Major in CS (no Minor) – cjm
Major in CS (no Minor) – mbm
Major in CS (no Minor) – kpp
Major in CS (Minor in Philosophy) – ss

Music Genre Classification in Hebbian-Based Spiking Neural Networks using Textural Feature Extraction from Spectrographic Imagery

Christian Martin (cjm378)
Shashank Sharma (ss2795)
Karn Patel (kpp67)
Mohammad Bilal Memon (mbm186)

12/15/17
443/525 Brain Inspired Computing
Prof. Konstantinos Michmizos

Abstract

The rise of Big Data, Social Media, Cloud Computing, Machine Learning, and the re-emergence of artificial (deep) neural networks (ANNs) have together given rise to an impressive array of multimedia classification and recognition systems that approach, or even, in some cases, surpass human-level accuracy. Most of the methods for multimedia classification, which include tasks such as facial recognition, image categorization, video content detection, and music genre determination, are based on classical machine learning paradigms, e.g. SVMs, or so-called second generation ANNs, e.g. Convolutional Neural Networks. However, third-generation ANNs, based on biologically plausible Spiking Neural Networks (SNNs), have not been widely employed for multimedia classification. In this paper, we present a method for music genre recognition using textural features extracted from spectrographic image representations of musical files retrieved from the GTZAN music database. We show that the spectrographic representation of music provides enough cross-class differentiation to adequately train a SNN using Hebbian-based learning such that it can accurately classify music between two highly dissimilar genres (near 81.2% between Metal and Classical), while showing moderate accuracy between acoustically similar genres (Pop and Disco). Our results approach some of the better results under Deep Learning-based classification schemes, thus affirming the promise of brain-inspired computing approaches for artificial intelligence-based tasks.

1. Introduction

Music has been an integral, culturally resonant part of human civilization for all of our history. The digitization of music has increased demand for music immeasurably. As such, it has been a major part of the movement to digital, cloud-based, on-demand delivery of multimedia content, which must be catalogued and searchable using increasingly sophisticated techniques. Traditionally, music is classified by genre by the producer and conveyed via metadata encoded in the packaging. During distribution, human curators select, categorize, and present music based on genre as specified in the packaging. This is the classic approach employed by record stores for decades.

Recently, the rise of Internet-based music services has changed the way people all over the world consume music. With the ability to have music delivered instantaneously from anywhere, and from any producer, the ability to search for, select, and curate music according to a preferred class has become more important than ever. AI-based recommendation systems have given rise to more sophisticated curation schemes, ones that create sub-genres of music driven by mood, texture, instrumentation, melodic structure, and so on, thereby making traditional classification schemes near obsolete. Most of the recommendation systems in use today employ some form of second-generation Artificial Neural Network, also known as a Deep Neural Network, to classify and categorize music based on genre and sub-genre (among many other characteristics). These mechanisms have proven highly effective, allowing users to hear nearly anything that suits their desire with the tap of a screen or a verbal request to an AI agent in the cloud.

Traditional methods of music information retrieval are based on feature extraction from the waveforms and from the timing, frequency, positional, timbre, pitch, beat, and intensity information encoded within a digital representation of a musical track. Music theory is quite

complex, however, so an intuitive understanding of acoustic theory and musicality would be beyond our scope to attain. Instead, we chose to approach the music genre classification scheme based on inspiration from [1, 2, 11] that perform classification using image-based spectrograms, Mel-spectrograms, and Mel frequency cepstral coefficients (MFCC). Because image classification is so widely successful using ANNs, including SNNs, we chose to use textural data from a spectrogram and perform image classification on the spectrogram in the same way as a CNN detects pictures of cats from a set of pictures. This approach saved us from having to extract complex auditory, acoustical, and other musical features from a musical track, and instead focus on just the image of a musical piece as represented in the spectrograms. Figures 1 and 2 show a pair of spectrograms from two genres, classical an reggae. A simple glance reveals a similarity in class and a difference between classes. Certainly, a modern CNN should be able to detect these similarities and differences. Our goal is to show that a SNN can as well.

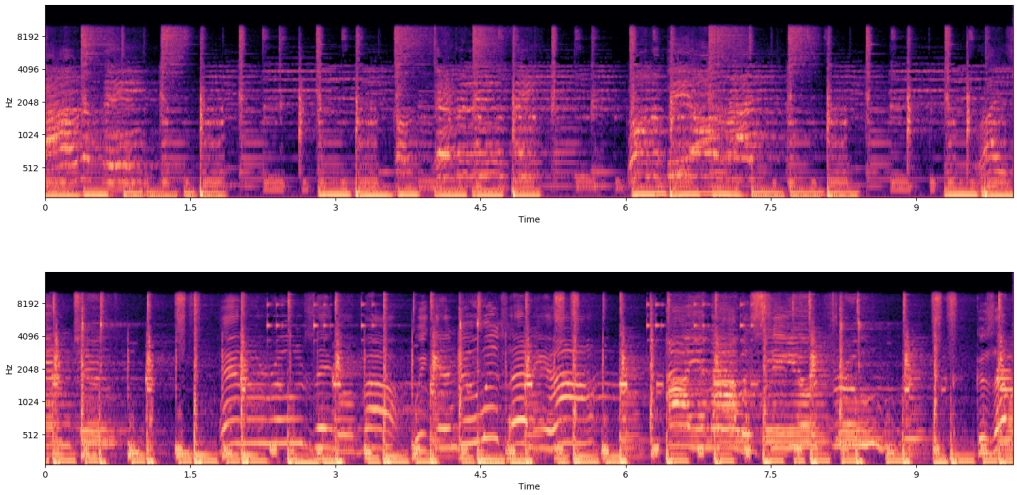

Figure 1 - Mel spectrograms of two different reggae tracks

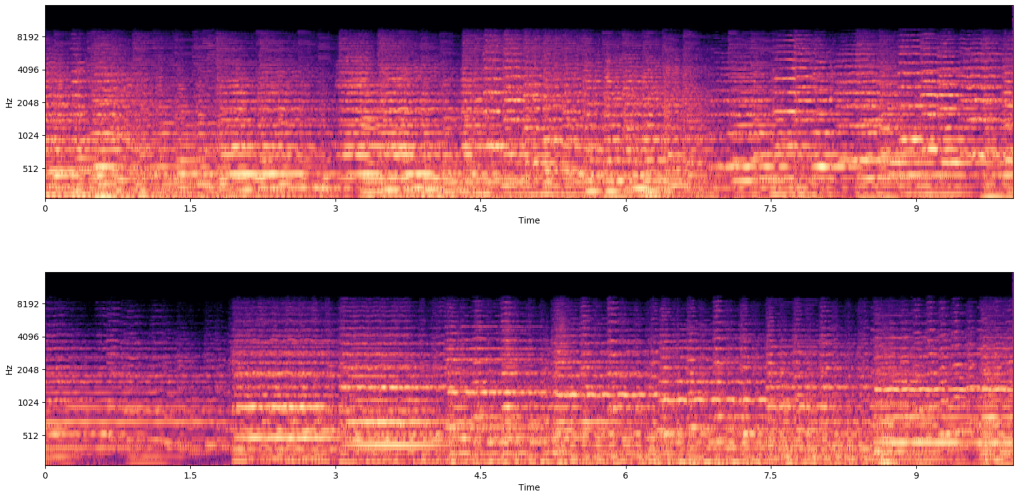

Figure 2 - Mel spectrograms of two different classical tracks

Since a musical track can be readily transformed into a spectrogram using well understood mathematical transforms (Fast Fourier, Discrete Cosine Transform, STFT, etc.), we can readily

determine whether or not spectrograms contain intrinsic properties on a per genre basis that can be classified using image recognition mechanisms. Previous work in this area has shown success in leveraging Mel spectrograms [2], MFCC's [4], and power spectrograms [5], for image classification using convolutional neural networks. Our goal was to determine if these image-based classification systems apply to a third-generation spiking neural network. We built a spiking neural network using Hebbian learning and Oja's Rule, and used the GTZAN database to both train and test the network. Based on our results, we show it is quite possible to use spectral imagery to extract textural features and classify music by genre using spiking neural networks.

2. Background (or Theory)

Music Information Retrieval (MIR) generally consists of extracting characteristics from audio that can be translated into useful data, whether its detecting which instruments are being used, whether there is a voice present, what tone or mood is being conveyed, or what genre a song fits into. This is a large field with many entities interested in progress, both in academia and commercially. It includes commercial entities like Spotify, but also companies that are interested in the development of machine hearing to improve workflows and create automatic control for consumer devices, such as in the areas of augmented and virtual reality (AR/VR).

The first step to successful MIR is feature extraction. A seminal work by Tzanetakis [1], encoded individual signals for a small window and derived features like the centroid, rolloff, flux, and Zero Crossing Rate to represent musical information in a specific time window. Limiting to small time windows was necessary in the past, due to limited processing power. The modern approach is to create spectrogram and mel-spectrograms for the time window of a song and extract features from the spectrogram. The spectrogram x axis represents time, y represents frequency, and amplitude, or changes in intensity, are represented by brightness or color gamut intensity. Genres that are obviously different, like rock and classical, produce very different spectrograms, which is why binary classification with NN for such varying types of genres produce results with at least 82% accuracy (and up to 100% in some cases). However, with genres that are intuitively closer together, like disco and pop, the results are less perfect.

When it comes to producing the spectrograms for these kinds of classifications, it's important to produce spectrograms that are precise in their exemplification, are a close representation to how the song is experienced by human listeners, and lose the least amount of information while also minimizing noise.

Key features that are looked at in the spectrogram representation are; the *spectral centroid*, which is average frequency the energy the spectrum takes), *spectral bandwidth;* which is a weighted standard deviation; *spectral contrast,* which is a function of the spectral peak and spectral valley at each time frame; *autocorrelation*, which is a comparison of a signal to a time-lagged version of itself (which is a useful indicator of patterns); and *spectral rolloff,* which is a frequency behind which a specified percentage of the energy lies. This jargon suggests domain experitise is necessary to make meaningful progress in MIR. However, image classification isn't as parochial a space, and is now widely employed and well understood, especially in 2nd generation CNNs.

The search for effective music information retrieval and classification schemes to detect music genre is being studied copiously in the machine learning and artificial intelligence research. While consistent agreement on what type of music information comprises a specific genre, there is enough agreement to identify a smaller subset of common, western genres, despite the plethora of

music genres around the world. Such work in active learning concentrates on a variety of approaches.

One interesting approach is to use textural structure in the images of spectrograms and Mel-spectrograms. With spectrograms, a visual representation of the spectrum of frequencies and intensity vary with time. On the other hand, mel–spectrograms, are based on scaled original spectrograms. By using frequencies in a logarithmic *Mel-Scale*, we are able to represent sound more naturally, and more accurately, in a way commensurate with how humans sense different sound frequencies as melodies (hence MELody scale). The mels, or buckets of melodies, are calculated using the following equation, where m is the number of mels, and f is the frequency of a segment or slice of a music track.

$$m = 2595 \log_{10}\left(1 + \frac{f}{700}\right)$$

When considering an audio signal, such as a .WAV file, spectrograms visually represent the critical data, such as bit rate, audio sample size and channel #, that would be most useful to a classification scheme. From generating the Mel-spectrograms, the data is then considered for extraction from the images. These are shown in Figures 1 and 2. There are also other interesting characteristics that can be derived from spectrograms, including *chromagrams*, and *mel-frequency cepstral coefficients*, as seen in Fig. 3.

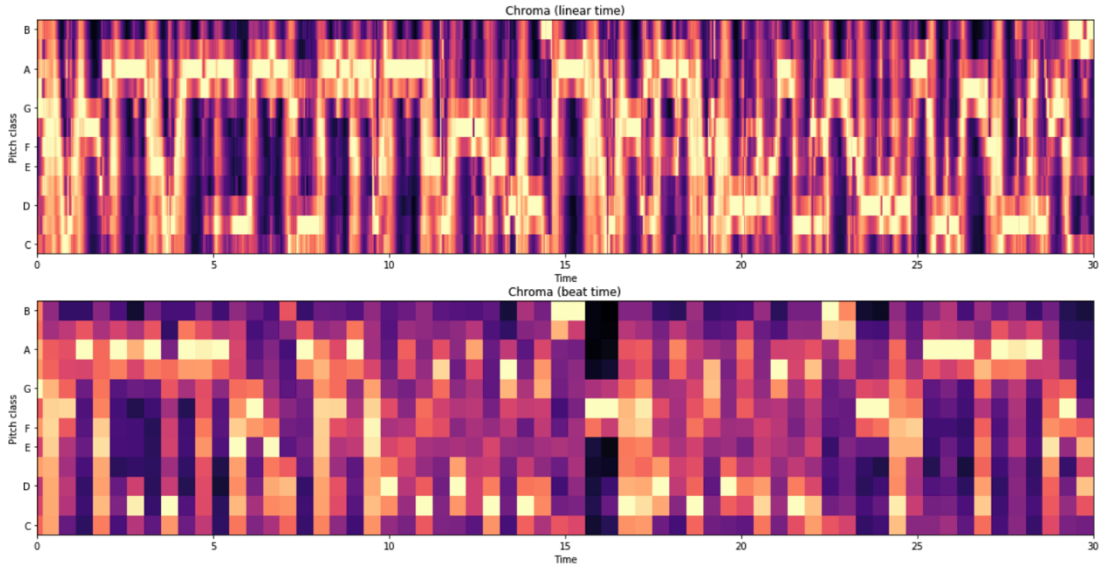

Figure 3 - Chroma and MFCC

Mel-spectrograms have shown to be highly useful in both speech and music classification systems, often surpassing standard FFT and STFT-based spectrographic representations. In [8], "The actual transformation used is shown to impact the classification accuracy, with Mel-scaled STFT outperforming the other discussed methods slightly and baseline MFCC features to a large degree." FFT is good; however, STFT is faster when the duration of data is smaller. The different transformations all do the same thing retrospectively, however they provide different efficiency results and work under different timescales.

TABLE I
MEDIAN AND MEDIAN ABSOLUTE DEVIATION OF ACCURACIES (%) FOR ESC-50 AND URBANSOUND8K

	Linear-STFT		Mel-STFT		CQT		CWT	MFCC
	wideband	*narrowband*	*wideband*	*narrowband*	*wideband*	*narrowband*		
ESC-50								
Conv-5: M×3	44.50±2.00	**46.62±2.25**	**46.25±2.00**	**48.00±1.63**	42.00±2.37	42.62±1.50	38.25±1.50	30.50±1.50
3×3	49.25±0.75	50.00±1.88	**50.87±2.50**	**53.75±1.75**	46.87±1.13	48.62±2.00	40.50±2.13	36.62±2.13
Conv-3: M×3	52.12±1.12	**55.12±1.88**	**56.37±1.63**	**56.25±1.75**	**54.37±2.25**	**53.50±1.87**	46.50±1.63	35.25±2.75
3×3	**55.00±1.37**	53.00±1.62	**54.00±1.25**	**55.00±1.63**	51.75±1.25	51.62±2.25	46.62±1.87	35.00±0.75
UrbanSound8K								
Conv-5: M×3	61.19±4.81	**63.44±3.39**	**62.22±5.19**	**64.97±3.69**	**62.87±3.25**	**63.12±3.25**	56.90±2.10	59.23±3.24
3×3	**67.94±4.22**	62.83±4.73	**69.59±4.19**	**65.31±2.19**	**69.25±4.69**	64.33±3.60	61.56±1.80	57.15±1.81
Conv-3: M×3	**68.81±4.50**	66.72±2.72	**70.69±4.06**	**68.29±3.00**	**70.94±4.06**	**67.06±3.12**	64.00±2.17	64.87±2.17
3×3	**70.94±2.94**	68.19±3.25	**74.66±3.39**	**71.25±1.85**	**73.03±3.56**	68.31±2.35	64.75±1.44	62.81±4.03

Although we've used two dimensional Numpy arrays, note that the intensity of the color (the brightness) indicates the amplitude, or power signature, of the spectrogram. 3-dimensional representations provide a better view of how amplitude and frequency vary over time based on genre, as shown in Figures 4 and 5.

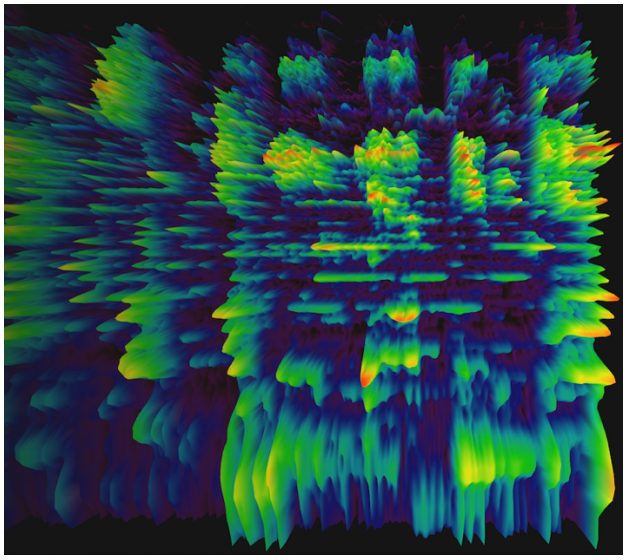

Figure 4 - Kenny Loggins: Return to Pooh Corner (Soft)

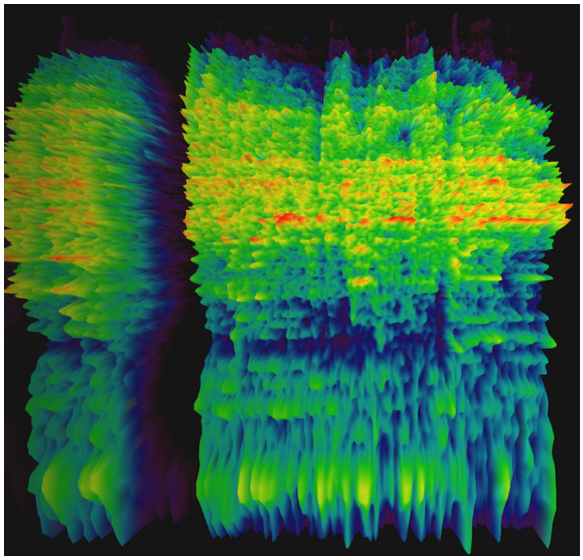

Figure 5 - Metallica: Master of Puppets (Metal)

Spiking Neural Networks

SNN's are third generation artificial neural networks which incorporate time into their operating model. It overtly cogitates the timing of inputs. However, such a network also has to be trained and often coached, or taught, in order to be fully functional. The most popular and best-known model of threshold fire neurons correlated with SNN's are both Integrate-and-fire (IF), and also Leaky integrate-and-fire (LIF), which is what we implemented.

Hebbian learning is a training method where when two neurons fire "close in time" then the strength of synaptic connection between them increases. Therefore, the weights reflect correlations between firing times. Therefore, if an input neuron frequently leads to the firing out the output neuron, the synapse is strengthened.

$$\Delta w_{ij}(t) = \eta v_i v_j g(t_{v_i}, t_{v_j})$$

Equation 1: Hebb Learning basic formula

However, such an algorithm does have its drawbacks, such as that it allows unbounded growth for synaptic weights, and that it fails to induce competition. Therefore, to fix such issues, we decided to also implement an extension of the training method called Oja's rule.

Oja's rule is a rule based on normalized weights. This extension provides a more general, and efficient way to stabilize weight update within Hebbian learning.

$$\frac{d}{dt} w_{ij} = \gamma(v_i v_j - w_{ij} v_i^2)$$

Equation 2: Oja's rule formula

To fix the issue of inducing competition, Oja's rule resolves this by squaring the weights within a sum.

$$\sum_j w_{ij}^2 = 1$$

Equation 3: Oja's rule to induce competition

The sum implies competition between the synapses that make connections to the same postsynaptic neuron, i.e., if some weights grow, others must decrease [3].

3. Experimental/Modeling Design

We built a spiking neural network with Leaky Integrate and Fire (LIF) neurons with 3 hidden layers according to the topology in Figure 3. The network was trained using Hebbian learning by applyimng Oja's rule.

In the SNN, we had an input layer that took various values in, with 138 neurons each that represented a single cell in a row of the values. Through experimentation, we found a threshold that would make a neuron fire which was optimal for training, by drowning out the lower frequencies shared across almost all genres, which can be seen in our spectrograms. Therefore, with both the training and rule implemented, we were able to normalize the weights, and also induce competition.

Making use of these learning rules, we constructed a Hebbian based Spiking Neural Network with an input layer, three hidden layers, and an output layer. The input layer contains 138 neurons, one for each cell in each row of our data csv, and the internal nodes contain various number of LIF neurons that worked when we experimented and changed the values around. Figure 6 shows our SNN.

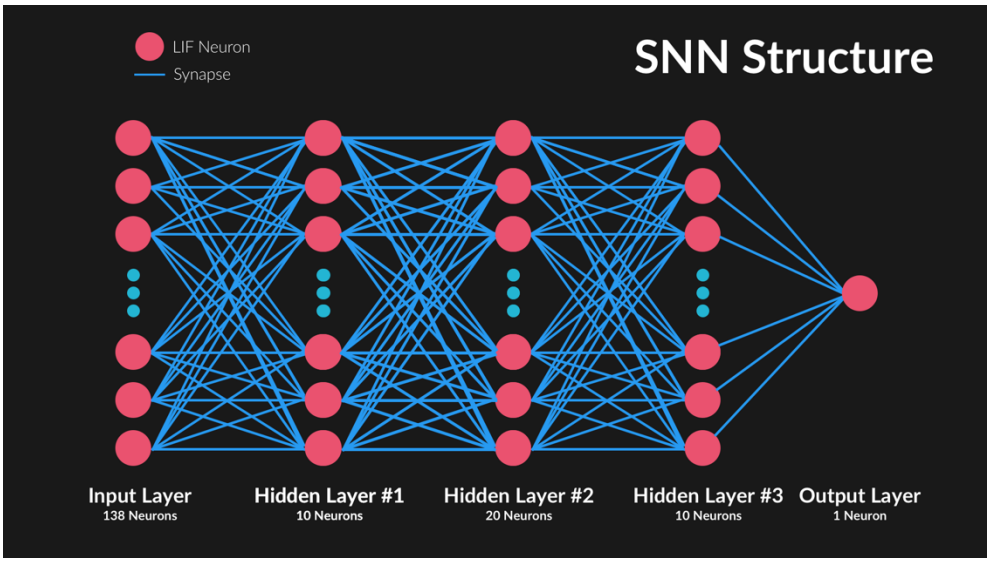

Figure 6 - Spiking Neural Network for Music Classification with 3 Hidden Layers

We initialized all the weights for the first two hidden layers in the range [-1,1], to allow the network to train specific neurons to be excitatory, or inhibitory. For the third and last hidden layer, we manually split up the first five neurons to be excitatory and the last five to be inhibitory, so that with enough training, specific neurons would get good at detecting the correct genre, causing the output layer to fire, and other neurons would get good at detecting the incorrect genre, and would attempt to prevent the output layer from firing.

For the actual data, we input various songs into a combined, labeled array that was then shuffled to reduce any chance of bias in our network. Using this list, we trained our network with Oja's rule, letting the data pass through all the hidden layers. Before we input the data from the last hidden layer to the output layer, we used the labels in each row to act as a teaching input for the output layer, to force it to fire when specific rows were of the correct genre, and prevent it from firing for incorrect genres. Using this, if any neuron in the hidden layer fired with the output layer, the weight was increased. This ensured that the neurons that were correctly detecting the genres would be prioritized, since competition was also introduced through Oja's rule.

Using this design, we were able to create a network that successfully detected the correct genre. Since we used a lot of data points to train our network (around a million rows total), and the classification task is binary, the tree is over-trained to detect specific frequencies, and would face trouble while trying to correctly classify songs from similar genres, like Pop and Disco, but for detection between genres that are complete different, like metal and classical, the network works extremely well.

We downloaded the GTZAN database, which contains 9 genres of music, each with 100, 30 second song clips. We first converted the music files from .au to .wav format using Mac OS X's `afconvert` utility. We leveraged the `librosa` python library to read in the .wav files, down –sample them from 48kHz to 11kHz audio (to minimize training load), extract features from the .wav files through Fourier transforms, and created the spectrograms, mel-spectrograms, and mel-

frequency cepstral coefficients using matplotlib. The mel-spectrograms were parameterized as follows:

Number of Fast Fourier Transforms	1024
Hop Length	1/200
Number of Mels	138
Minimum Frequency (Hz)	16000
Lag (ms)	2
Maximum size (MB)	3

Using `sckit-image`, we extracted textural information from the resulting images, including Local Binary Pattern and Rotation Invariant LBP [8]. We then took the spectral data from the Numpy arrays resultant from the FFTs. These arrays are then reshaped into 138x2005 dimensional vectors, which are fed into the neural network.

4. Results and Discussion

Within our SNN, we decided to have 1 input layer, 3 hidden layers, and 1 output layer. We randomized the inputs between the genres we offer to the SNN. Input 1 denotes Metal and 0 denotes Classical.

Using the structure denoted in the previous sections, our SNN was fed 1000 rows of CSV data, 500 from both genres, to test the accuracy of classifying between metal and classical music. As an output from our SNN, we received:

```
Ouput firing rate:  0.6  for genre  1.0
Ouput firing rate:  0.0  for genre  0.0
Ouput firing rate:  0.0  for genre  0.0
Ouput firing rate:  0.0  for genre  0.0
Ouput firing rate:  0.0  for genre  0.0
Ouput firing rate:  0.0  for genre  1.0
Ouput firing rate:  0.1  for genre  1.0
Ouput firing rate:  0.0  for genre  0.0
Ouput firing rate:  0.0  for genre  0.0
Ouput firing rate:  0.0  for genre  0.0
Ouput firing rate:  0.30000000000000004  for genre  1.0
Ouput firing rate:  0.1  for genre  0.0
Ouput firing rate:  0.7000000000000001  for genre  1.0
Ouput firing rate:  0.1  for genre  1.0
Ouput firing rate:  0.5  for genre  1.0
Correctly Classified:  812
IncorrectlyClassified:  188
```

In this output, genre 1.0 represents lines that were metal, and genre 0.0 represents lines that are classical, and there are a thousand "output firing rate: " lines present. Given that input, we were able to correctly classify 812 rows of the CSV as the correct genre, while incorrectly classifying 188 rows. This gives us a correct classification of **81.2%**, which is relatively high for the intricacies that are present in music that define what a song's genre is. This 81.2% figure is extremely similar to the figures given in other classification tests performed, and accurately represents our network.

Some reasons for why the classification accuracy might still be relatively low has to do with the fact that we are doing very naïve feature detection from the mel-spectrograms, as we are using nothing but the volume of specific frequencies in music to try to detect genre of music, and as made evident in the various papers we read, detecting genre using just this data doesn't account for the nuances present in music. Also, our network is relatively shallow, with only 40 total hidden neurons, and only 3 relatively small layers. Given this, if we were to use more data, a deeper network, and better feature detection with potentially higher resolution mel-spectograms, the neural network could definitely be improved to get closer to a 100% accuracy, but we are extremely proud of the 81.2% accuracy we were able to accomplish within a few short weeks.

We attempted a more naïve approach by just passing the entire RGP pixel map into the network. After training the network, our classification scheme was 100% accurate in detecting the two genres, but we believe that, because we used the entire spectral map and because the genres are so similar in-class and so dissimilar between classes, we over-fit the data. It is likely that more similar classes would not be as accurate in classification due to overfitting. These results are shown below.

```
Ouput firing rate:  3.9  for genre  0.0
Ouput firing rate:  3.9  for genre  0.0
Ouput firing rate:  4.1  for genre  0.0
Ouput firing rate:  5.500000000000001  for genre  1.0
Ouput firing rate:  5.500000000000001  for genre  1.0
Ouput firing rate:  5.500000000000001  for genre  1.0
Ouput firing rate:  4.1  for genre  0.0
Ouput firing rate:  5.500000000000001  for genre  1.0
Ouput firing rate:  5.500000000000001  for genre  1.0
Ouput firing rate:  4.0  for genre  0.0
```

The line of separation in each of the 10 training samples is distinct. This is true each of the 4 experiments with different input files.

```
Ouput firing rate:  1.6  for genre  0.0
Ouput firing rate:  4.6000000000000005  for genre  1.0
Ouput firing rate:  1.8  for genre  0.0
Ouput firing rate:  1.7  for genre  0.0
Ouput firing rate:  4.6000000000000005  for genre  1.0
Ouput firing rate:  4.5  for genre  1.0
Ouput firing rate:  4.4  for genre  1.0
Ouput firing rate:  4.5  for genre  1.0
Ouput firing rate:  1.6  for genre  0.0
Ouput firing rate:  1.8  for genre  0.0
```

```
Ouput firing rate:  0.2  for genre  1.0
Ouput firing rate:  0.0  for genre  0.0
Ouput firing rate:  0.0  for genre  0.0
Ouput firing rate:  0.2  for genre  1.0
Ouput firing rate:  0.2  for genre  1.0
Ouput firing rate:  0.2  for genre  1.0
Ouput firing rate:  0.0  for genre  0.0
Ouput firing rate:  0.2  for genre  1.0
Ouput firing rate:  0.0  for genre  0.0
Ouput firing rate:  0.0  for genre  0.0


Ouput firing rate:  0.4  for genre  0.0
Ouput firing rate:  1.5  for genre  1.0
Ouput firing rate:  1.7000000000000002  for genre  1.0
Ouput firing rate:  1.5  for genre  1.0
Ouput firing rate:  0.30000000000000004  for genre  0.0
Ouput firing rate:  0.30000000000000004  for genre  0.0
Ouput firing rate:  1.5  for genre  1.0
Ouput firing rate:  1.5  for genre  1.0
Ouput firing rate:  0.4  for genre  0.0
Ouput firing rate:  0.4  for genre  0.0
```

5. Conclusions

Our results show that for genres that are very acoustically dissimilar (Metal and Classical), we can achieve genre classification accuracy of near 81.2% accuracy using a biologically inspired spiking neural network based on Hebbian learning with Oja's rule. For more similar genres (Disco and Pop), accuracy falls to less than 80% on the GTZAN dataset. Part of this is due to known problems with the GTZAN dataset (duplications, mislabeling), part of this is due to the lack of deep textural data in mel-spectrograms alone, and part of it is because our network is relatively slow to train given that is implemented in Python, an interpreted language. A compiled implementation or an implementation accelerated on some form of neuromorphic hardware (e.g. SpiNNaker) would have allowed us to use larger, more information-rich spectrographic data that may have helped us achieve better training accuracy.

Acknowledgments

We'd like to thank Prof. Konstantinos Michmizos for bringing to life a magnificent area of study in CS443/674, as well as for his help on guiding our research project and providing the outline and structure of the paper. We also drew inspiration from many of the pioneers in the fields or Pattern Recognition and Deep Learning, including Andrew Ng of Coursera fame, whose YouTube videos taught us how Neural Networks work. Finally, we'd like to thank Guangzhi Tang and Jeff Ames for their helpful suggestions and guidance on the project.

References

1. Tzanetakis G, Essl G, Cook P (2001) *Audio analysis using the discrete wavelet transform*. In: Proceedings of WSES international conference, acoustics and music: theory and applications (AMTA), Skiathos, Greece
2. Guo G, Li SZ (2003) *Content-based audio classification and retrieval by support vector machines*. IEEE Trans Neural Netw 14(1):209–215 Kailath T (1974) A view of three decades of linear filtering theory. IEEE Trans Inf Theory 20(2):146–181
3. Logan B (2000) *Mel frequency cepstral coefficients for music modeling*. In: Proceedings of the international symposium on music information retrieval (SMIR)
4. Markel JD, Gray A (1976) *Linear prediction of speech*. Communication & Cybernetics. Springer, Heidelberg McGarry KJ, Wermter S, McIntyre J (1999)
5. *Knowledge extraction from radial basis function networks and multi-layer perceptrons*. In: Proceedings of international joint conference on neural networks (IJCNN), Washinton, vol 4, pp 2494–2497
6. A. Krizhevsky, I. Sutskever, G.E. Hinton, *ImageNet classification with deep convolutional neural networks*, in: Advances in Neural Information Processing Systems, 2012, pp. 1097–1105.
7. Y. LeCun, Y. Bengio, G.E. Hinton, *Deep learning*, Nature 521 (2015) 436–444.
8. T. Lidy, C.N. Silla Jr., O. Cornelis, F. Gouyon, A. Rauber, C.A.A. Kaestner, A.L. Koerich, *On the suitability of state-of-the-art music information retrieval methods for analyzing, categorizing and accessing non-western and ethnic music collections*, Signal Process. 90 (4) (2010) 1032–1048.
9. Y.M.G. Costa, L.S. Oliveira, A.L. Koerich, F. Gouyon, *Music genre recognition based on visual features with dynamic ensemble of classifiers selection*, in: International Conference on Systems, Signals and Image Processing, 2013, pp. 55–58.
10. T. Feng, *Deep learning for music genre classification*. Technical report, University of Illinois, 2014
11. Y.M.G. Costa, L.S. Oliveira, A.L. Koerich, F. Gouyon, *Music genre recognition using spectrograms*, in: International Conference on Systems, Signals and Image Processing, 2011, pp. 154–161.

Appendix A – Source Code

LIF Neuron Code

```
1.  import numpy as np
2.  import matplotlib.pyplot as plt
3.  from scipy import integrate
4.  from pylab import *
```

```
from random import *
import math

class Neuron:
    def __init__(self, durationForSimulation, numInputs):
        self.T = durationForSimulation #Time to simulate, in ms
        self.dT = 0.1 #the dT time step in dV/dT
        self.VArray = zeros(int((self.T/self.dT) + 1)) #Array of membrane potentials, for plotting later

        self.Vt = 1 #Threshold, in V
        self.Vr = 0 #Reset potential, in mV
        self.initialV = 0 #Initial Membrane Potential = Formula is change in mV
        self.R = 1 #Membrane resistance, in kOhms
        self.C = 10 #Capacitance in uF
        self.tauM = self.R*self.C #Membrane time constant, in miliseconds
        self.firingRate = 10

        self.currentChangeInPotential = float(10.0) #Change in current membrane potential - Used in array

        self.numFired = 0
        self.notFired = 0
        self.fired = 0
        self.spikeRateForData = []
        self.totalSpikingRate = 0

        self.classificationRate = 0
        self.classificationActivity = 0

        self.weights = [];
        for i in range(numInputs):
            randomWeight = math.ceil(uniform(0,2000)-1000)/1000
            self.weights.append(randomWeight)

    def runNeuron(self, inputCurrent):
        self.counter = 0.0
        I = inputCurrent #input current, in Amps - Given by parameter, plus minimum threshold to actually fire

        spikeSum = 0.
        self.VArray = zeros(int((self.T/self.dT) + 1)) #Array of membrane potentials, for plotting later

        self.currentChangeInPotential = 0

        for i in range(1, len(self.VArray)) :
            self.currentChangeInPotential = (-1*self.VArray[i-1] + self.R*I)
            self.VArray[i] = self.VArray[i-1] + self.currentChangeInPotential/self.tauM*self.dT
            if(self.VArray[i] >= self.Vt):
                self.VArray[i] = self.Vr
                spikeSum += 1

        return (math.ceil((spikeSum/self.T)*1000))/1000

    def showGraph(self):
        currTime = arange(0, self.T+self.dT, self.dT)
        plt.plot(currTime, self.VArray)
        plt.xlabel('Time in miliseconds')
        plt.ylabel('Membrane Potential in Volts')
        plt.title('Simulation - LIF Neuron')
        plt.ylim([0,4])
        plt.show()
```

```
        print(spikeSum/self.T)

time = 20
sampleNeuronRun = Neuron(time, 5)
print("", 1.4, ": ", sampleNeuronRun.runNeuron(5))
# sampleNeuronRun = Neuron(time)
# print("", 1.4, ": ", sampleNeuronRun.RunNeuron(1.26))
# sampleNeuronRun = Neuron(time)
# print("", 1.4, ": ", sampleNeuronRun.RunNeuron(1.27))
# sampleNeuronRun = Neuron(time)
# print("", 1.5, ": ", sampleNeuronRun.RunNeuron(1.28))
# sampleNeuronRun = Neuron(time)
# print("", 1.5, ": ", sampleNeuronRun.RunNeuron(1.29))
# sampleNeuronRun = Neuron(time)
# print("", 1.5, ": ", sampleNeuronRun.RunNeuron(1.31))
```

Spiking Neural Network with Hebbian Learning Source Code

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import integrate
from pylab import *
from FinalProjectNeuron import Neuron
import math
from sklearn.preprocessing import normalize

class GenreClassifier:
    def __init__(self):
        self.timeStep = 2
        self.learningRate = 0.001
        self.spikingThreshold = 0.8         #Found through testing my specific neurons
        self.inputLayerSize = 138
        self.numSeconds = 2005                      #Number of input neurons/pixels
        self.timeThreshold = 10                 #Time to simulate neuron for

        self.classifications = 2
        self.hiddenLayerNum = 3
        self.neuronPerLayer = [10, 20, 10]

        self.dataList = []
        self.isFirst = 0

        self.inputLayer = []
        for i in range(self.inputLayerSize):
            self.inputLayer.append(Neuron(self.timeThreshold,0))

        self.middleLayer = []
        currNumInputs = 138
        for i in range(self.hiddenLayerNum):
            currLayer = []
            for j in range(self.neuronPerLayer[i]):
                currLayer.append(Neuron(self.timeThreshold, currNumInputs))
            self.middleLayer.append(currLayer)
            currNumInputs = self.neuronPerLayer[i]

        self.outputLayer = Neuron(self.timeThreshold, 0)

        weights = []
        for i in range(math.floor(currNumInputs/2)):
```

```
42.             self.outputLayer.weights.append(math.ceil(uniform(0,1000))/1000)
43.         for i in range(math.floor(currNumInputs/2), currNumInputs):
44.             self.outputLayer.weights.append(math.ceil(uniform(0,1000))/1000)
45.
46.
47.     def getMetalFiles(self):
48.         self.metalFiles = []
49.         for i in range(5):
50.             currName = "metal specs/metal.0000"
51.             currName = currName + str(i)
52.             currName += ".au.wav.csv"
53.             self.metalFiles.append(currName)
54.
55.         # for i in range(10,100):
56.         #   currName = "metal specs/metal.000"
57.         #   currName += str(i)
58.         #   currName += ".au.wav.csv"
59.         #   self.metalFiles.append(currName)
60.
61.         for j in range(5):
62.             array = np.genfromtxt(self.metalFiles[j], delimiter=',')
63.             array = np.transpose(array)
64.             array = array[0:100,:]
65.
66.             labeledArray = []
67.
68.             for i in range(array.shape[0]):
69.                 labeledArray.append(np.append(array[i], 1))
70.
71.             labeledArray = np.array(labeledArray)
72.             if(self.isFirst == 0):
73.                 self.dataList = labeledArray
74.                 self.dataList = np.array(self.dataList)
75.                 self.isFirst = 1
76.             else:
77.                 self.dataList = np.concatenate((self.dataList, labeledArray), axis=0)
78.
79.     def getClassificationMetalInput(self):
80.         metalFiles = []
81.         dataList = []
82.         for i in range(10,15):
83.             currName = "metal specs/metal.000"
84.             currName = currName + str(i)
85.             currName += ".au.wav.csv"
86.             metalFiles.append(currName)
87.
88.         for j in range(5):
89.             array = np.genfromtxt(metalFiles[j], delimiter=',')
90.             array = np.transpose(array)
91.             array = array[0:100,:]
92.
93.             labeledArray = []
94.
95.             for i in range(array.shape[0]):
96.                 labeledArray.append(np.append(array[i], 1))
97.
98.             labeledArray = np.array(labeledArray)
99.             dataList.append(labeledArray)
100.
101.         return dataList
102.
```

```
    def getClassificationClassicalInput(self):
        classicalFiles = []
        dataList = []
        for i in range(10,15):
            currName = "classical specs/classical.000"
            currName = currName + str(i)
            currName += ".au.wav.csv"
            classicalFiles.append(currName)

        for j in range(5):
            array = np.genfromtxt(classicalFiles[j], delimiter=',')
            array = np.transpose(array)
            array = array[0:100,:]

            labeledArray = []

            for i in range(array.shape[0]):
                labeledArray.append(np.append(array[i], 0))

            labeledArray = np.array(labeledArray)
            dataList.append(labeledArray)

        return dataList


    def getClassicalFiles(self):
        self.classicalFiles = []
        for i in range(5):
            currName = "classical specs/classical.0000"
            currName += str(i)
            currName += ".au.wav.csv"
            self.classicalFiles.append(currName)

        # for i in range(10,100):
        #   currName = "classical specs/classical.000"
        #   currName += str(i)
        #   currName += ".au.wav.csv"
        #   self.classicalFiles.append(currName)

        for j in range(5):
            array = np.genfromtxt(self.classicalFiles[j], delimiter=',')
            array = np.transpose(array)
            array = array[0:100,:]

            labeledArray = []

            for i in range(array.shape[0]):
                labeledArray.append(np.append(array[i], 0))

            labeledArray = np.array(labeledArray)
            if(self.isFirst == 0):
                self.dataList = labeledArray
                self.dataList = np.array(self.dataList)
                self.isFirst = 1
            else:
                self.dataList = np.concatenate((self.dataList, labeledArray), axis=0)

    def train(self):
        input = self.dataList
        numFired = 0
        numNotFired = 0
```

```
        avgSpikeRate = 0
        for k in range(len(self.inputLayer)):
            neuron = self.inputLayer[k]
            currentSpikeRate = 0
            totalSpikeRate = 0
            numIncreased = 0
            numDecreased = 0

            currentSpikeRate = 0
            for i in range(len(input)):
                currentSpikeRate = neuron.runNeuron(input[i][k]*15+7.9)
                neuron.spikeRateForData.append(currentSpikeRate)
            for i in range(len(input)):
                if(currentSpikeRate >= self.spikingThreshold):
                    neuron.numFired += 1
                    numIncreased+=1
                elif (currentSpikeRate < self.spikingThreshold):
                    neuron.notFired += 1
                    numDecreased += 1

            if(neuron.numFired > neuron.notFired):
                print("Fired! ")
                numFired+=1
                neuron.fired = 1
            else:
                print("Not Fired for input: ", )
                numNotFired += 1                            #Store current spike rate in array for train
    ing next
        print(neuron.weights,"\nNum fired: ",numFired, " Num not fired:  ", numNotFired)
        print("Average Spike Rate: ", avgSpikeRate, " ", avgSpikeRate/len(input))

        print("\n\n\n\n\n")

        for k in range(len(self.middleLayer[0])):
            neuron = self.middleLayer[0][k]
            currentSpikeRate = 0
            totalSpikeRate = 0
            numFired = 0

            for i in range(len(input)):
                totalSpikeRate += currentSpikeRate
                # preSpikeRate =
                currentSpikeRate = 0
                for j in range(len(input[0])-1):
                    multiplier = 1
                    if(input[i][138] == 0):
                        multiplier *= 0.7
                    currentSpikeRate += neuron.runNeuron(multiplier*(self.inputLayer[j].spikeRateForData
    [i])*neuron.weights[j]*2.2)
                    neuron.spikeRateForData.append(currentSpikeRate)
    #Store current spike rate in array for training next
                print("Curr spike rate: ", currentSpikeRate)
                for j in range(len(input[0])-1):
                    if(currentSpikeRate >= self.spikingThreshold and self.inputLayer[j].fired == 1):
                 #If both fire, increase weight
                        currWeight = neuron.weights[j]
                        deltaW = (self.learningRate * 1 * (1 - 1*currWeight))/self.timeStep
                        if(currWeight+deltaW <= 1 and currWeight+deltaW>-1):
                            neuron.weights[j] += deltaW
                            # neuron.weights[j] = round(neuron.weights[j])
                        elif(currWeight+deltaW == 1):
```

```
                            neuron.weights[j] = 1.000
                        # print("increased weight from ", currWeight, " to ", neuron.weights[j], " with delta ", (deltaW), " for input ", " ",numFired)
                        numIncreased += 1
                        neuron.numFired += 1
                    elif (currentSpikeRate < self.spikingThreshold - 0.1 and self.inputLayer[j].fired == 1):         #if pre fires and post doesnt, decrease weight
                        currWeight = neuron.weights[j]
                        deltaW = (self.learningRate * 1 * (1 - 1*currWeight))/self.timeStep
                        if(currWeight+deltaW >= -1):
                            neuron.weights[j] -= deltaW
                            # neuron.weights[j] = round(neuron.weights[j])
                        elif(currWeight+deltaW == -1):
                            neuron.weights[j] = -1.000
                        neuron.notFired += 1
                        # print("decreased weight from ", currWeight, " to ", neuron.weights[j], " with delta ", (-1*deltaW), " for input ", " ",numFired)
                        numDecreased += 1
                if(neuron.numFired > neuron.notFired):
                    neuron.fired = 1
                    print("Fired numFired:", neuron.numFired, " notFired: ", neuron.notFired)
                neuron.spikeRateForData.append(currentSpikeRate)                                    #Store current spike rate in array for training next
                # print(i,"\nNum increased: ",numIncreased, " Num Decreased:  ", numDecreased)
                # for j in range(len(input[0])):
            print(neuron.weights,"\nNum increased: ",numIncreased, " Num Decreased:  ", numDecreased, " for input ")
            neuron.totalSpikeRate = totalSpikeRate/4
        print("\n\n\n\n\n")
        for k in range(len(self.middleLayer[1])):
            neuron = self.middleLayer[1][k]
            currentSpikeRate = 0
            totalSpikeRate = 0
            numFired = 0

            for i in range(len(input)):
                totalSpikeRate += currentSpikeRate
                # preSpikeRate =
                currentSpikeRate = 0
                for j in range(len(self.middleLayer[0])):
                    multiplier = 1
                    if(input[i][138] == 0):
                        multiplier *= 0.8
                    currentSpikeRate += neuron.runNeuron(multiplier*(self.middleLayer[0][j].spikeRateForData[i])*neuron.weights[j]*1.55)
                # print("Curr spike rate: ", currentSpikeRate)
                for j in range(len(self.middleLayer[0])):
                    if(currentSpikeRate >= self.spikingThreshold and self.inputLayer[j].fired == 1):
                #If both fire, increase weight
                        currWeight = neuron.weights[j]
                        deltaW = (self.learningRate * 1 * (1 - 1*currWeight))/self.timeStep
                        if(currWeight+deltaW <= 1 and currWeight+deltaW>-1):
                            neuron.weights[j] += deltaW
                            # neuron.weights[j] = round(neuron.weights[j])
                        elif(currWeight+deltaW == 1):
                            neuron.weights[j] = 1.000
                        # print("increased weight from ", currWeight, " to ", neuron.weights[j], " with delta ", (deltaW), " for input ", " ",numFired)
                        numIncreased += 1
                        neuron.numFired+=1
```

```
                    elif (currentSpikeRate < self.spikingThreshold - 0.1 and self.inputLayer[j].fired ==
    1):         #if pre fires and post doesnt, decrease weight
                        currWeight = neuron.weights[j]
                        deltaW = (self.learningRate * 1 * (1 - 1*currWeight))/self.timeStep
                        if(currWeight+deltaW >= -1):
                            neuron.weights[j] -= deltaW
                            # neuron.weights[j] = round(neuron.weights[j])
                        elif(currWeight+deltaW == -1):
                            neuron.weights[j] = -1.000
                        neuron.notFired += 1
                        # print("decreased weight from ", currWeight, " to ", neuron.weights[j], " with
    delta ", (-1*deltaW), " for input ", " ",numFired)
                        numDecreased += 1
                if(neuron.numFired > neuron.notFired):
                    neuron.fired = 1
                    print("Fired numFired:", neuron.numFired, " notFired: ", neuron.notFired)
                neuron.spikeRateForData.append(currentSpikeRate)                                    #Sto
    re current spike rate in array for training next
                # print(i,"\nNum increased: ",numIncreased, " Num Decreased:  ", numDecreased)
                # for j in range(len(input[0])):
            # print(neuron.weights,"\nNum increased: ",numIncreased, " Num Decreased:  ", numDecreased,
    " for input ")
            neuron.totalSpikeRate = totalSpikeRate/4
        print("\n\n\n\n\n")
        for k in range(len(self.middleLayer[2])):
            neuron = self.middleLayer[2][k]
            currentSpikeRate = 0
            totalSpikeRate = 0
            numFired = 0

            for i in range(len(input)):
                totalSpikeRate += currentSpikeRate
                # preSpikeRate =
                currentSpikeRate = 0
                for j in range(len(self.middleLayer[1])):
                    multiplier = 1
                    if(input[i][138] == 0):
                        multiplier *= 0.8
                    currentSpikeRate += neuron.runNeuron(multiplier*(self.middleLayer[1][j].spikeRateFor
    Data[i])*neuron.weights[j]*1.55)
                # print("Curr spike rate: ", currentSpikeRate)
                for j in range(len(self.middleLayer[1])):
                    if(currentSpikeRate >= self.spikingThreshold and self.inputLayer[j].fired == 1):
                #If both fire, increase weight
                        currWeight = neuron.weights[j]
                        deltaW = (self.learningRate * 1 * (1 - 1*currWeight))/self.timeStep
                        if(currWeight+deltaW <= 1 and currWeight+deltaW>-1):
                            neuron.weights[j] += deltaW
                            # neuron.weights[j] = round(neuron.weights[j])
                        elif(currWeight+deltaW == 1):
                            neuron.weights[j] = 1.000
                        # print("increased weight from ", currWeight, " to ", neuron.weights[j], " with
    delta ", (deltaW), " for input ", " ",numFired)
                        numIncreased += 1
                        neuron.numFired+=1
                    elif (currentSpikeRate < self.spikingThreshold - 0.1 and self.inputLayer[j].fired ==
     1):         #if pre fires and post doesnt, decrease weight
                        currWeight = neuron.weights[j]
                        deltaW = (self.learningRate * 1 * (1 - 1*currWeight))/self.timeStep
                        if(currWeight+deltaW >= -1):
                            neuron.weights[j] -= deltaW
```

```
                            # neuron.weights[j] = round(neuron.weights[j])
                        elif(currWeight+deltaW == -1):
                            neuron.weights[j] = -1.000
                        neuron.notFired += 1
                        # print("decreased weight from ", currWeight, " to ", neuron.weights[j], " with delta ", (-1*deltaW), " for input ", " ",numFired)
                        numDecreased += 1
                if(neuron.numFired > neuron.notFired):
                    neuron.fired = 1
                    print("Fired numFired:", neuron.numFired, " notFired: ", neuron.notFired)
                neuron.spikeRateForData.append(currentSpikeRate)                                    #Store current spike rate in array for training next
                # print(i,"\nNum increased: ",numIncreased, " Num Decreased:  ", numDecreased)
                # for j in range(len(input[0])):
            # print(neuron.weights,"\nNum increased: ",numIncreased, " Num Decreased:  ", numDecreased, " for input ")
            neuron.totalSpikeRate = totalSpikeRate/4
            # print("\n",totalSpikeRate/4,"\n")

        self.trainExcitatoryNeurons(input)
        self.trainInhibitoryNeurons(input)

    def saveWeights(self):
        layer1 = []
        for i in range(len(self.middleLayer[0])):
            currArray = []
            for j in range(len(self.middleLayer[0][i].weights)):
                currArray.append(self.middleLayer[0][i].weights[j])
            layer1.append(currArray)

        layer1 = np.array(layer1)

        np.savetxt('layer1.csv', layer1, delimiter=",")

        layer2 = []
        for i in range(len(self.middleLayer[1])):
            currArray = []
            for j in range(len(self.middleLayer[1][i].weights)):
                currArray.append(self.middleLayer[1][i].weights[j])
            layer2.append(currArray)

        layer2 = np.array(layer2)
        np.savetxt('layer2.csv', layer2, delimiter=",")

        layer3 = []
        for i in range(len(self.middleLayer[2])):
            currArray = []
            for j in range(len(self.middleLayer[2][i].weights)):
                currArray.append(self.middleLayer[2][i].weights[j])
            layer3.append(currArray)

        layer3 = np.array(layer3)
        np.savetxt('layer3.csv', layer3, delimiter=",")

        outputLayer = []
        for i in range(len(self.outputLayer.weights)):
            outputLayer.append(self.outputLayer.weights[i])

        outputLayer = np.array(outputLayer)
        np.savetxt('outputLayer.csv', outputLayer, delimiter=",")

```

```
    def getWeights(self):
        layer1 = genfromtxt('layer1.csv', delimiter=',')
        for i in range(len(self.middleLayer[0])):
            for j in range(138):
                self.middleLayer[0][i].weights[j] = layer1[i][j]

        layer2 = genfromtxt('layer2.csv', delimiter=',')
        for i in range(len(self.middleLayer[1])):
            for j in range(len(self.middleLayer[1][0].weights)):
                self.middleLayer[1][i].weights[j] = layer2[i][j]

        layer3 = genfromtxt('layer3.csv', delimiter=',')
        for i in range(len(self.middleLayer[2])):
            for j in range(len(self.middleLayer[2][0].weights)):
                self.middleLayer[2][i].weights[j] = layer3[i][j]

        outputLayer = genfromtxt('outputLayer.csv', delimiter=',')
        for i in range(len(self.outputLayer.weights)):
            self.outputLayer.weights[i] = outputLayer[i]

    def trainExcitatoryNeurons(self, input):
        for k in range(int(math.floor(len(self.outputLayer.weights)/2))):
            # print("Classification rate: ",self.middleLayer[k].spikeRateForData)
            currentSpikeRate = 0
            totalSpikeRate = 0
            numFired = 0

            for i in range(len(input)):
                preSpikeRate = self.middleLayer[2][k].spikeRateForData[i]
                preActivity = 1 if preSpikeRate >= self.spikingThreshold else 0
                currWeight = self.outputLayer.weights[k]

                # print("\nCurrSpikeRate: ",currSpikeRate, " preSpikeRate ", preSpikeRate)
                if(self.dataList[i][138] == 1):
                    currSpikeRate = self.outputLayer.runNeuron(50)
                else:
                    currSpikeRate = (self.outputLayer.runNeuron(preActivity*currWeight*20))
                # print("CurrSpikeRate: ",currSpikeRate, " preSpikeRate ", preSpikeRate)
                if preSpikeRate >= self.spikingThreshold and (self.dataList[i][138] == 1):
                    currWeight = self.outputLayer.weights[k]
                    deltaW = (self.learningRate * 1 * (1 - 1*currWeight))/self.timeStep
                    if (self.dataList[i][138] == 1):
                        deltaW = math.fabs(deltaW)*2
                    if(currWeight+deltaW <=1):
                        self.outputLayer.weights[k] += deltaW
                        self.outputLayer.weights[k] = round(self.outputLayer.weights[k])
                    else:
                        self.outputLayer.weights[k] = 1.000
                    # print("increased weight from ", currWeight, " to ", self.outputLayer.weights[k], "
    with delta ", round(deltaW), " for input ", input[i], " ")
                elif preSpikeRate >= self.spikingThreshold and self.dataList[i][138] == 0:
                    currWeight = self.outputLayer.weights[k]
                    deltaW = (self.learningRate * 1 * (1 - 1*currWeight))/self.timeStep
                    if(currWeight-deltaW >=-1):
                        self.outputLayer.weights[k] -= deltaW
                        self.outputLayer.weights[k] = round(self.outputLayer.weights[k])
                    else:
                        neuron.weights[j] = -1.000
                    # print("decreased weight from ", currWeight, " to ", self.outputLayer.weights[k], "
    with delta ", round(deltaW), " for input ", input[i], " ")
                print("Weight for excitatory output ", input[i][138], " = ", self.outputLayer.weights)
```

```
443.
444.     def trainInhibitoryNeurons(self, input):
445.         for k in range(int(math.floor(len(self.outputLayer.weights)/2)), self.hiddenLayerNum):
446.             currentSpikeRate = 0
447.             totalSpikeRate = 0
448.             numFired = 0
449.             for i in range(len(input)):
450.                 preSpikeRate = self.middleLayer[2][k].spikeRateForData[i]
451.                 preActivity = 1 if preSpikeRate >= self.spikingThreshold else 0
452.                 currWeight = self.outputLayer.weights[k]
453.
454.                 currSpikeRate += (self.outputLayer.runNeuron(preActivity*currWeight*20))
455.                 if preSpikeRate >= self.spikingThreshold and self.dataList[i][138] == 0:
456.                     currWeight = self.outputLayer.weights[k]
457.                     deltaW = (self.learningRate * 1 * (1 - 1*currWeight))/self.timeStep
458.                     if(currWeight-deltaW >=-1):
459.                         self.outputLayer.weights[k] -= deltaW
460.                         self.outputLayer.weights[k] = round(self.outputLayer.weights[k])
461.                     else:
462.                         self.outputLayer.weights[k] = -1.000
463.                     # print("decreased weight from ", currWeight, " to ", self.outputLayer.weights[k], "
    with delta ", round(deltaW), " for input ", input[i], " ")
464.                 elif preSpikeRate >= self.spikingThreshold and self.dataList[i][138] == 1:
465.                     currWeight = self.outputLayer.weights[k]
466.                     deltaW = (self.learningRate * 1 * (1 - 1*currWeight))/self.timeStep
467.                     if(currWeight+deltaW <=1):
468.                         self.outputLayer.weights[k] += deltaW
469.                         self.outputLayer.weights[k] = round(self.outputLayer.weights[k])
470.                     else:
471.                         self.outputLayer.weights[k] = 1.000
472.                     # print("increased weight from ", currWeight, " to ", self.outputLayer.weights[k], "
    with delta ", round(deltaW), " for input ", input[i], " ")
473.                 print("Weight for inhibitory output ", input[i][138], " = ", self.outputLayer.weights)
474.
475.     def classify(self, inputs):
476.         correctlyClassified = 0
477.         incorrectlyClassified = 0
478.         total = len(inputs)
479.         firingRates = []
480.
481.         for x in range(len(inputs)):
482.             input = inputs[x]
483.             currGenre = input[x][138]
484.             for k in range(len(self.inputLayer)):
485.                 neuron = self.inputLayer[k]
486.                 currSpikeRate = 0
487.                 for i in range(len(input)):
488.                     currSpikeRate += neuron.runNeuron(input[i][k]*5+7.9)
489.                 neuron.classificationRate = currSpikeRate
490.
491.             for i in range(len(self.inputLayer)):
492.                 neuron = self.inputLayer[i]
493.                 currSpikeRate = 0
494.                 currActivity = 1 if neuron.classificationRate > self.spikingThreshold else 0
495.                 neuron.classificationActivity = currActivity
496.
497.             #Layer 1
498.             for k in range(len(self.middleLayer[0])):
499.                 neuron = self.middleLayer[0][k]
500.                 currSpikeRate = 0
501.                 multiplier = 0.7
```

```
                if(self.inputLayer[k].classificationActivity == 0):
                    multiplier = 0.5
                for i in range(len(self.middleLayer[0][k].weights)):
                    currSpikeRate += neuron.runNeuron(multiplier*neuron.weights[k]*self.inputLayer[k].cl
assificationRate*1.7)
                neuron.classificationRate = currSpikeRate
                # print("Layer 1: ", currSpikeRate)

            for i in range(len(self.middleLayer[0])):
                neuron = self.middleLayer[0][i]
                currSpikeRate = 0
                currActivity = 1 if neuron.classificationRate > self.spikingThreshold else 0
                neuron.classificationActivity = currActivity
            #layer 2
            for k in range(len(self.middleLayer[1])):
                neuron = self.middleLayer[1][k]
                currSpikeRate = 0
                multiplier = 0.7
                for i in range(len(self.middleLayer[1][k].weights)):
                    if(self.middleLayer[0][i].classificationActivity == 0 or input[0][138] == 0):
                        multiplier = 0.5
                    currSpikeRate += neuron.runNeuron(multiplier*neuron.weights[i]*self.middleLayer[0][i
].classificationRate*1.7)
                neuron.classificationRate = currSpikeRate
                # print("Layer 2: ", currSpikeRate)

            for i in range(len(self.middleLayer[1])):
                neuron = self.middleLayer[1][i]
                currSpikeRate = 0
                currActivity = 1 if neuron.classificationRate > self.spikingThreshold else 0
                neuron.classificationActivity = currActivity


            #layer 3
            for k in range(len(self.middleLayer[2])):
                neuron = self.middleLayer[2][k]
                currSpikeRate = 0
                multiplier = 0.7
                for i in range(len(self.middleLayer[1][k].weights)):
                    if(self.middleLayer[1][i].classificationActivity == 0 or input[0][138] == 0):
                        multiplier = 0.5
                    currSpikeRate += neuron.runNeuron(multiplier*neuron.weights[i]*self.middleLayer[1][i
].classificationRate*1.7)
                neuron.classificationRate = currSpikeRate
                # print("Layer 3: ", currSpikeRate)

            for i in range(len(self.middleLayer[2])):
                neuron = self.middleLayer[2][i]
                currSpikeRate = 0
                currActivity = 1 if neuron.classificationRate > self.spikingThreshold else 0
                neuron.classificationActivity = currActivity

            #output layer
            outputSpikingRate = 0
            currSpikeRate = 0
            multiplier = 0.7
            for i in range(len(self.middleLayer[2])):
                if(self.middleLayer[2][i].classificationActivity == 0  or input[0][138] == 0):
                    multiplier = 0.5
                currSpikeRate += self.outputLayer.runNeuron(multiplier*self.outputLayer.weights[i]*self.
middleLayer[2][i].classificationRate*1.7)
```

```
            outputSpikingRate = currSpikeRate

            print("Ouput firing rate: ",outputSpikingRate," for genre ", currGenre)

    def round(input):
        return math.ceil(input*100000)/100000


test = GenreClassifier()
test.getMetalFiles()
test.getClassicalFiles()
np.random.shuffle(test.dataList)
# print("Reading file: \n\n")
# test.dataList = np.genfromtxt("global.csv", delimiter=',')
print(test.dataList, "\nShape: ", test.dataList.shape)
# test.train()
test.saveWeights()
test.getWeights()
test.isFirst = 0
classificationInput = np.concatenate((test.getClassificationMetalInput(), test.getClassificationClassica
lInput()),axis=0)
np.random.shuffle(classificationInput)
test.classify(classificationInput)
# test.train()
# test.train()
# test.classify()
```

Spectrogram Generation Code Using Librosa

```
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
import os
import librosa
import librosa.display

n_fft = 1024
lag = 2
n_mels = 138
fmin = 27.5
fmax = 16000.
max_size = 3


source = '.'
EXT = ('.wav')
for root, dirs, filenames in os.walk(source):
    for f in filenames:
        if f.endswith(EXT):
            print (f)
            fullpath = os.path.join(source, f)
            log = open(fullpath, 'r')

            # Read wac
            y, sr = librosa.load(f,
                     sr=11025,
                     duration=10,
```

```
                    offset=0)
            hop_length = int(librosa.time_to_samples(1. / 200, sr=sr))
            # Generate Spectrogram array
            S = librosa.feature.melspectrogram(y, sr=sr, n_fft=n_fft,
                                               hop_length=hop_length,
                                               fmin=fmin,
                                               fmax=fmax,
                                               n_mels=n_mels)

            np.savetxt(f + ".csv", S, delimiter=",")

            # Plot Spectrogram
            plt.figure(figsize=(20, 4))
            librosa.display.specshow(librosa.power_to_db(S, ref=np.max),
                                     y_axis='mel', x_axis='time', sr=sr,
                                     hop_length=hop_length, fmin=fmin, fmax=fmax)

            plt.savefig(f + ".png")
```