the new thread from the computed source code).
3. For each training sample, get the data from the main thread, send it to the worker thread.
4. Also, for each training sample send the current network memory (weights and biases).
5. The worker thread handles the activation and back-propagation for a single training sample and returns the updated network weights.
6. The main thread logs info messages and decides whether the training has completed, i.e. the activation error on the test dataset (the difference between the output generated by the network and the desired output) is lower than specified error.

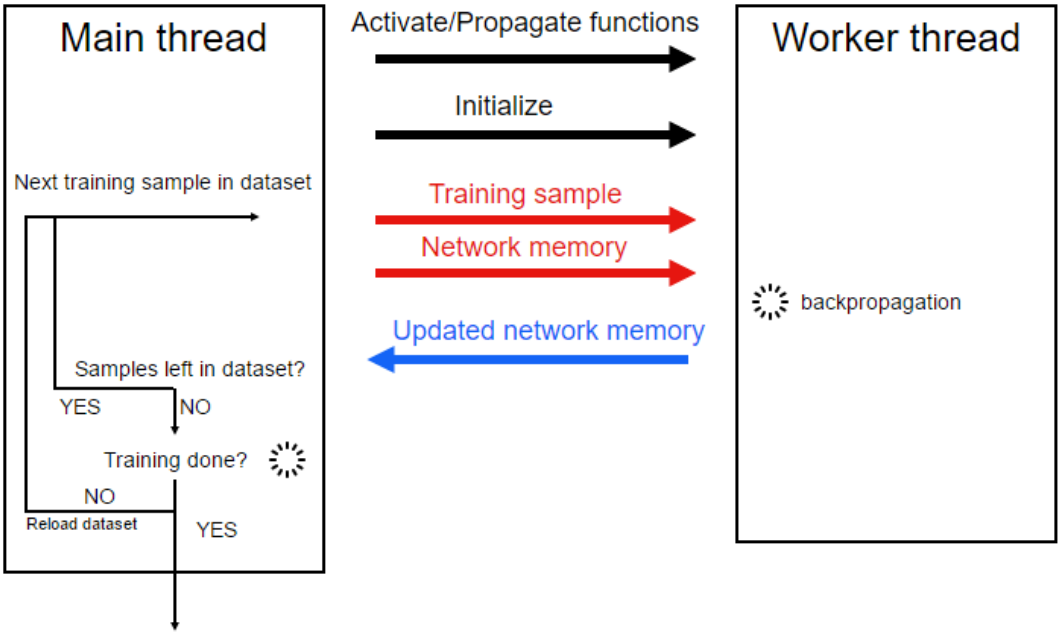


Figure 10: Diagram showing the previous implementation of worker training.

While the steps above perform the computations on the worker thread, the overhead of sending back and forth the training samples (input/output data) and the network memory (weights and biases) is very high. I noticed, using the CPU profiler provided by Chrome developer tools, that, because of the

overhead previously described, the worker thread was over 90% of the time idle. This means that instead of using the CPU power to train the network, the processor's clocks were spent transferring data between the threads. I have managed to fix this issue (and submit a pull-request that was later accepted by the library maintainer) by replacing the above steps with the following ones:

1. Copy the current network memory (weights and biases) to the worker source code.
2. Copy the activation and propagate functions to the worker source code.
3. Copy the entire training dataset to the worker source code.
4. Initialize the worker.
5. Train the network on the worker thread only, with no intervention from the main thread.
6. Send log messages from the worker thread to the main thread so that training progress can be displayed.
7. The worker thread decides when training has been completed and after that sends the new, updated, network memory to the main thread.

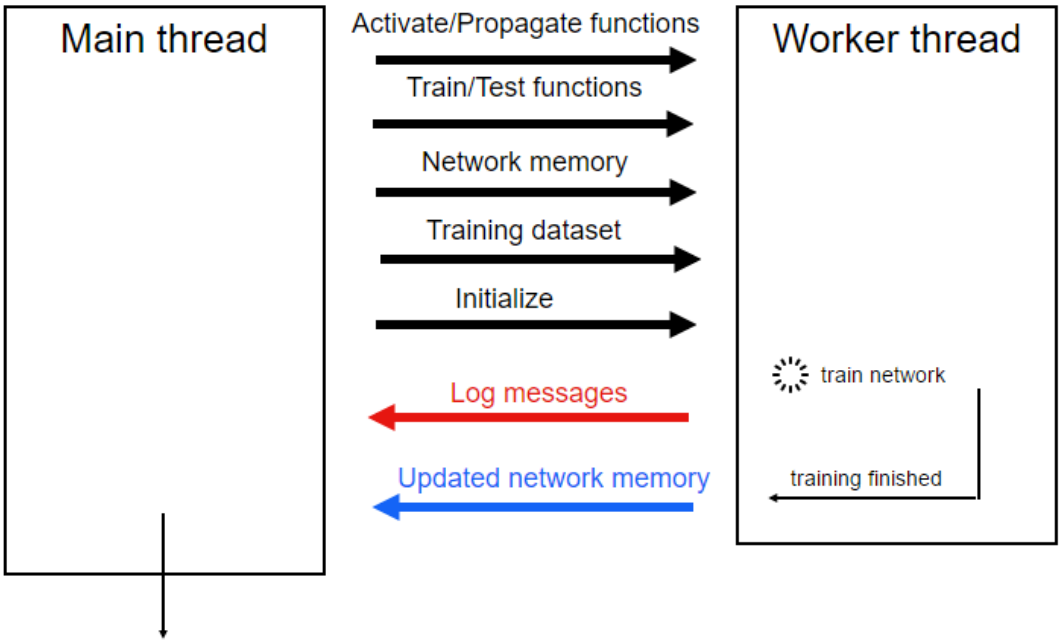


Figure 11: Diagram showing the improved worker training functionality.

Using this approach, the memory transfer overhead was eliminated (even though now a bit more data is being transferred in the worker initializing phase) and the worker idle time was reduced to 0%. This means that, for the background thread, the CPU usage is 100% while the network is being trained.

This has substantially reduced the time it takes to train the network from over 5 minutes (per approximately 200 training samples) to well under 15 seconds. This allowed for easier trial and error network architecture testing and ultimately lead to faster finding the best network architecture suited for classifying guitar sounds.

4.2 Visualizing the data

Graphically visualizing the sound data helped with deciding what filter to use and what parts of the input data to be sent to the neural network for training, as it proved that most of the recorded data was not relevant from the training point of view (i.e. only a small interval of frequencies were important for sound recognition, the rest of frequencies being either non-existent or they were not improving the training at all).

Apart from the charts presented in the Chart.js section, there was also implemented a bar chart for displaying similar training data (samples with similar labels) in order to see if the variance between the input data is large enough or how different samples within the same class are.

Here is a screenshot of the **dataviewer** showing the first 4 frequency bins (out of the total of 512 bins outputted by *fast Fourier transform* algorithm used) of the samples recorded for all strings:

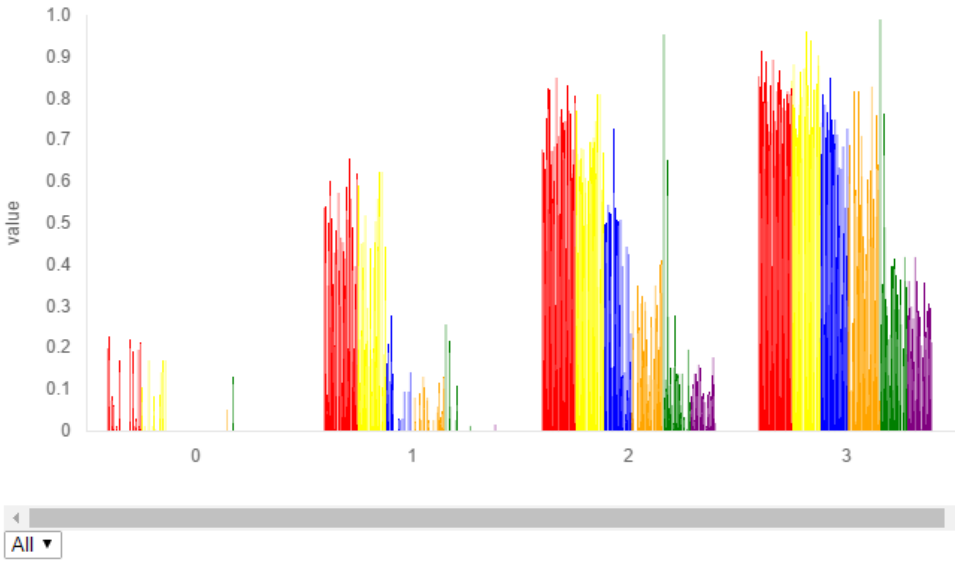


Figure 12: Four frequency bins. Bar charts with color coded strings.

The data above for each string shows all the samples regardless the fret that was played but a small clustering of the data is visible.

Using the dropdown menu in the bottom-left corner (set to *All* in the screenshot above) we can select which string data to be visible (all strings at once or one string individually). For example, below is the chart of the samples recorded only for the first string (note that the chart is cropped, only the first 4 bins are visible and the chart data is based only a part of all the samples recorded in order to improve performance while the chart is being drawn).

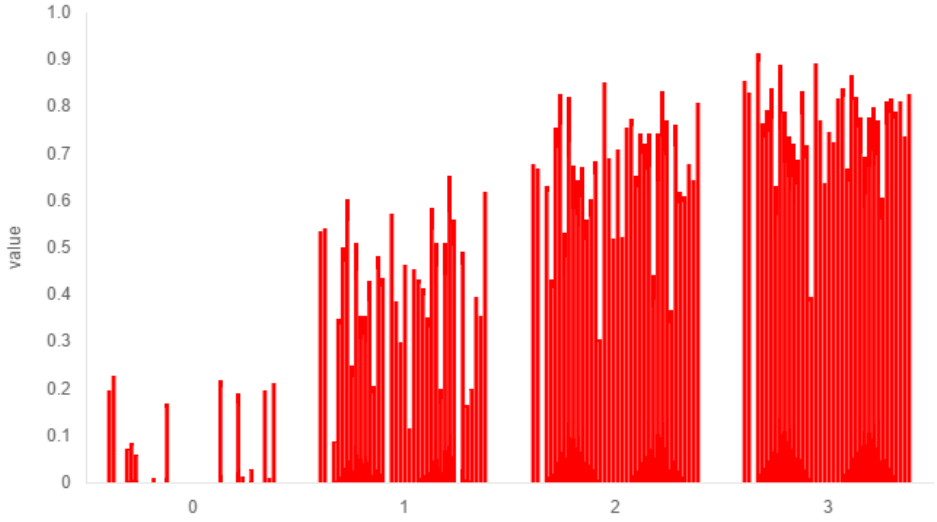


Figure 13: Data viewer chart for samples recorded for the first string.

Using all those charts I noticed that the values for frequencies above 4500Hz were always close to 0 and the ones above 1300HZ were also very small. This enabled the reduction of the neural network input size from 90 neurons to only 30 neurons by eliminating the part of the input data that was less relevant, leading to faster training time and more a lower recognition error.

This visualizer can be accessed by opening the **dataViewer.html** page inside a modern browser.

4.3 Filtering the sound

As explained in the learning chapter, the direct input from the microphone is too noisy to be fed directly a neural network. To fix this a *biquadFilter* (a filter method available for the *AnalyserNode)* was used.

The settings used for the biquadFilter were:

- Type: **bandpass** (a low-pass filter could have also been used, but band-pass worked better)
- Frequency value: **940** (this is the value, in Hz, that is close to the middle of the frequency range of the sounds produced by a guitar)
- Q value: **1** (this changes the width of the band-pass filter; this value was chosen through experimentation)

By echoing the recorded sound into the headphones (this was done by connecting the source node to the Audio Context destination *ctx.destination, as shown in the screenshot below*) I found the best

values to use for the *biquadFilter* in order to reduce noise but still keep the relevant frequencies of the sound input.

In the screenshot below the entire code required for receiving input from the microphone, filtering it and echoing it back to the speakers is presented. It also creates the *AnalyserNode* that is used for getting sound time domain data and applying the fast Fourier transform.

```
15      initAudioContext(stream) {
16          let ctx = this.audioCtx = new (window.AudioContext || window.webkitAudioContext)();
17
18          // Create an AudioNode from the stream.
19          let source = ctx.createMediaStreamSource( stream );
20
21          // Noise filter
22          let biquadFilter = ctx.createBiquadFilter();
23          biquadFilter.type = 'bandpass';
24          biquadFilter.frequency.value = 940;
25          biquadFilter.Q.value = 1;
26          source.connect(biquadFilter);
27
28          let outNode = biquadFilter;
29
30          // Setup the analyser
31          let analyser = this.analyser = ctx.createAnalyser();
32          analyser.fftSize = 1024;
33          let bufferLength = analyser.frequencyBinCount;
34          outNode.connect(analyser);
35
36          // Echo
37          outNode.connect(ctx.destination);
38
39          // Create reusable Array buffers
40          this.dataArray = new Float32Array(bufferLength); // for the waveform chart
41          this.fftDataArray = new Uint8Array(bufferLength); // for getting FFT byte data
42      }
```

Figure 14: The creation of the audio context and audio pipeline.

It is worth mentioning that thanks of the new HTML5 Web Audio API the above implementation was drastically simplified.

5 THE NEURAL NETWORK

Choosing the neural network architecture was one of the most challenging parts of this project. The neural network had to be small enough so that it could be trained in an acceptable amount of time but big enough to be able to recognize a total of 48 distinct classes. The architecture suffered drastic changes during the development period and all those changes will be mentioned in the following paragraphs.

5.1.1 The input data

The input data, as previously mentioned, is the normalized fast Fourier transformed sound data. Because the FFT size was set to 512 and the fast Fourier transform is symmetric (mirrored around the half of the interval) there were 256 input values available. Now, some of those values, more exactly the ones of the high-frequency bins were removed and only the first 90 bins were stored as input data. In other words, for each sound sample recorded the input saved was represented by 90 float values.

5.1.2 The initial architecture

Because starting with a complex architecture is never a good idea as there are too many unknown variables that have to be accounted for, the first architecture was used to test whether the neural network approach is feasible. In this version of the neural network, a perceptron with one input layer, one hidden layer and one output layer was used. The input layer had 90 neurons for the 90 input values, between 5 and 25 neurons in the hidden layers (this value was changed frequently in order to test which is the best size for the hidden layer) and only 6 output neurons. There were only 6 output neurons as only 6 classes were being recognized (one class per string) and no fret data was being stored.

The initial architecture was used to output which of the 6 strings is being (openly, not fretted) played. While this architecture was ok for recognizing the 6 required classes, the accuracy was lower than I expected so I decided to scrap this one and find another one that could also detect the fret number for each string.

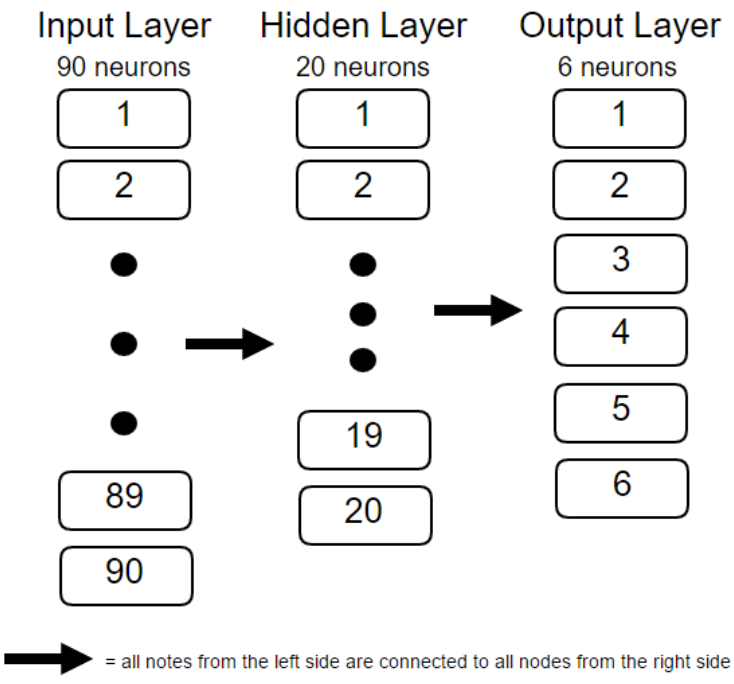


Figure 15: The initial network architecture

5.1.3 Separating the strings

One option to detect all the 48 classes would have been to simply set the output layer size to 48 neurons. In theory this should work, but in practice this adds a lot of unnecessary connections between the neurons, the network size grows exponentially. Because each string could be played individually from the others it meant that multiple outputs could be active at the same time, which for the sigmoid activation function being used meant that none of the outputs will have their value close to 1 (as the sum of probabilities for the outputs is 1). Considering this, the next network architecture was created by actually splitting the network into six separate networks: one for each string. By doing so, the networks have independent outputs and each of the networks can be trained on a more specific set of inputs, leading to faster training and higher accuracy. Because each string has 13 different classes (11 frets, one for string openly played and one for when the string is muted) each of the six networks has 13 output neurons.

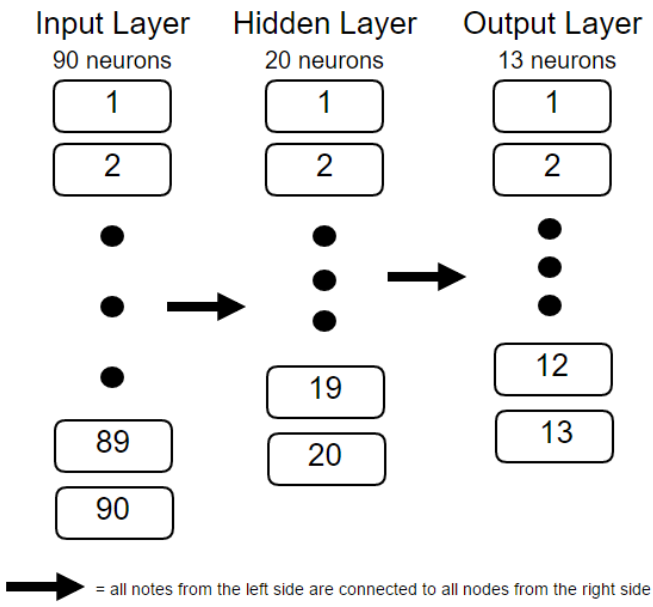


Figure 16: The structure of one of the six new separated networks.

This architecture, with six different networks, proved to be good enough in practice and using it the sounds were usually correctly classified. Now it was the first time a new problem appeared: the thickest, low frequency strings. It seems that recognizing low frequency sounds is much harder than sounds with a frequency of over ~100Hz or so. This problem occurs because an octave has the high frequency limit twice higher than the low frequency limit. This means that if we try to classify the note at 20Hz, the one which is one octave higher is at 40Hz, which is only 20Hz apart. But if the note

is at 200Hz, the note one octave higher is at 400Hz, being a whole 200Hz away. Because of how the notes on the guitar neck are arranged it was very hard to train the network to correctly recognize sounds played on the lowest two strings. Some improvements had to be made.

5.1.4 The final improvements

Keeping in mind the note regarding notes at low frequencies, I went back to the normalized fast Fourier transform chart and noticed that a higher resolution was needed at the lower frequencies spectrum. Because changing the sound sampling rate is not easily done inside a browser the FFT size was changed from 512 to 1024. This meant that instead of having 256 values per sound sample (remember that FFT is symmetric and only half of the data is relevant) now we had access to 512 values per sample. By further analyzing the charts it was noticed that, even though only the first 90 values of the 512 were saved as input data, the values between [30,90] were not varying that much (that interval, for a bin assigned to a 46Hz range, represents the sound with frequencies between 30*46 = 1,3Khz and 90*46 = 4,1Khz). By excluding the input data for frequencies over 1,3khz and only keeping the relevant values the input size was reduced from 90 values to only 30. With only 30 input values meant that the network only needed 30 input neurons.

The final architecture consists of 6 individual networks (one for each string), each one having 30 input neurons, 20 hidden neurons and 13 output neurons. Reducing the number of inputs improved both the training time and recognizing accuracy. The network is now able to correctly recognize sounds played on the low strings after being trained on a very small dataset (approx. 14 samples per class, 182 samples per string).

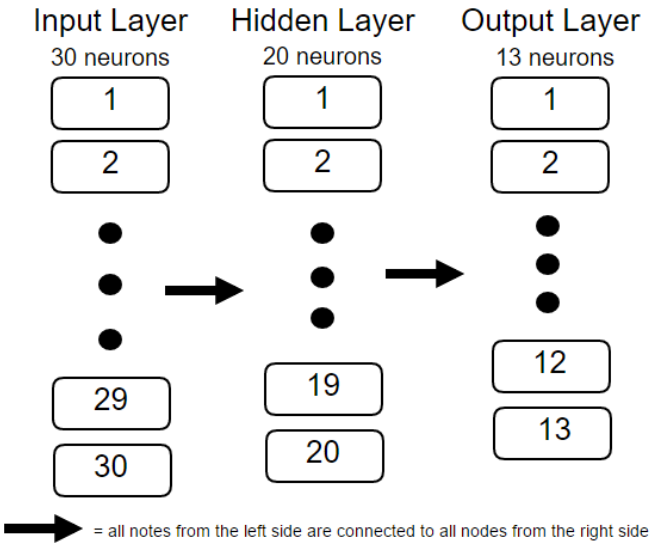


Figure 17: The final network structure with only 30 input neurons.

6 RECORDING LABELED TRAINING DATA

At the core of each machine learning software are the data models and datasets. As previously explained, the data model has been decided to have 30 input values and 13 output values for each of the 6 strings. In order to make use of the neural networks presented above a training dataset has to be provided.

6.1 The interface

Instead of searching for datasets of guitar sounds, which might have taken a lot of time or even worse, might give no results, an interface has been created that allows quick recordings of training datasets.

The interface has 6 rows (one for each string) with 13 buttons each (numbered from 0 to 12) where 0 represents the open string, [1-11] represents the fret numbers from one to eleven and 12 means that the string is muted. An *AutoRecord* button has been added that, when pressed, starts a routine that automatically clicks, in order, all the 48 buttons of the interface, i.e. sequentially records a sample for each output of each string. Keyboard shortcuts have been added on digits [1-6]. When key "2" is pressed on the keyboard a sample is automatically recorded for the open second string. An additional fret selection dropdown menu has been added so that when keyboard shortcuts are used, instead of recording for the output "0", the recorded output number is set to the selected fret number, e.g. if key "1" is pressed when fret #6 is selected, the next sample recorded will be labeled as belonging to the String1-fret6 class.

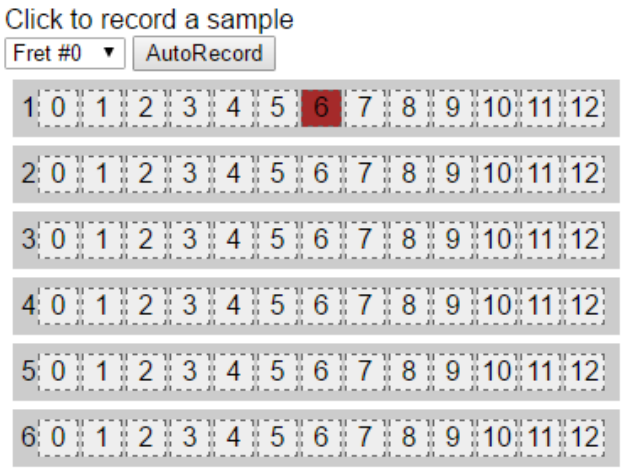


Figure 18: Sample recording interface before recording data for 1st string, 6th fret.

The way this interface works is: the user clicks on a fret number inside a specific string row, the box is colored in red, symbolizing a waiting state and after about 400 milliseconds the box will turn green

and the input from the microphone will be processed and stored as input data for the training sample and the index of the row and box pressed will be stored as output data.

Example:

- User clicks the box containing number "2" in the 2nd row

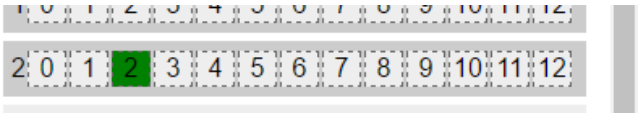


Figure 19: Green box while input is being recorded for 2nd string, 2nd fret.

- User plays the second string, second fret on his guitar while the box is green (preferably starts playing it while the box is in red, waiting state, so background noise is not incorrectly labeled as guitar noise).

- The recorded input data and marked output data is stored. In the console this can be seen:

  Data: Object {input: Array[30], output: Array[13], stringNumber: 1}

  The input array has 30 values, the output array has 13 values, 12 of which are 0 and only one value is set to 1, in this case the value *output[2]* is 1 because the fret number played was 2. The last value stored is *stringNumber* which stores the 0-indexed number of the string that was played, in this case the value is 1 (1st string played has stringNumber = 0, 2nd string played has stringNumber = 1, etc.).

All the training samples recorded, each represented as an object with three keys, as shown above: input, output, stringNumber, is stored in memory, inside the *dataSet* array. In order to have persistence between browsing sessions, a "*Save current dataset.*" button has been added that stores the contents of the dataSet array inside *localStorage*. The *"Load dataset."* button simply restores the in-memory dataSet array using the values stored in localStorage.

Pressing the "*Train on current dataset."* button will start the worker train procedure for all 6 neural networks using the current *dataSet* for training. Progress, while training, is shown into the console, logging the current learning rate and error after each 100 iterations. Once the required training error has been reached or the number of iterations exceeds the maximum iterations number set, the "*DONE #X"* message will appear, where *X* is the index of the neural network that has finished training. The final error, total number of iterations and time the training has taken is also displayed after the training has finished.

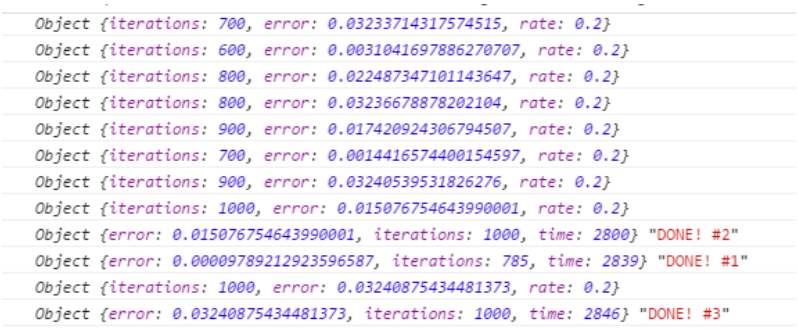


Figure 20: Log messages received from the training workers.

The way the final datasets were recorded for this project is by pressing the *AutoRecord* button and then playing all the notes on the guitar in order (the auto play starts with [string1 fret0], [string1 fret1] … [string 6 fret 12], where fret 12 actually means no sound being played).

Each *AutoRecord* dataset generated was saved individually and took under 2 minutes to record. In order to save each dataset in a different file, the custom *console.save(data, fileName)* method has been used, that saves the given data inside a new file with the given filename and extension. All the datasets were then combined in a single JavaScript file so the training dataset could be easily created by concatenating all the arrays in that file. Below is shown a fragment of the file that stores that recorded datasets.

```
1  var Tdata = [];
2
3  // Three strums per string
4  Tdata.push([{"input":[0.0625,0.23295454545454544,0.5795454545454546
5  Tdata.push([{"input":[0,0.15527950310559005,0.5838509316770186,0.79
6  Tdata.push([{"input":[0,0,0,0,0,0,0,0,0.1506849315068493,0.20547945
7  Tdata.push([{"input":[0,0.11560693641618497,0.5202312138728323,0.72
8
9  // Extra training data for low strings
10 Tdata.push([{"input":[0,0,0.2653061224489796,0.2653061224489796,0.1
11
12 // One strum per string
13 Tdata.push([{"input":[0,0.11627906976744186,0.627906976744186,0.744
14
```

Figure 21: The file that stores the recorded training datasets.

6.2 A big improvement

An odd thing started to occur: after recording more training data, the recognizing accuracy started to drop. It later turned out that the problem was with the difference in time between samples from when the note was played and when the data was recorded. In some samples the sound was recorded as soon as the string was hit, in other samples the sound was recorded while the string sound was fading out. To solve this issue a small improvement was made, which lead to much better recorded training data: instead of recording a single sample each time a button from the training interface was pressed, more samples at different time intervals were stored. In the picture below it can be seen that instead of a single sample, 6 samples at different time intervals are recorded every time a new box from the training menu is clicked:

```
85
86  function burstRecording($el) {
87      $el.addClass('waiting');
88      currentSample = 0;
89
90      // Record 6 samples instead of only one
91      setTimeout(() => storeSoundData($el), 350);
92      setTimeout(() => storeSoundData($el), 400);
93      setTimeout(() => storeSoundData($el), 425);
94      setTimeout(() => storeSoundData($el), 450);
95      setTimeout(() => storeSoundData($el), 500);
96      setTimeout(() => storeSoundData($el), 550);
97  }
```

Figure 22. Recording 6 samples at once. From the *input_recorder.js* file.

The first sample is recorded after 350ms so the user has time to position his fingers over the required string and fret to be played. After that, five more samples are recorded at different time intervals, last one being recorded 200ms after the first one (and 550ms after the user was first shown the red, *waiting state* box).

After this change was implemented, training the neural networks on the data recorded from a single round of *AutoRecord* samples lead to very good results. The final training datasets consist of five rounds of *AutoRecord* done for all the strings (four for when the strings are quickly strummed and one for when the strings are strummed only once) and an extra dataset for the two lowest strings, as the problem with low-frequency sounds was still noticeable.

7 THE GAME DEMO

The game demo that can be accessed by opening the *game/index.html* file inside a browser has been developed in order to show possible use cases of the sound recognizer created. The guitar game demo requires microphone access so that when the user is prompted to play a note it can decide whether the correct note was played or not.

7.1 Game mechanics

The core game mechanic is that the user has to play the first note visible in order to destroy it, and has time to do so until the note reaches the forbidden area at the bottom of the screen. By default, random notes appear at the top and slowly move towards the bottom of the screen. Each new note can only differ from the previous note by either the string number or fret number, not both at the same time, in order to make the game easier. There was also implemented the ability to have a custom list of notes (i.e. song tabs) that would appear in the given order. To use the *playlist* feature, the *js/Game.js* file has to be changed: in playlist array the notes, in order, have to be added as tuples of [stringNumber, fretNumber] and also the variable *playFromPlayist* has to be set to true, otherwise random notes will be spawned instead.

A score system was implemented, every time the user hits the correct note he gains +1 score and every time a note reaches the bottom forbidden area the player loses one of his three initial lives. When the player reaches 0 lives, the game is reset.

7.2 The UI

The game UI was designed with simplicity in mind. As few image elements as possible were used in order to minimize the game size (e.g. the note elements, the bullet that the player shoots and life icons are all create using only HTML elements stylized with CSS, no images were used for them).

Below is a screenshot of the game. The red text was overlaid to show the different game elements. The score and lives are in the top right corner. The notes are color coded, the color represents the string to be played and the number represents the fret to be played. The colors for the strings are, in this order, from the lowest to the highest string: red, yellow, blue, orange, green, purple.

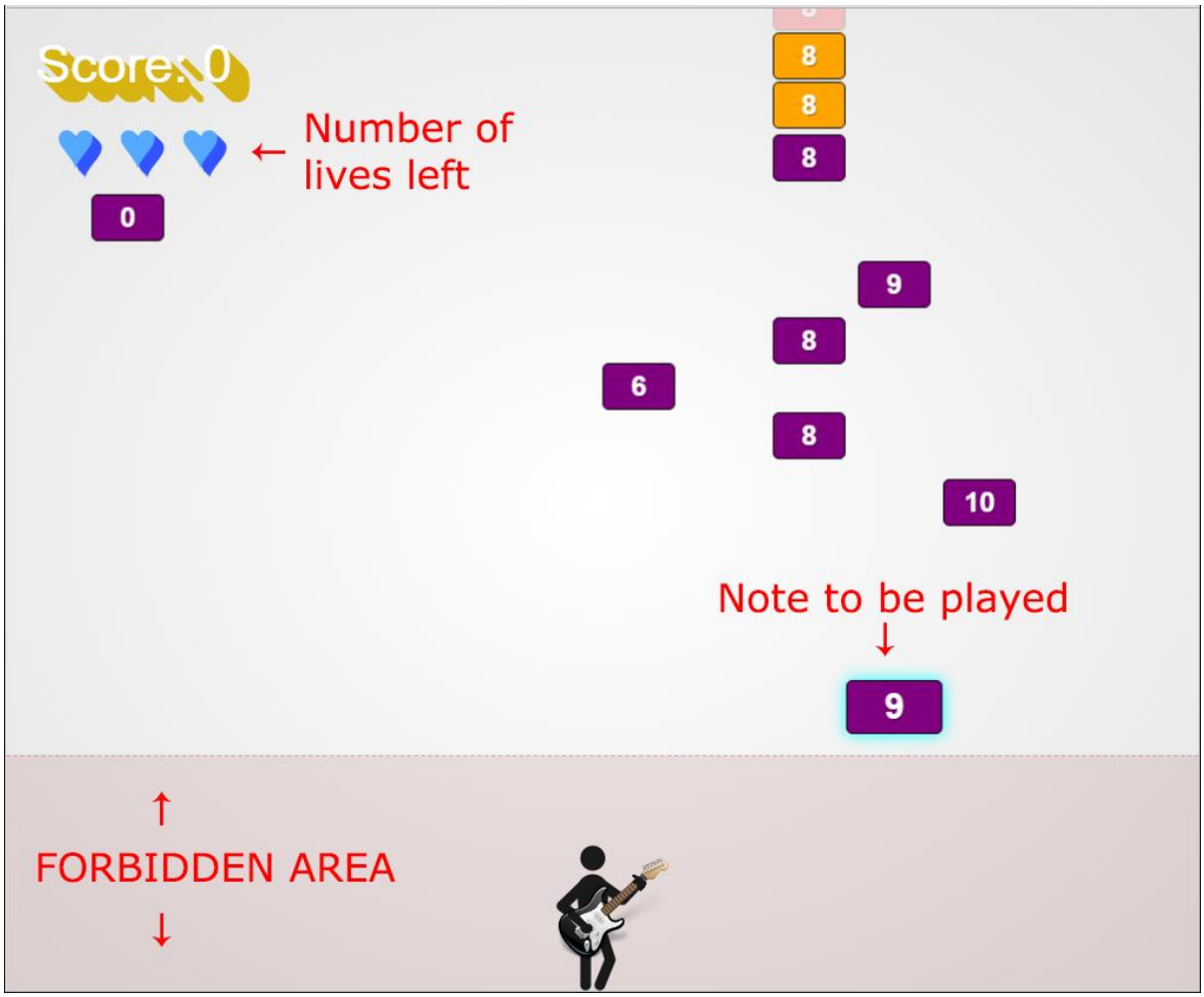


Figure 23: Screenshot of the game demo.

7.3 Improving clarity

Clarity in games refers to how visible and recognizable important elements of the game are and how easy is it to spot the changes in game that require user input. Clarity is a very important part of the game UI that directly affects the user experience. In the case of this game, clarity improvements had to be made so that the next note to be played is easily recognizable and also changes had to be made so the play can easily see whether he correctly played a note or not.

To better make the next note that has to played stand out there were made two major changes:

1. A colored, animated outer glow was added so the next note looks “special” when compared to the others.

2. The size of the note is changed by a looped growing animation as the human eye's attention is easily attracted by moving elements. Also, when the note is bigger, the fret number is easier to read.

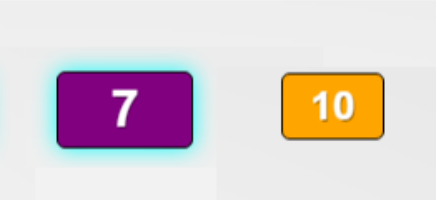


Figure 24: Note that has to be played at full size vs normal note design.

8 PUTTING IT ALL TOGETHER

Once the game mechanics and UI have been completed the next step was to integrate the recognizing software to the game. Before this integration a special game mode has been created in order to test the game UI; in this special game mode notes were automatically set as being correctly played after random time intervals.

8.1 Exporting the activation functions

The neural network codebase (composed of the Synaptic.js library plus the function calls to that library) includes functionalities like training, testing or backpropagation that are not required when all that is needed is to activate the network on a specific output. The Synaptic.js library provides a *standalone* method that returns the source code for a single activation function, that accepts one parameter (the input to activate the network for) and returns an Array object containing the output data resulted after activating the network on the given input. Using this *standalone* function and the *console.save* method a new procedure, *exportNetworks* was created that, when called, saves the activation function in a separate file named *Recognizer.js*. This file actually contains 6 activation functions (one for each of the 6 networks) and exposes them through the global array *activateNetwork*.

Once this file has been created, it was included in the game demo and it is used using this syntax: activate *activateNetwork[string](input)* where *string* is the index of the string we want to recognize (between 0 and 5) and *input* is the processed sound captured by the microphone. This returns a probabilities array containing *13* float numbers with values between 0 and 1, representing the probability that the fret given by the index of that value was played.

```
60  // Export networks function to standalone file
61  function exportNetworks() {
62      let out = `var activateNetwork = []; \n\n`;
63
64      for(let i = 0; i < networks.length; ++i) {
65          let net = networks[i];
66          let func = net.standalone();
67          func = 'activateNetwork.push(' + func + '); \n\n';
68
69          out += func;
70      }
71
72      console.save(out, 'Recognizer.js');
73  };
```

Figure 25: The function used to export networks, found inside the neural_network.js file.

8.2 Detecting when a note is played

After the neural network activation function was integrated a new question appeared: how often should the activation function be called? In other words, how often should we check whether a note was played or not? The average guitar player can play at a highest speed of approx. 300bpm (beats per minute) that is a 200ms between each note. But using a 200ms polling time would make the game unresponsive and hard to play as the player would need to have a 100% accurate timing when playing at 300bpm, which is impossible to achieve. To solve this problem, the time interval between network activations was set to a much lower value of 16ms, value that reduces both the perceived input latency and the problems with the game not "listening" for the note played at the right time. To further reduce the input latency and reduce the number of false-positives found by the neural network, the activation function is only called when the sound volume increases (the activation function is actually called, but the output is saved in an auxiliary variable and only used when the sound volume at the current frame is higher than the sound volume of the last frame). Also, after a note was played and recognized, in order to reduce the chance of duplicate notes to be incorrectly recognized (e.g.: if note X has to be played twice in a row the software will incorrectly mark both required notes as being played, even though the user only played the note once) a small delay is added so the note has to actually be played again in order to be correctly recognized.

9 POSSIBLE IMPROVEMENTS

9.1 Recognizer improvements

Firstly, ways through which the recognizing accuracy can be improved will be discussed. The changes suggested are related to the structure of the neural network and the training methodology.

9.1.1 Even more specific neural networks

In its current form, the neural network architecture presented consists of six different networks, one for each string. While separating the networks was a good idea, this can be further improved. The network was trained with samples recorded from an acoustic guitar, but the network was tested using an electric guitar too. It turned out that the lower strings were not detected as good for the electric guitar. A solution would be to also record samples for the electric guitar for all the six networks but the problem is that having more diverse samples means that the training will be slower and the same network would have to recognize a broader range of sounds. The suggested solution, that should solve the recognition problems for the different types of guitar, is to have another 6 networks for each guitar type which would be only be trained on sound samples coming from that specific guitar type, e.g. have 6 networks (one for each string) for recognizing acoustic guitars sounds, 6 networks for recognizing electric guitar sounds, etc. And, to get the final output of the recognizing software, the different networks will have their outputs combined and forwarded to another network that has the same number of outputs. For example, if we were to have 6 networks for an acoustic guitar and 6 networks for an electric guitar, the output from the first "acoustic" network will be combined with the output from the first "electric" network (a total of 13+13 = 26 outputs) and fed into a new, "combined" network that has 26 inputs and 13 outputs, i.e. the combining decides for each fret of each string whether to take the result from the "acoustic" network, the "electric" network or a combination of those two. By using this boosting technique, the final combined network should be able to correctly classify sounds coming from different types of guitars.

9.1.2 Boosting

Let's go back to the case when a single guitar type is to be recognized. With the current architecture, the outputs from the 6 strings are independent (the 6 networks are not connected in any way). The recognizing accuracy could be improved by adding additional networks that have the same number of outputs, but instead of taking inputs from the recorded data, they take as inputs the outputs from the existing networks.

Another network that could be created, is actually the one described in *The initial architecture* section of *THE NEURAL NETWORK* chapter: a network with only 6 outputs that shows which of the 6 strings is being played. This output data could be used as an input for our 6 different networks so the final output would more accurately recognize whether a specific string was the one that was played (this should eliminate cases when the networks positively recognizes sound from other strings as coming from the string assigned to them).

9.1.3 Larger training dataset

As previously mentioned, one of the most important parts of machine learning algorithms is the training dataset. In most cases increasing the quality or quantity of the training data corresponds to increased classifying accuracy. A larger training set would improve the recognizing accuracy of this application, even more if the new training data covers different cases and environments (sound samples with high background noise, sample sounds of guitar played with a guitar pick, sample sounds of guitar played with the fingers, different guitars, different microphones, etc.).

9.2 Game improvements

The game improvements suggested below mostly referrer to new features being added. The game demo itself is pretty basic and has been only developed for showing potential use cases for the recognizing software, but with some improvements it can be updated into a full-featured educational game for the masses.

9.2.1 Export to mobile devices

One of the first words that comes to mind when the word “guitar” is mentioned is the word “freedom”. Guitar players can take their instrument everywhere travel (unlike drummers, pianists, etc.). Having the game demo as a desktop game only limits the reach to the target audience. By having the game as a mobile app, the player would only require having his smartphone and guitar with him for taking advantage of the game’s benefits. Being a web browser application it can easily be exported to Android and iOS devices using cloud compilers like PhoneGap or Coocon.js. To use those compilers, the archive containing the web project has to be uploaded and after a short time the compiled iOS and Android apps can be downloaded.

9.2.2 Upload your own tabs

Playing random notes is good for improving muscle memory and muscle strength but does not really help the user to learn how to play good sounding music. A useful feature would be implementing a tab-sharing directory where each user could upload playable tabs and also browse and rate tabs from other players. By focusing on the social part of the game users will both have access to more *playlists* and also be motivated, by a public scoring system, to keep practicing a certain piece until their score is higher than others.

9.2.3 Progression system

As different players have different guitar skills and different experience levels a progression system should be implanted that makes the game easy to pick up by beginners and challenging for moderate

or even experimented players. This could be implemented by either a leveling system (where the user starts at level one and gradually gains experience) and different game modes and playlists could be unlocked, based on their difficulty and the user level. Another progression feature could be added for each song: when a certain song is first being played only some notes are required to be played and as the player successfully plays them the complexity is gradually increased until 100% of the song notes are shown to the user.

10 THE DEVELOPMENT ENVIRONMENT

10.1 Main software and hardware used

The final software is the form of a web application, so the development environment used includes:

- **Google Chrome** was the main browser used while developing this project but other browsers such as **Mozilla Firefox** or **Opera** were also used in order to check cross-browser compatibility. I have chosen to use **Chrome** because it has, in my opinion, the best designed and most useful developer tools from all the browsers. Some of the developer tools used were:
    - Profiles - Heap snapshot: which shows the memory distribution among the JavaScript objects used on the current page. This was used to detect and fix memory leaks by taking snapshots at different time intervals.

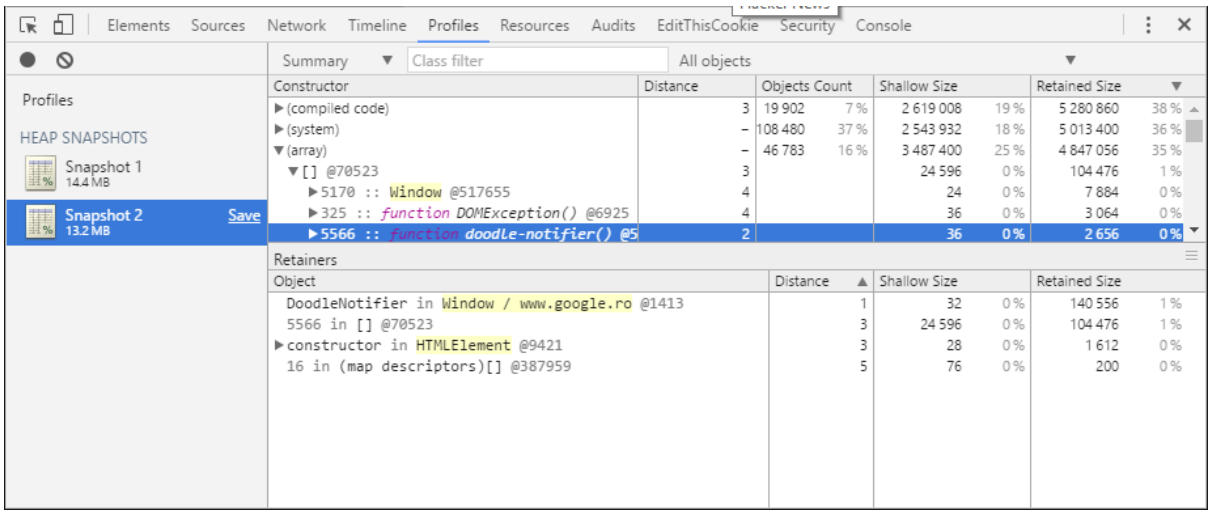


Figure 26: Google Chrome Heap Snapshots

- Sources: Not only does this tab show the current JavaScript scripts included on the page but it also shows the content of the *WebWorkers* that are currently running in background. This allowed me to test and improve the *workerTrain* functionality of the *synaptic.js* library.

- **Sublime Text 3** as the main code editor. I used this editor because, first of all it is free, and is one of the most popular editors for web development. This popularity comes from the fact that there is a great number of packages available for it. Some of the packages that proved to be helpful while developing this project are *DocBlockr*, *Emmet*, *JSFormat*, *SublimeCodeIntel*.

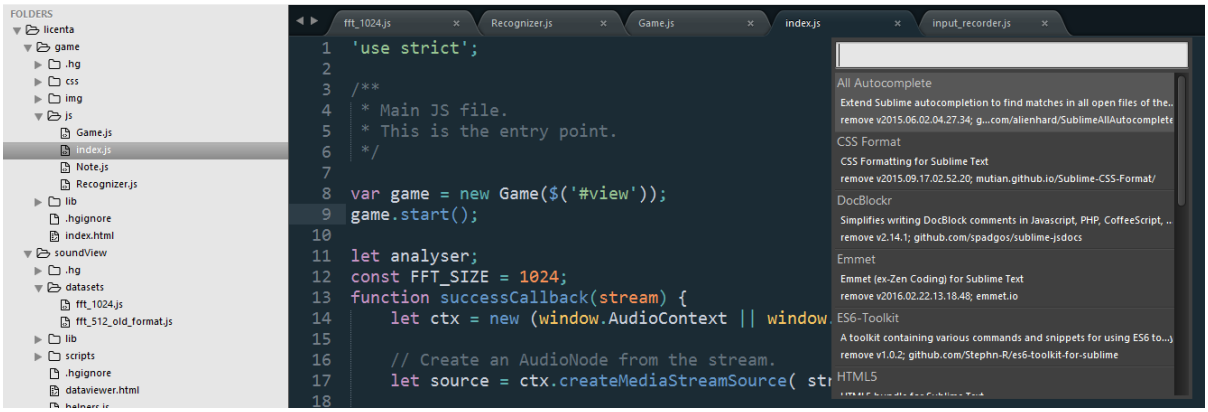


Figure 27: Sublime Text 3 with Oceanic Next color scheme

- PC configuration – because recording and processing the sound data and training the neural network are costly computational processes, the PC hardware configuration was important in order to reduce the amount of time it takes to obtain the trained neural network. The relevant components that affected the training time are the CPU (Intel Core i5 3470) and the amount of RAM (12 GB at 1333MHz). The GPU (GTX 660) is only relevant for the performance of the demo where different elements are animated by changing the CSS3 transform3D property, which in fact uses the GPU being part of the hardware-accelerated CSS properties.

- Extra hardware:

    - **Microphones** to test the application and possibly to create training data. Multiple microphones were used in order to make sure that the training data is not biased towards some specific microphone characteristics. Microphones used include the microphone from a Microsoft LifeCam webcam, a direct connection between the electric guitar output and the Line In input from the sound board, the microphone from a laptop.

- **Acoustic Guitar** - good for training data and testing as there are no connections that have to be made, sound being recorded through an external microphone.
- **Electric Guitar** - a lot less noise as no microphone is being used so it can be used to calibrate and create the auxiliary software required for sound analysis
- **Mobile devices** (Apple iPhone 5C smartphone and Asus Nexus 7 tablet) which were used to test the final demo. Because the application is a web application, running the demo on those devices was easily done by accessing, through the default browser, the web server where the demo was located.

10.2 Version control software

In order to keep track of all the file changes made and to be able to revert the codebase to an earlier version in case something goes wrong, a version control software was used. The software used was Mercurial along with the TortoiseHG workbench. Version control is usually used when multiple people are working on the same project but it is also very useful when a single person has access to the codebase as it easily enables to create branches for different features or to see exactly what was changed since last commit. Having version control acts like a “safety net”, especially when a back-up of the repository is also created. To back-up the repository all the changes made were also pushed to a Dropbox folder that is safely stored in the “cloud”.

Two distinct repositories were created: one for the recognizing software development and one for the game demo. The screenshot below shows the TortoiseHG interface and the commits that were done during the development of the sound recognizing software.

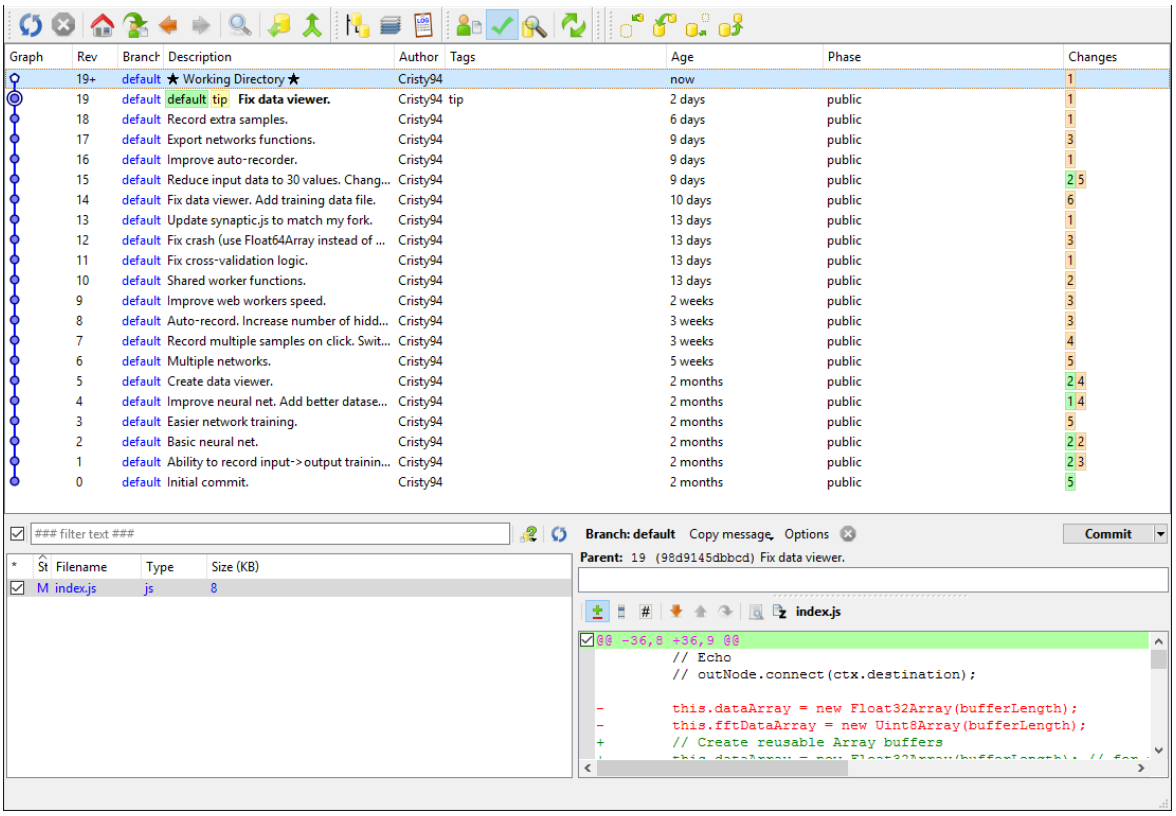
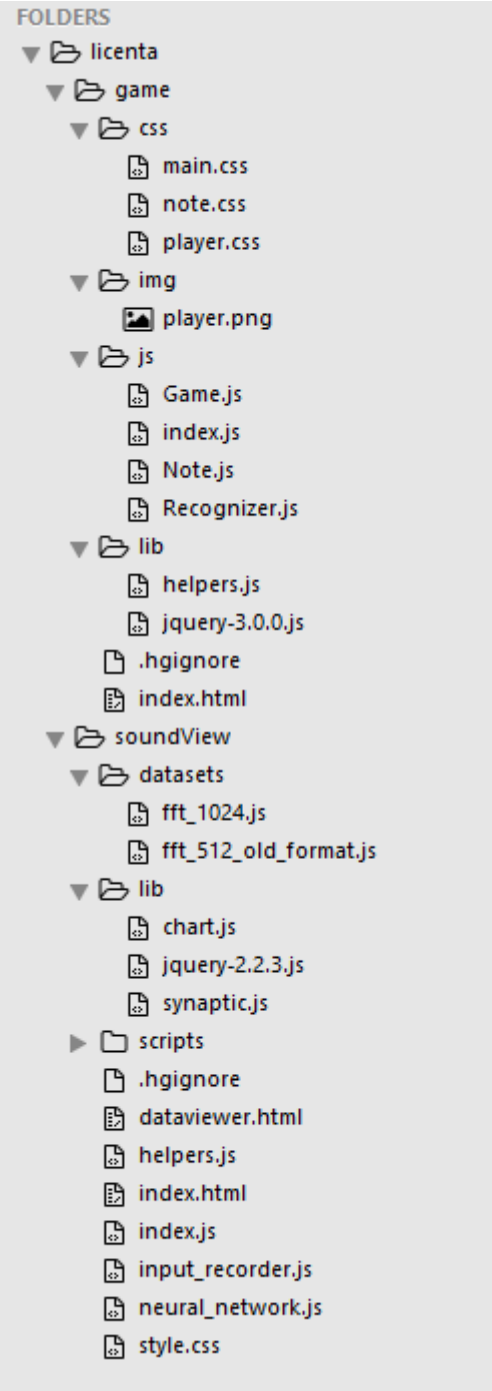


Figure 28: TortoiseHG workbench containing the commit history for the sound recognizing software.

10.3 Software application structure

The folder structure is detailed in the screenshot below. Explanations for the most important folders and files are presented in the right-side list column.



- **licenta** – root folder of the project
  - **game** – subfolder for the game demo
    - **css** – stylesheets
    - **img** – all the images used for the game
    - **js** – custom javascript files
    - **lib** – javascript libraries used
    - **index.html** – game entry point
  - **soundView** – subfolder for the recognizing software
    - **datasets** – recorded training data
    - **lib –** javascript libraries used
    - **scripts** – folder containing only the **dataviewer.js** file, which is used only in the dataviewer.html page
    - **helpers.js** – file containing auxiliary functions for normalizing and shuffling an array and the implementation for *console.save*
    - **index.html –** main entry point for the recognition software interface
    - **index.js** – main JS file, containing the *Visualiser* class and its instantiation
    - **input_recorder.js –** file containing the code responsible for recording training data
    - **neural_network.js** – here the neural networks are created and functions to train and export the networks are exposed

11 CONCLUSIONS

Machine learning, even though at first look seems a very complex and deep field, is very easy to approach as a beginner. Knowing the basics of data modelling and training samples generation is enough to get started with a machine learning project. Learning by doing is one of the best ways to improve knowledge in the artificial intelligence field, as the computers these days are powerful enough to be used for numerous trial and error experiments in a short amount of time. In the specific case of neural networks, trying different network architectures or changing can actually be automated so that the best architecture for a specific problem is found by procedurally generating, training and testing different networks.

This project, even though complex, containing many "moving" parts and different modules, was brought to completion by starting simple, with as few basic components as possible and then slowly adding more components and improving the existing ones until the final requirements are met. As a general rule, always being able to test the newly added functions, integrated in the application, as soon as they were created led to reduced development time and also fewer bugs were created. So, one of the rules followed while developing this project that reduced development time is: start small and iterate.

While developing this project, another time-saving practice that proved efficient was to spend a bit more time at the start of the project to create auxiliary tools help automating tasks that would either be time consuming or tedious. Creating the real-time graph charts for the input sound data and processed sound data helped with quickly finding the data model that can be used as a good, relevant input data for the neural networks. Similarly, using the *AutoRecord* procedure significantly improved the training data recording procedure. Having to manually click each button while holding the guitar in my hands and quickly trying to play the required noted was so hard to do that it could take over 10 minutes to record a single training dataset. Again, the lesson learned from this was to minimize the time it takes to do any task that takes longer than it should, even if this means spending more time at the start of the project to create the required helper functions.

12 BIBLIOGRAPHY

[Mih13] – *Applied Intelligent Data Analysis – Algorithms for Information Retrieval and Educational Data mining,* Publisher *Zip Publishing*, 2013

13 WEB REFFERENCES

[Mar05] - Artificial Neural Network for Speech Recognition
http://www.cs.toronto.edu/~ruiyan/csc411/ANNSpeechRecognition.pdf

[Gra13] - SPEECH RECOGNITION WITH DEEP RECURRENT NEURAL NETWORKS
http://www.cs.toronto.edu/~fritz/absps/RNN13.pdf

[LumiNND] – Neural Networks Demystified
http://lumiverse.io/series/neural-networks-demystified

[Bur05] – Music and Computers - A Theoretical and Historical Approach
http://music.columbia.edu/cmc/MusicAndComputers/

[ML-Ng] - The Machine Learning course from Stanford University, taught by Andrew Ng.
https://www.coursera.org/learn/machine-learning

[BBCEdu] – Representing text, images and sound
http://www.bbc.co.uk/education/guides/zpfdwmn/revision/3

[WikANN] – Wikipedia Artificial neural network article
https://en.wikipedia.org/wiki/Artificial_neural_network

[WikBaPa] – Wikipedia Band-pass filter article
https://en.wikipedia.org/wiki/Band-pass_filter

[WikFFT] – Wikipedia Fast Fourier transform article
https://en.wikipedia.org/wiki/Fast_Fourier_transform

[GitSyn] – Synaptic.js library hosted on GitHub
https://github.com/cazala/synaptic

[ChartJS] – “Simple yet flexible JavaScript charting for desginers & developers” by Nick Downie
http://www.chartjs.org/

[WebAud] – Web Audio API specifications
https://webaudio.github.io/web-audio-api