Compose AI

CSI4106: Project Report

December 12, 2018

Group # 44
Aniela Opolski 7629248
Paul Laplante 7719659

Introduction

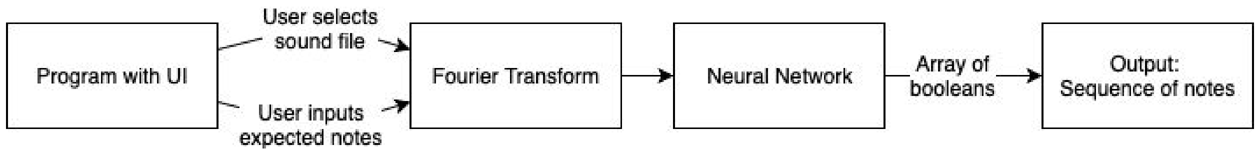
Compose AI is an exploration of our own approach for an application, project type 9. The application is designed to eventually be used in a mobile app that recognizes musical notes and transcribes them to sheet music. Music transcription done by humans is a task that is difficult for most. Sometimes two different people will transcribe the same song differently from each other, which can be seen on websites such as ultimate-guitar.com, which has multiple different versions of music chords and tabs as transcribed by different musicians for a single song. Music transcription is also a task that can be very tedious for most musicians, as it forces them to put down their instrument and write each note. Thus, using AI to transcribe music would be highly beneficial to the musical community, and music beginners.

The application that we chose to explore is AmadiOS, a side project that attempted to write sheet music using a phone's microphone. Some issues seen in AmadiOS were the lack of octave recognition, the app would only recognize notes in one octave. The goal of our exploration was to implement our knowledge from this class to improve the accuracy of the application.

For the context of this project, we have decided to reduce the scope of the application to just note recognition. The application will be coded in such a way that the neural network is independent from the application used to train it, so that it can then be reused in a mobile application to do music transcription. The purpose of the application we've created for this project is to train the neural network to recognize musical notes. The sound files that we will use will have notes of equal, predefined length. That way the application doesn't need to recognize how long each note is, it only needs to focus on pitch recognition.

Description of the Application

This application is a windows app created in C#. The application will have two sections, the training part and the testing part. The training part of the application is designed to take in a wav file as well as expected results, and then train the network to recognize those results. The testing part of the application will take in the same thing, but instead of training the network it will simply show the results and the accuracy of the network.



The neural network is written in a way that it's completely serializable, so that it can be saved before the application is closed and loaded each times the application is opened. That way once the network is trained, it can be saved and loaded in different applications. The network is serialized to xml specifying the weights for each node in each layer of the network. A sample of the saved network is attached to the submission of this report as SavedNetwork.xml.

As described by the block diagram above, we have 3 main modules, UI, Fourier Transform and Neural Network. The neural network has 3 submodules, the network class, the layer class and the node class. The Neural Network module uses knowledge learned from this class, specifically gradient descent.

Configuration of Neural Network

Theory and relevant information

When it comes to the frequencies of musical notes, there are two different systems that exist: just temperament and equal temperament [4]. Just temperament (sometimes also called harmonic tuning) uses the natural frequencies of notes as a basis to determine frequencies. When playing music, just temperament has a tendency to sound better, but instruments that are tuned to play in just temperament are only able to play in one key. This principal is sometimes used in orchestras, but in general equal temperament is used because it allows instruments to play in any key, and the human ear generally isn't good enough to tell the difference.

Because the majority of instruments use equal temperament (including guitar and piano, which are two very popular instruments), we've decided to use equal temperament for the context of this project.

The formula for frequencies using the equal temperament scale is as follows:
$f_n = f_0 * (a)^n$
Where
f_0 = the frequency of a fixed note that must be defined.
n = the number of half steps this note is above the starting note.
f_n = the frequency of the note that is n semitones above the note f_0.
$a = 2^{12}$ because when it multiplied by itself 12 times it is equal to 2.

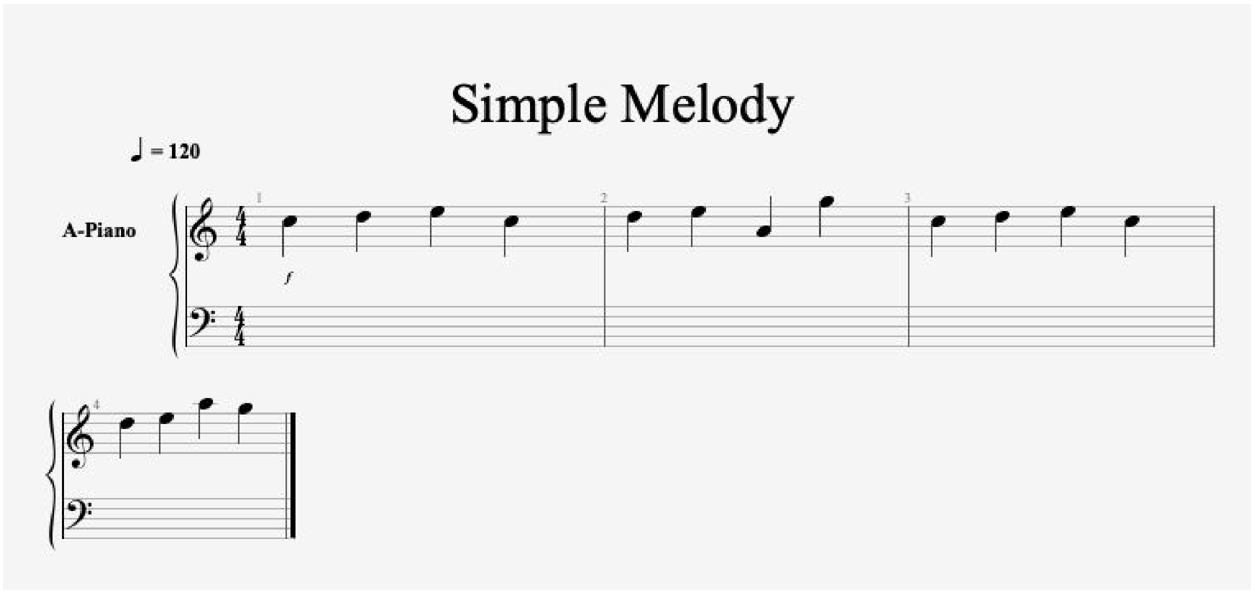
To understand the value of *a*, it is important to understand that there are 12 semitones in an octave, and that doubling the frequency of a note will give the same note again but none octave higher.

We've decided to use A_0 (which has a frequency of 27.5Hz) as our value for f_0, and the highest note that we are trying to detect is the C8 (which has a frequency of 4,186Hz). These two notes are the highest and lowest notes on most keyboard [6]. The human ear can typically hear sounds within the range of 20 to 20,000 Hz [5].

The network is configured to have one input corresponding to the frequency of each note of the piano, which represents 88 inputs. Our plan was to have one hidden layer with 44 neurons and then one output layer, with 36 outputs (which represents 3 octaves). We wanted to start with these values and then increase the number of outputs if we felt like that didn't require too much processing power. We were hoping we'd have time to do some testing with different configurations to see which one was the best, but because we ran into challenges with our implementation of the fourier transform we didn't have enough time to do any testing.

Data Used

Our dataset for training includes 6 wav files of the simple melody seen below. Each wave file was this melody in a different octave. This was created for training purposes, so that our model could detect the same notes in different octaves. Wav files attached to submission.



Pseudo-code Algorithms From Our Implementation

```
[3] Algo FourierTransform
        //reverse the data
        Reverse(data)
        // do transform
        double sign = forward or backward  // using forward for our implementation
        mmax = 1
        while n > mmax:
                istep = 2 * mmax
                theta = sign * PI / mmax
                wr = 1
                wi = 0
                wpr = Cos(theta)
                wpi = Sin(theta)
                for m = 0 to istep, m += 2:
                    for  k = m to 2 * n, k += 2 * istep:
                        j = k + istep
                        tempr = wr * data[j] - wi * data[j + 1]
                        tempi = wi * data[j] + wr * data[j + 1]
                        data[j] = data[k] - tempr
                        data[j + 1] = data[k + 1] - tempi
                        data[k] = data[k] + tempr
                        data[k + 1] = data[k + 1] + tempi
                    t = wr // trig recurrence
                    wr = wr * wpr - wi * wpi
                    wi = wi * wpr + t * wpi
          mmax = istep;

Algo gradientDescent:
        float delta
        float nabla //weight
        float gamma //bias
        for int i = 0 to weights.Length:
              weights[i] = weights[i] + nabla * delta * outputDerivative(lastInput) * lastInput[i]
        bias = bias + gamma * delta * outputDerivative(lastInput);
```

Implementation

The main application is a Windows application that was developed using Visual Studio. Currently the solution requires Visual Studio to run. To run the code, clone the repository and open the solution using Visual Studio.

Unfortunately our implementation does not match the application we described, because we didn't have enough time to overcome all the challenges we faced while trying to implement the system. That being said, a lot of improvements were made compared to the original implementation of AmadiOS.

The major improvement that was brought to the application was the use of the Xamarin platform. The original application was coded in Swift for iOS, and only ran on iPhones. This made it long and hard to train the network, because all the processing was done by the phone. In our current implementation, the neural network is implemented inside of a portable class library, which means it can be referenced by multiple different types of projects. This allowed us to create a Windows application that references the network and uses is for training purposes.

A very popular way to create cross-platform mobile applications using Xamarin is by using portable class libraries. By default, when creating this type of project, three projects are created: one portable class library that contains all the code that's common throughout the application, and two more projects which contain native implementations for Android and iOS. We configured the neural network in such a way that it can be referenced by Windows applications and cross platform mobile applications.

Here's a link to the Github repository containing the project: https://github.com/extraPaul/AI-Project/tree/master

Future Work

We plan to complete this project for personal use, as a sheet music app would be very beneficial to our musician lives. Going forward we will either have to optimize our current C# code, or do the training of the model in python, save the network to the xml file mentioned earlier and load that saved network to the xamarin app.

Challenges

Throughout this project we faced some challenges in our implementation. One challenge was with reading the wav file into a format that the FFT could take in. Some FFT libraries expected a complex array as a parameter and others expected a double or float array instead of the byte array we get from the file stream. When we tried using out own implementation of the FFT algorithm it was really slow, which forced us to change implementations. However, this might be due to the fact that the file stream ended up giving us a byte array with the size 300000! Most of the bytes were just 0 or null, and there didn't seem to be much of a pattern in the array either. Another challenge actually happened to be in the FFT module. We tried many different FFT libraries, some took over 30 min to complete, and others finished but returned the same data that was inputted.

Lessons Learned

One of the biggest lessons learned from this project was don't use C# for AI. There is very little resources for AI in C# and the resources we could find were too advanced for our knowledge of the subject to complete the implementation. We discovered a little late that python actually had the libraries we needed to successfully implement this project. Our reasoning behind using C# for development was that we could build it into a dll that could be used by Xamarin for a future mobile app. In hindsight we would have used python since C# caused us a lot more grief than it was worth.

Conclusion

In conclusion, we believe the approach we had would have given us better results than the original implementation of AmadiOS but unfortunately due to the challenges we faced we were unable to prove this claim. Overall we learned a lot about music recognition and the challenges of working with sound in a digital format. Based on the work that was originally done with AmadiOS we know it's possible to do note recognition using neural networks, but it might not be realistic to perform such note recognitions on a cell phone because of the amount of processing power necessary.

References

[1] "Musical note", *En.wikipedia.org*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Musical_note. [Accessed: 12- Dec- 2018].

[2] "Formula for frequency table", *Pages.mtu.edu*, 2018. [Online]. Available: http://pages.mtu.edu/~suits/NoteFreqCalcs.html. [Accessed: 12- Dec- 2018].

[3] C. Lomont, "FFT Implementation", *Lomont.org*, 2018. [Online]. Available: http://www.lomont.org/Software/Misc/FFT/LomontFFT.cs. [Accessed: 12- Dec- 2018].

[4] "Just vs Equal Temperament", *Pages.mtu.edu*, 2018. [Online]. Available: http://pages.mtu.edu/~suits/NoteFreqCalcs.html. [Accessed: 12- Dec- 2018].

[5] "Frequency Range of Human Hearing - The Physics Factbook", *Hypertextbook.com*, 2018. [Online]. Available: https://hypertextbook.com/facts/2003/ChrisDAmbrose.shtml. [Accessed: 12- Dec- 2018].

[6] “Chasin Conversion Chart”, *Audiology.org*, 2018. [Online]. Available: https://www.audiology.org/sites/default/files/ChasinConversionChart.pdf. [Accessed: 12- Dec- 2018].