MusEEG: An Acessible, Gesture-Based Brain-Computer Music Interface

Hugo Flores García
Department of Electrical and Computer Engineering
Georgia Southern University
Statesboro, GA
hf01049@goergiasouthern.edu

***Abstract*—An open-source, gesture-based brain-computer interface (BCI) for music performance is proposed. After a user performs a pre-defined facial gesture, the gesture is classified using feature extraction and machine learning methods. Raw EEG data is streamed using the commercial grade Emotiv® EPOC+, preprocessed using a 4-level wavelet decomposition, and classified using an artificial neural network (ANN) classifier. Each gesture is mapped into a MIDI event, which is then performed in a software instrument of the user's choice. By using a commercial-grade headset and a workflow that is easy to comprehend, MusEEG aims to provide people with motor disabilities and little musical experience with an opportunity to make music.**

Keywords—Brain-Computer Interface, MIDI, Artificial Neural Networks

I. Introduction

Due to the rigorous physical nature of performing an instrument, accessibility tends to be a problem in music. Traditional instruments such as the piano, guitar, and violin, require the performer to have a high level of manual dexterity in order to perform the instrument properly. This presents a challenge to people with disabilities, notably those with motor impairments. Recent research has addressed this problem, introducing new approaches to music making through the use of digital musical instruments, such as prosthetic devices, mouth-operated controllers, touchscreen controllers and mouse-controlled interfaces [15].

Recent interest in human-computer interaction (HCI), computer music and brain-computer interfaces (BCIs) have allowed for the creation of music-specific BCIs, also known as brain-computer music interfaces (BCMIs). BCMIs have been designed for numerous applications, such as musical neurofeedback [2], music composition [3], and special needs [1]. Recently, BCMIs have been widely used for music performance applications [3, 5-10]. Through direct interaction with the human brain, BCMIs have essentially redefined several aspects of traditional music making, introducing novel musical pieces written for brain-computer interfaces, adding direct affective components to music performance and directly generating musical ideas from EEG signals through sonification.

Though these approaches introduce great artistic, compositional, and experimental value, the use of such BCMIs for accessible applications may imply a steep learning curve, as these systems may be hard to set up or difficult to understand for a novice musician. In addition, some of these systems are designed for medical-grade EEG devices, making affordability an issue when considering widespread use.

The purpose of MusEEG is to reduce the accessibility gap present in music performance by introducing a musical instrument that not only is accessible to people who possess severe motor disabilities but also provides a plug-and-play interface with different levels of musical abstraction, allowing for people with little musical experience to be able to make music.

II. Data Acquisition, Preprocessing, and Training

A. The Emotiv® EPOC+

Although easily modifiable, the MusEEG package designed to work with the Emotiv® EPOC+. The EPOC+ is an affordable commercial-grade EEG headset that records 14 EEG channels and samples at a rate of 256Hz with 14-bit resolution, with a least significant bit value of approximately 0.51μV [4]. The EPOC+ headset proves to be an economically feasible alternative to a medical-grade headset as various studies confirm its viability for noncritical applications [10-12].

Fig. 1. Emotiv EPOC+ Headset [4].

B. Data Acquisition

To maximize classification accuracy, the MusEEG package is designed with a train-it-yourself structure, meaning that the end-user will have to train their own ANN model to work with their preferred set of facial or body gestures. Because of the train-it-yourself nature of the package, data was recorded for a single subject only.

9 gestures were recorded using the Emotiv® PRO application. 60 samples were recorded of the following gestures:

- Smile
- Bite Lower Lip
- Hard Blink

- Look Left
- Look Right
- Neutral
- Scrunch
- Stick Tongue Out

To expedite recording times, the MusEEG package was designed to allow the user to record all samples of a single gesture to a single .csv file. The MusEEG package then aids the user through curating and cutting the samples into individual chunks for feature extraction and classification.

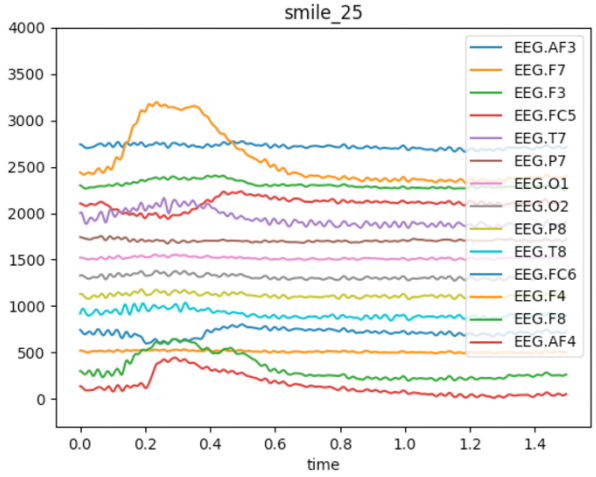

Fig. 2. Raw EEG plot of smile_25 (bigChunk).

During the curation process, the samples were examined for any discontinuities in contact quality, noise, and other artifacts. Samples that appeared to be corrupted were discarded, while samples that were clean were stored in the project's saved chunks directory. From each sample, two different sample chunks were created: a bigChunk (1500ms, 384 samples) for gesture recognition, and a smallChunk (250ms, 64 samples) for gesture/no gesture classification. Examples of a bigChunk and smallChunk can be observed in Figs. 2 and 3, respectively.

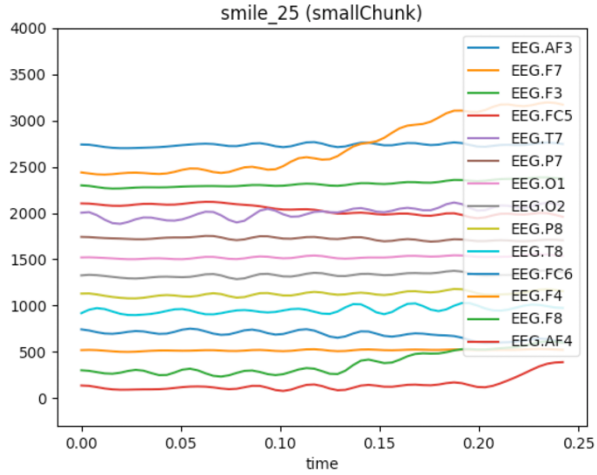

Fig. 3. Raw EEG plot of smile_25 (smallChunk).

C. *Feature Extraction*

A feature extraction method similar to the one in [14] was used. A 4-level wavelet decomposition using Daubechies order-2 (db2) mother wavelet is performed on all 14 channels of the chunk of EEG data.

One advantage of wavelet analysis over other time-frequency distribution methods (e.g. STFT) is that wavelet analysis varies the time-frequency aspect ratio, producing good frequency localization at low frequencies (long time windows), and good time localization at high frequencies (short time windows). This results in a segmentation of the time-frequency plane that will reveal transient features of the signal, which are typically not obvious during Fourier analysis [14].

Following the wavelet decomposition, the first four statistical moments (mean, variance, skewness, kurtosis) are calculated for each wavelet vector. Since four moments are calculated for each of the 5 wavelet decomposition vector per EEG channel, a total of 14 x 4 x 5 (280) features are calculated. These features are used as an input for the classification models.

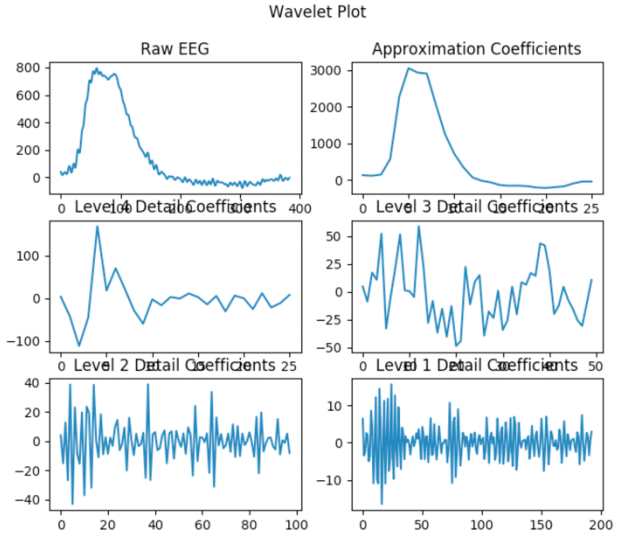

Fig. 4. Wavelet Coefficients plot for smile gesture.

D. *Classification*

A three-layer Artificial Neural Network (ANN) architecture was used for the classification of the EEG signals. Two models were created: a small model designed to analyze 64-sample chunks and determine whether the chunk contains a facial gesture or not (smallBrain), and a large model designed to analyze 384-sample chunks to determine what gesture is present in the chunk (bigBrain).

The smallBrain model is meant to perform a quick classification of whether a gesture is present in a 64-sample chunk. The smallBrain model was designed with the intent of having an always-on processor for real-time implementation. By analyzing a chunk with a duration of 250ms, the smallBrain model returns a result fairly quickly, allowing for continuous real-time classification with little latency.

```
brain.build_model(inputShape=brain.inputShape,
                  hiddenNeurons=100,
                  hiddenActivation='elu',
                  numberOfTargets=max(train_targets)+1,
                  regularization='l1',
                  loss='sparse_categorical_crossentropy')
```

Fig. 5. ANN architecture for smallBrain model.

The smallBrain model consists of a three-layer ANN with 100 neurons in its hidden layer, uses L1 regularization methods and a sparse categorical cross entropy loss function, as shown in Fig. 5. During testing, the model exhibited 98.75% accuracy. Its test confusion matrix can be observed in Fig. 6.

```
Test accuracy: 0.9875
[[38  0]
 [ 1 41]]
```

Fig. 6. Confusion Matrix for smallBrain model

The bigBrain model consists of a three-layer ANN with 175 neurons in its hidden layer, uses L1_L2 regularization methods and a sparse categorical cross entropy loss function, as shown in Fig. 7. During testing, the model exhibited 92.59% accuracy. Its test confusion matrix can be observed in Fig. 8.

```
brain.build_model(inputShape=brain.inputShape,
                  hiddenNeurons=175,
                  hiddenActivation='elu',
                  numberOfTargets=max(train_targets) + 1,
                  regularization='l1_l2 ',
                  loss='sparse_categorical_crossentropy')
```

Fig. 7. ANN architecture for bigBrain model.

```
Test accuracy: 0.9259259
[[21  0  0  0  0  0  0  0  1]
 [ 0 14  0  0  0  0  0  0  0]
 [ 0  0 14  0  0  0  0  0  0]
 [ 0  0  0 13  0  0  0  0  0]
 [ 3  0  0  0 14  0  0  0  0]
 [ 0  1  0  0  1 12  0  0  0]
 [ 0  0  0  1  0  0 13  0  0]
 [ 0  0  0  0  1  0  0 11  0]
 [ 1  0  0  0  0  0  1  0 13]]
```

Fig. 8. Confusion matrix for bigBrain model.

III. Application Development

The MusEEG package is a Python package containing useful classes and methods that aid developers in preprocessing and classifying EEG data, creating MIDI objects, and designing a Brain-Computer Music Interface that allows a user to trigger MIDI events from performing facial gestures. The MusEEG project is available as an open-source package and can be downloaded from its GitHub repository.

A. Open Source Implications

Due to the open-source, train-it yourself nature of the MusEEG project, a particular subject's EEG data is handled by only the subject himself, as it is advised that the neural network model is trained for a single individual only for higher classification accuracy.

In order for a user to create their own personalized neural network model, the user must first download the MusEEG package from GitHub to their personal computer. The MusEEG eegData module's training methods facilitate the data collection process, organizing and labeling the training data into processing chunks inside the MusEEG directory. The processed and labeled training data is then used to create and train a new neural network model customized to the user's training data, which will also be stored inside the MusEEG directory.

It should be noted that all of the EEG data acquired by the user is processed and saved inside the user's local MusEEG directory and is never uploaded to the web. Thus, each user's EEG data and trained neural network models reside inside the same user's computer, and are never shared, accessed, or viewed by any other subject without the original user's consent and deliberate intent.

IV. Project Structure

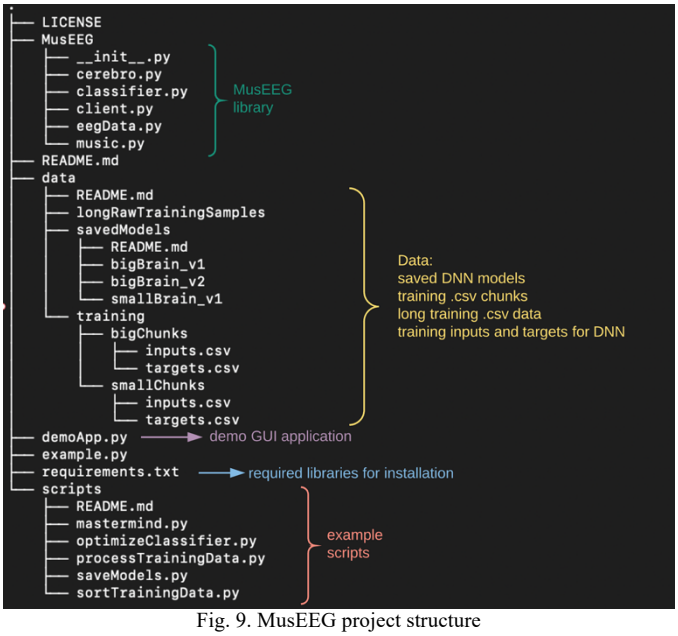

Fig. 9. MusEEG project structure

The MusEEG directory (Fig. 9) consists of a data directory, an example scripts directory, the MusEEG library, a README file, a library requirements file, a demo application, and an MIT License file.

A. Data Directory

The /data directory of the MusEEG module stores EEG data in all of its different training stages, inputs and targets for the ANN models, as well as the ANN models themselves.

The /data/longRawTrainingSamples subdirectory stores .csv files with multiple samples of a single gesture. During the normal workflow of the data acquisition process, the files from the longRawTrainingSamples subdirectory are curated and stored into individual chunks in the /data/savedChunks subdirectory, which in itself contains subdirectories for smallChunks and bigChunks. The /data/savedModels subdirectory saves Keras models that have been trained and are ready for usage, while the /data/training subdirectory stores preprocessed and prelabeled input and target vectors for designing ANN models.

B. MusEEG Library

The MusEEG library contains all the required classes to build a brain-computer interface for music performance. It is organized into five modules:

- eegData.py (import, process, plot, save EEG data).
- music.py (MIDI objects, chords, and melodies)
- classifier.py (build, train, save keras models easily)
- cerebro.py (methods to use music, classifier, and eegData together)
- client.py (TCP client setup to receive live raw EEG data stream from EPOC+ headset)

```
class eegData:
    threshold = 1000
    sampleRate = 256
    chunkSize = int(256*1.5)
    smallchunkSize = int(chunkSize/6)
    backTrack = 35
    nchannels = 14
    emotivChannels = [...]
    thresholdChannel = 'F7'

    def wavelet(self):...

    def extractStatsFromWavelets(self):...

    def flattenIntoVector(self):...

    ...
    def plotRawEEG(self, figure=None, offset=200, plotTitle='eeg'):...

    def plotWavelets(self, channel):...

    def loadChunkFromTraining(self, subdir, filename):...

    def cutChunk(self):...

    def process(self):...
```

Fig. 10. Overview of eegData class

The eegData class (Fig. 10) helps a user import raw EEG data from .csv files, curate it into processable chunks, process using wavelet decomposition and statistical extraction, as well as plot the 14-channel EEG signal or the five coefficient vectors created by the wavelet decomposition. It is currently designed around the EPOC+ model, but can be redesigned for any other EEG system, provided that the developer has a means of obtaining a raw EEG stream from such system. The eegData class builds upon the PyWavelets, matplotlib, SciPy, and Pandas Python libraries.

```
class music:
    midiChannel = 0
    tempo = 120

    def set_tempo(self, tempo):...

    def panic(self):...

    def pause(self,nQuarterNotes):...

class chord(music):
    def __init__(self, notelist, name):...

    def playchord(self, vel=64, qtrnotes=2):...


    def play(self, vel=64):...

    def stop(self):...

    def stopAllNotes(self):...

    def arpeggiate(self, notelength, vel, numTimes):...

class melody(music):
    currentTime = 0

    def __init__(self, midiname):...

    def addnote(self, note, duration, vel=64):...
```

Fig. 11. Overview of music class

The music class (Fig. 11) is designed to allow the user to create, load, and save different MIDI events for use during performance. As of now, the music class allows the creation of chord and melody objects, though future iterations will also include the option to load pre-written MIDI files created using third-party MIDI sequencing software. The music class is built upon the Mido and Audiolazy libraries.

```
class classifier:
    hiddenNeurons = 20
    numberOfTargets = 10
    inputShape = 350

    def __init__(self):...

    def loadTrainingData(self, percentTrain=0.75,
                         address=os.path.join(parentDir, 'data', 'training'),
                         subdir='bigChunks',
                         inputFilename='inputs.csv',
                         targetFilename='targets.csv'):...

    # build the model
    def build_model(self, inputShape, hiddenNeurons, numberOfTargets, hiddenActivation='relu',
                    outputActivation='softmax', regularization='l2_l2', optimizer='adam',
                    loss='sparse_categorical_crossentropy'):...

    # train the model
    def train_model(self, train_inputs, train_targets, nEpochs, verbose=0):...

    def evaluate_model(self, test_inputs, test_targets, verbose=2):...

    def print_confusion(self, test_inputs, test_targets):...

    def classify(self, inputVector):...

    def clear(self):...

    def savemodel(self, filename, address=os.path.join(parentDir, 'data', 'savedModels')):...

    def loadmodel(self, filename, address=os.path.join(parentDir, 'data', 'savedModels')):...

    def createLiteModel(self, model):...
```

Fig. 12. Overview of classifier class

The classifier class (Fig. 12) builds upon the TensorFlow and Keras libraries to expedite the process of creating, training, loading, saving, and analyzing the performance of ANN models.

```
class cerebro:
    """..."""
    demomsg = (
        '...')
    eeg = eegData()

    gestures = [...]

    """..."""
    cmaj7sharp11add13 = chord(['C4', 'E4', 'G4', 'B4', 'D5', 'F#4', 'A5'],
                              name='cmaj7sharp11add13')
    fminmaj7 = chord(['F4', 'Ab4', 'C5', 'E5'], name='fminmaj7')
    fmaj7 = chord(['F4', 'A4', 'C5', 'E5', 'G5'], name='fmaj7')
    ab69 = chord(['Ab4', 'C5', 'F5', 'Bb5', 'C6'], name='ab69')
    dmin7b5 = chord(['D4', 'F4', 'Ab4', 'C5', 'E5'], name='dmin7b5')
    g7b9 = chord(['G4', 'B4', 'D5', 'F5', 'Ab5'], name='g7b9')
    c5 = chord(['C3', 'G3'], name='c5')
    noChord = chord([], name='nochord')
    polychordcde = chord(
        ['C3', 'E3', 'G3', 'D4', 'F#4', 'A4', 'E5', 'G#5', 'B5'], name='E/D/C')  # todo:
    dbmaj7 = chord(['Db4', 'F4', 'Ab4', 'C5', 'Eb5'], name='dbmaj7')
    margaretsmagicchord = chord(['D4', 'F4', 'A#4'], name='margschord')

    defaultchordlist = [cmaj7sharp11add13.notelist, fminmaj7.notelist,
                        fmaj7.notelist, ab69.notelist, dmin7b5.notelist,
                        c5.notelist, noChord.notelist,
                        polychordcde.notelist, dbmaj7.notelist]

    """
    this dictionary is where chords are referenced to facial gestures.
    """

    def __init__(self):...

    def setupClient(self):...

    def updateChordList(self, chordlistlist):...

    def loadFromDataSet(self, name):...

    def processAndPlay(self, arp, tempo, arpDurationFromGUI, noteDurationFromGUI):...

    def perform(self, musician, arp):...
```

Fig. 13. Overview of cerebro class

The cerebro class serves as a high-level API for less experienced programmers to make use of the MusEEG package, as it contains methods that use the eegData, classifier, and music classes in conjunction to create brain-computer music interface systems. The cerebro class is built on top of the eegData, music, classifier, and client classes from the MusEEG package.

```
class client:
    BUFFER_SIZE = eegData.chunkSize
    host = "127.0.0.1"
    port = 5555
    sampleRate = eegData.sampleRate
    streamIsSimulated = False

    # Named fields according to Warren doc !
    FIELDS = {...}

    def data2dic(self, data):...

    def setup(self):...

    def stream(self):...

    def getChunk(self, chunkSize=eegData.chunkSize):...

    def simulateStream(self, gesture):...


if __name__ == "__main__":
    client = client()
    client.setup()
```

Fig. 14. Overview of client class

The client class (Fig. 15) sets up a TCP client that receives raw EEG data packets from an EEG stream server and packs them into chunks to create eegData objects. The client class is also capable of creating a raw EEG streaming simulation by effectively streaming a pre-recorded .csv EEG session into the server.

C. *Example Scripts*

The MusEEG/scripts directory contains a series of sample scripts that can be useful during the creation of a user's training samples and ANN model.

sortTrainingData.py provides an example of importing long .csv files that contain multiple samples of the same gesture from the MusEEG/data/longRawTrainingSamples directory and using the TrainingDataMacro class to evaluate, curate, and save individual training samples to the /data/savedChunks directory

processTrainingData.py grabs the curated chunks from the /data/savedChunks directory and performs the preprocessing and feature extraction routine (wavelet transform and statistical extraction), as well as creates training inputs and targets and stores them in the /data/training directory

Because the dataset is quite small and one-dimensional, training ANN models is relatively computationally inexpensive. This lets a user perform an exhaustive search of different ANN models with different activation units, hidden layer sizes, and regularization parameters within the span of a couple of minutes. OptimizeClassifier.py iterates over every combination of different hidden layer activation functions, output layer activations, regularization parameters, number of hidden neurons, and loss functions to find the one that performs with the highest accuracy. The optimizeClassifier.py script creates a .csv file in the /data/ClassifierOptimizations directory which contains the test results for each of the combinations tried.

saveModels.py trains and saves a Keras model in the /data/savedModels directory.

D. *Demo Application*

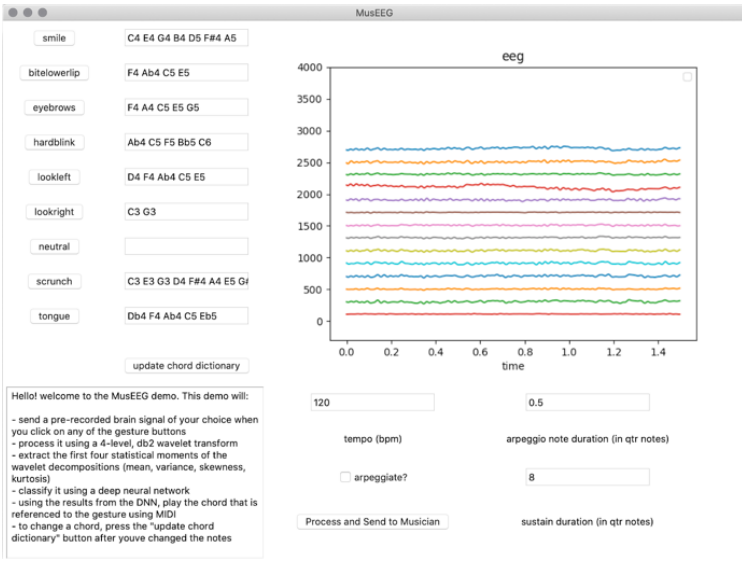

Fig. 15. Demo Application GUI

The demo application was created as a static proof-of-concept that provides a demonstration of the basic functionality of the MusEEG package. In short, the MusEEG demo application lets the user perform the following actions:

- Create a MIDI chord dictionary by entering note names next to the desired gesture
- Load a pre-recorded facial gesture sample from the dataset
- Process the pre-recorded sample using the 4-level, db2 wavelet decomposition as well as perform the statistical moments calculation.
- Classify the pre-recorded facial gesture sample using the bigBrain classifier
- Refer the resulting gesture from the classification result to its corresponding chord in the MIDI dictionary
- Send the MIDI event to a virtual MIDI port with a set of music-related control parameters (arpeggiation, tempo, sustain duration, arpeggio note duration).

The Gesture Buttons and Chord Dictionary

The top left corner of the demo application allows the user to load a random sample from the existing facial gesture dataset by pressing on one of the facial gesture buttons. Once loaded, the 14-channel raw EEG signal will be plotted in the plot box.

Next to each facial gesture button is a text entry field that allows the user to define the set of notes that will be played when the facial gesture is sent to the main processor. Note: whenever the user changes notes in the chord dictionary, the *update chord dictionary* button must be pressed in order for the changes to take effect.

Process and Send to Musician Button

The process and send to musician button performs the following actions:

1. wavelet transform of raw EEG signal
2. statistical moment calculations of wavelet decomposition vectors
3. creates DNN input array from extracted stat moments
4. the DNN classifies the signal into either of the available gestures
5. the gesture is referred to its matching chord in the chord dictionary, and a chord object is created from the matching chord
6. the playchord method is called on the chord object, sending a MIDI message according to the additional control parameters

Additional Controls

Additional controls are available for the demo app:

- arpeggiate: if the box is checked, the chord will play in an arpeggio as opposed to vertically.
- sustain duration: indicates how long (in quarter notes) the chord will be sustained (only applied if arpeggiate is unchecked)
- arpeggio note duration: indicates how long (in quarter notes) each note in the arpeggio will last.

V. Future Work

A. Real-Time Application

A real-time application of the brain-computer interface is currently in progress. The current iteration of the real-time application consists of a multithreaded process (Fig. 16). This method has not yet been tested properly and may need adjustments in order to ensure proper performance.

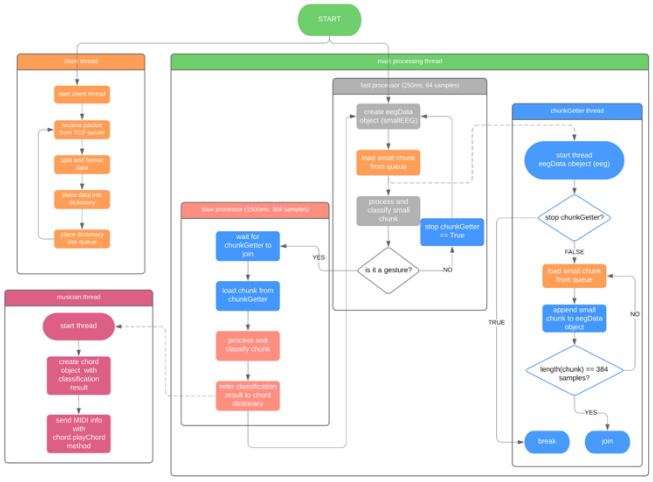

Fig. 16. Flowchart for real-time application

In the client thread (Fig. 17), a TCP client receives single EEG data packets from a live EEG stream using the EPOC+. Each packet is placed into a python LIFO queue (stack), waiting to be retrieved by the main processing thread when requested.

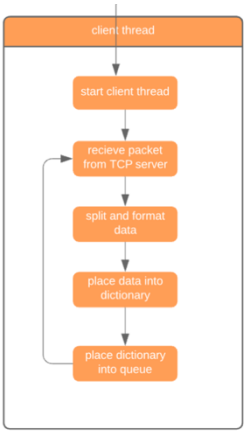

Fig. 17. Client thread

In the main processing thread (Fig. 18), a helper chunkGetter thread takes care of receiving EEG packets from the client thread and placing them into smallChunks and bigChunks for processing by smallBrain and bigBrain, respectively. Once the smallChunk array is full (250ms, 64 samples), the chunk is sent through preprocessing and feature extraction, and is classified into either a gesture/no gesture. While the smallChunk array goes through preprocessing and feature extraction, the chunkGetter thread takes care of filling up the bigChunk array,

in case the 384-sample chunk is needed for complete gesture classification.

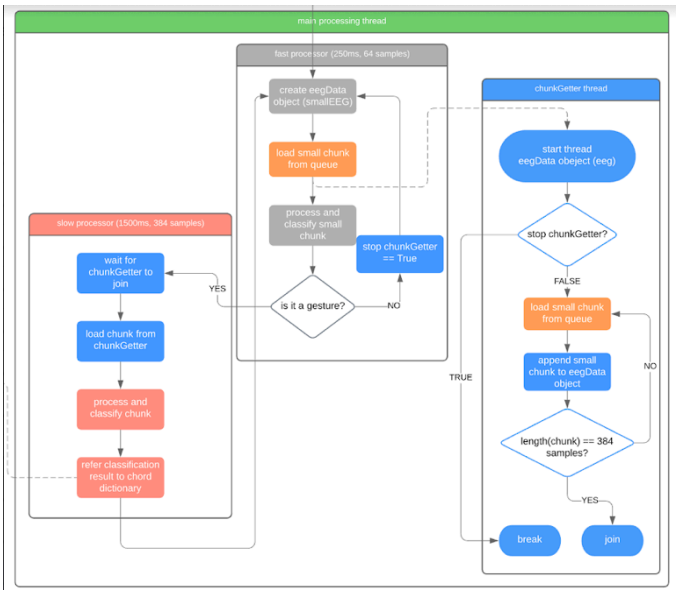

Fig. 18. Main processing thread

If the smallChunk classification indicates that a gesture was just performed, the main thread waits for the chunkGetter thread to finish collecting all 384 samples into a bigChunk, and sends the bigChunk to be processed and classified using bigBrain. The classification result is sent to a musician thread (Fig. 19), which in turn performs the predefined MIDI event indicated by the gesture-MIDI dictionary.

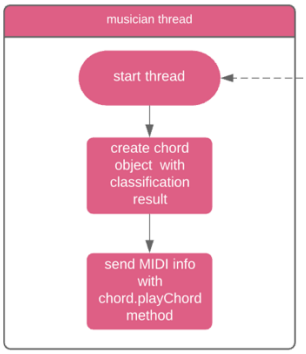

Fig. 19. Musician thread

If the smallChunk classification indicates that no gesture is present, the chunkGetter thread empties its chunks and breaks to start the process over again.

B. Affective Computing

One of the primary advantages of Brain-Computer Interfaces is their close connection to affective computing. Affective BCIs incorporate emotion into a system, allowing for a human's affective state to be used as a control parameter in the system.

An extensive literature review on the existing techniques and methods of extracting affective information from EEG signals will be performed. One of the possible ways of incorporating affective components into the MusEEG project includes measuring musical engagement using a method similar to the one proposed in [16]. By measuring the level of engagement present in the performer, the MusEEG module will be able to suggest different chord/melody banks or fluctuate tempo according to the level of engagement present in the performer.

VI. Conclusion

This report presented a BCMI with the purpose of providing an easy-to-use musical interface for people with motor disabilities. The MusEEG BCMI allows users to trigger and control MIDI events by performing facial gestures. By using gesture recognition and high-level musical objects such as pre-built chords and melodies, MusEEG seeks to provide an inclusive interface that enables users to create musical ideas with little musical experience. Implementing the MusEEG system requires nothing more than a laptop, a software instrument (either through a digital audio workstation or Max/MSP), and a commercial-grade EEG headset, making it one of the most affordable systems for inclusive music performance.

References

[1] E. R. Miranda, W. L. Magee, J. J. Wilson, J. Eaton, and R. Palaniappan, "Brain-Computer Music Interfacing (BCMI): From Basic Research to the Real World of Special Needs," Music and Medicine, vol. 3, no. 3, pp. 134–140, Jul. 2011.

[2] S. Le Groux, P. Verschure, "Neuromuse: Trianing Your Brain Through Musical Interaction," Proceedings of the International Conference on Auditory Display, Jan. 2009.

[3] E. R. Miranda, "Brain-computer music interface for composition and performance," International Journal on Disability and Human Development, vol. 5, no. 2, 2006.

[4] "EMOTIV EPOC 14-Channel Wireless EEG Headset," EMOTIV. [Online]. Available: https://www.emotiv.com/epoc/. [Accessed: 11-Dec-2019].

[5] J. Eaton, D. Williams, and E. Miranda, "The Space Between Us: Evaluating a multi-user affective brain-computer music interface," Brain-Computer Interfaces, vol. 2, no. 2-3, pp. 103–116, Mar. 2015.

[6] C. Mikalauskas, T. Wun, K. Ta, J. Horacsek, and L. Oehlberg, "Improvising with an Audience-Controlled Robot Performer," Proceedings of the 2018 on Designing Interactive Systems Conference 2018 - DIS 18, 2018.

[7] C. Levican, A. Aparicio, V. Belaunde, R. Cadiz, "Insight2OSC: using the brain and the body as a musical instrument with the Emotiv Insight," Proc. NIME'17, May15-19, 2017.

[8] T. Hamano, T. Rutkowski, H. Terasawa, K. Okanoya, K. Furukawa, "Generating an Integrated Musical Expression with a Brain-Computer Interface," Proc. NIME'13, May 27-30, 2013.

[9] T. Tokunaga, M. Lyons, "Enactive Mandala: Audio-visualizing Brain Waves," Proc. NIME'13, May-27-30, 2013.

[10] A. Mura, J. Manzolli, B. Rezazadeh, S. Le Groux, M. Sanchez, A. Väljamäe, A. Luvizotto, C. Guger, U. Bernardet, P. Verschure, "Disembodied and Collaborative

Musical Interaction in the Multimodal Brain Orchestra," Proc. NIME '10, 2010.

[11] M. Duvinage, T. Castermans, M. Petieau, T. Hoellinger, G. Cheron, and T. Dutoit, "Performance of the Emotiv Epoc headset for P300-based applications," BioMedical Engineering OnLine, vol. 12, no. 1, p. 56, 2013.

[12] N. A. Badcock, K. A. Preece, B. D. Wit, K. Glenn, N. Fieder, J. Thie, and G. Mcarthur, "Validation of the Emotiv EPOC EEG system for research quality auditory event-related potentials in children," PeerJ, vol. 3, 2015.

[13] N. A. Badcock, P. Mousikou, Y. Mahajan, P. De Lissa, J. Thie, and G. Mcarthur, "Validation of the Emotiv EPOC®EEG gaming system for measuring research quality auditory ERPs," PeerJ, vol. 1, 2013.

[14] P. Jahankhani, V. Kodogiannis, and K. Revett, "EEG Signal Classification Using Wavelet Feature Extraction and Neural Networks," IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing (JVA06), 2006.

[15] E. Frid, "Accessible Digital Musical Instruments—A Review of Musical Interfaces in Inclusive Music Practice," Multimodal Technologies and Interaction, vol. 3, no. 3, p. 57, 2019.

[16] G. Leslie, A. Ojeda and S. Makeig, "Towards an Affective Brain-Computer Interface Monitoring Musical Engagement," 2013 Humaine Association Conference on Affective Computing and Intelligent Interaction, Geneva, 2013, pp. 871-875.

Program Listing

eegData module

```
import os

import pandas
import numpy as np
from pywt import wavedec

import pickle
from scipy.stats import kurtosis, skew

import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
plt.ion()
from matplotlib.figure import Figure

from MusEEG import parentDir

class eegData:
    threshold = 1000
    sampleRate = 256
    chunkSize = int(256*1.5)
    smallchunkSize = int(chunkSize/6)
    backTrack = 35
    nchannels = 14
    emotivChannels = ["EEG.AF3", "EEG.F7", "EEG.F3", "EEG.FC5", "EEG.T7", "EEG.P7", "EEG.O1",
                      "EEG.O2", "EEG.P8", "EEG.T8", "EEG.FC6", "EEG.F4", "EEG.F8",
"EEG.AF4"]
    thresholdChannel = 'F7'

    def wavelet(self):
        """
        wavelet transform (4-level) for a single eeg chunk
        input argument is a chunk
        creates a self.wavelets list which contains np arrays with the coefficients
        """
        self.nchannels = len(self.chunk[0, :])
        self.cA4 = []
        self.cD4 = []
        self.cD3 = []
        self.cD2 = []
        self.cD1 = []
        for i in range(0, self.nchannels):
            cA4, cD4, cD3, cD2, cD1 = wavedec(self.chunk[:, i], 'db2', level=4)
            self.cA4.append(cA4)
            self.cD4.append(cD4)
            self.cD3.append(cD3)
            self.cD2.append(cD2)
            self.cD1.append(cD1)

        self.wavelets = [np.asarray(self.cA4), np.asarray(self.cD4), np.asarray(self.cD3),
np.asarray(self.cD2), np.asarray(self.cD1)]

    def extractStatsFromWavelets(self):
        """
        calculates mean, standard deviation, variance, kurtosis, and skewness from self.wavelets
object.
        creates self.mean, self.std, self.kurtosis, self.skew which are numpy arrays with 14
rows (eeg channels)
```

```
        and 5 columns (per coefficients)
        """
        self.mean = np.ndarray((self.nchannels, 5))
        self.var = np.ndarray((self.nchannels, 5))
        self.std = np.ndarray((self.nchannels, 5))
        self.kurtosis = np.ndarray((self.nchannels, 5))
        self.skew = np.ndarray((self.nchannels, 5))
        ## ojo, dimensions are transposed here
        for i in range(0, self.nchannels):
            self.mean[i, :] = [np.mean(self.cA4[i]), np.mean(self.cD4[i]), np.mean(self.cD3[i]),
np.mean(self.cD2[i]), np.mean(self.cD1[i])]
            self.var[i, :] = [np.var(self.cA4[i]), np.var(self.cD4[i]), np.var(self.cD3[i]),
np.var(self.cD2[i]), np.var(self.cD1[i])]
            self.std[i, :] = [np.std(self.cA4[i]), np.std(self.cD4[i]), np.std(self.cD3[i]),
np.std(self.cD2[i]), np.std(self.cD1[i])]
            self.kurtosis[i, :] = [kurtosis(self.cA4[i]), kurtosis(self.cD4[i]),
kurtosis(self.cD3[i]), kurtosis(self.cD2[i]), kurtosis(self.cD1[i])]
            self.skew[i, :] = [skew(self.cA4[i]), skew(self.cD4[i]), skew(self.cD3[i]),
skew(self.cD2[i]), skew(self.cD1[i])]

    def flattenIntoVector(self):
        """
        creates an input array for ANN, structured as:
            [mean, var, std, kurtosis, skew]
            each of these is 14 channels * 5 wavelet coefficients long = 70 floats
            vector is flattened, and for 5 stat features * 70 numbers = 350 numbers
        """
        mean = self.mean.flatten()
        var = self.var.flatten()
        std = self.std.flatten()
        kurtosis = self.kurtosis.flatten()
        skew = self.skew.flatten()

        self.inputVector = np.array([mean, var, std, kurtosis, skew])
        self.inputVector = self.inputVector.flatten()
        return self.inputVector

    # def plotRawEEG(self, offset=200, title='eeg'):
    #     """
    #     :param title: title of the figure
    #     :param offset: DC offset between eeg channels
    #     :return: plot with all 14 eeg channels
    #     """
    #     # define time axis
    #     tAxis = np.arange(0, len(self.chunk))  # create time axis w same length as the data
matrix
    #     tAxis = tAxis / self.sampleRate  # adjust time axis to 256 sample rate
    #
    #     # use eeg matrix as y axis
    #     yAxis = self.chunk + offset * 13
    #
    #     # add offset to display all channels
    #     for i in range(0, len(self.chunk[0, :])):
    #         yAxis[:, i] -= offset * i
    #
    #     # plot figure
    #     plt.figure()
    #     plt.plot(tAxis, yAxis)
    #     plt.title(title)
    #     plt.ylim(-300, offset * 20)
    #     plt.legend(["EEG.AF3", "EEG.F7", "EEG.F3", "EEG.FC5", "EEG.T7", "EEG.P7", "EEG.O1",
    #                 "EEG.O2", "EEG.P8", "EEG.T8", "EEG.FC6", "EEG.F4", "EEG.F8", "EEG.AF4"],
    #                loc='upper right')
```

```
    #     plt.xlabel('time')
    #     plt.show(block=True)
    #     plt.pause(.001)
    def plotRawEEG(self, figure=None, offset=200, plotTitle='eeg'):
        """
        this version spits out a figure window for use in the UI
        :param title: title of the figure
        :param offset: DC offset between eeg channels
        :return: plot with all 14 eeg channels
        """
        # define time axis
        tAxis = np.arange(0, len(self.chunk))  # create time axis w same length as the data
matrix
        tAxis = tAxis / self.sampleRate  # adjust time axis to 256 sample rate

        # use eeg matrix as y axis
        yAxis = self.chunk + offset * 13

        # add offset to display all channels
        for i in range(0, len(self.chunk[0, :])):
            yAxis[:, i] -= offset * i

        # plot figure
        if figure is None:
            figure = Figure()

        figure.canvas.flush_events()
        ax = figure.add_subplot(111)
        ax.clear()
        ax.set_title(plotTitle)
        ax.set_ylim(-300, offset * 20)
        ax.legend(["EEG.AF3", "EEG.F7", "EEG.F3", "EEG.FC5", "EEG.T7", "EEG.P7", "EEG.O1",
                   "EEG.O2", "EEG.P8", "EEG.T8", "EEG.FC6", "EEG.F4", "EEG.F8", "EEG.AF4"])
        ax.set_xlabel('time')
        ax.plot(tAxis, yAxis)
        figure.canvas.draw()
        plt.pause(0.001)

        return figure

        # plt.pause(0.01)

    def plotWavelets(self, channel):
        """
        plots wavelet decomposition of a single channel self.chunk
        :param channel: (between 0 and 13) eeg channel to be plotted
        :return: figure
        """
        fig = Figure()
        # fig.suptitle('Wavelet Plot')
        ax = [0 for i in range(0, 6)]
        try:
            ax[0] = fig.add_subplot(611)
            ax[0].plot(self.chunk[:, channel])
            ax[0].set_title('Raw EEG')

            ax[1] = fig.add_subplot(612)
            ax[1].plot(self.cA4[:][channel])
            ax[1].set_title('Approximation Coefficients')

            ax[2] = fig.add_subplot(613)
            ax[2].plot(self.cD4[:][channel])
            ax[2].set_title('Level 4 Detail Coefficients')
```

```
            ax[3] = fig.add_subplot(614)
            ax[3].plot(self.cD3[:][channel])
            ax[3].set_title('Level 3 Detail Coefficients')

            ax[4] = fig.add_subplot(615)
            ax[4].plot(self.cD2[:][channel])
            ax[4].set_title('Level 2 Detail Coefficients')

            ax[5] = fig.add_subplot(616)
            ax[5].plot(self.cD1[:][channel])
            ax[5].set_title('Level 1 Detail Coefficients')
            fig.show()
        except AttributeError:
            pass
        #todo: make this work with the UI
        return fig

    def loadChunkFromTraining(self, subdir, filename):
        """
        :param subdir: subdirectory where chunk is located from MusEEG/data/savedChunks
        :param filename: filename with .csv at the end
        :return:
        """
        self.filename = filename
        self.chunk = pandas.read_csv(os.path.join(parentDir, 'data', 'savedChunks', subdir,
filename), usecols=self.emotivChannels)
        self.chunk = self.chunk.values
        self.AF3 = self.chunk[:, 0]
        self.F7 = self.chunk[:, 1]
        self.F3 = self.chunk[:, 2]
        self.FC5 = self.chunk[:, 3]
        self.T7 = self.chunk[:, 4]
        self.P7 = self.chunk[:, 5]
        self.O1 = self.chunk[:, 6]
        self.O2 = self.chunk[:, 7]
        self.P8 = self.chunk[:, 8]
        self.T8 = self.chunk[:, 9]
        self.FC6 = self.chunk[:, 10]
        self.F4 = self.chunk[:, 11]
        self.F8 = self.chunk[:, 12]
        self.AF4 = self.chunk[:, 13]
        return self.chunk

    def cutChunk(self):
        """
        for smallBrain: cut chunk to smallchunkSize
        """
        self.chunk = self.chunk[0:(self.smallchunkSize-1), :]

    def process(self):
        self.wavelet()
        self.extractStatsFromWavelets()
        inputVector = self.flattenIntoVector()
        return inputVector

#todo: this has to work with relative paths
class TrainingDataMacro(eegData):
    """
    child eegData class meant for user to evaluate a long .csv file with multiple training
samples in it
    reads rawdata and creates a long self.rawData object with ALL the data
```

```
    self.matrix contains only numbers w DC offset removed

    """
    def __init__(self):
        super().__init__()
        self.curatedChunk = []
        self.label = []

    def importCSV(self, subdir, filename, tag):
        """
        :param subdir: subdirectory under MusEEG/data/longRawTrainingSamples where the .csv
files are located
        :param filename: filename of the .csv file
        :param tag: the label you would like to associate the file with. typically the same as
filename.
        :return:
        """
        self.rawData = pandas.read_csv(os.path.join(parentDir, 'data', 'longRawTrainingSamples',
subdir, filename), skiprows=1, dtype=float, header=0,
                                       usecols=self.emotivChannels)
        self.markers = pandas.read_csv(os.path.join(parentDir, 'data','longRawTrainingSamples',
subdir, filename), skiprows=1, usecols=['EEG.MarkerHardware'])
        self.address = subdir
        self.filename = filename
        self.tag = tag
        self.matrix = np.array(self.rawData.values - 4100)
        self.AF3 = self.matrix[:, 0]
        self.F7 = self.matrix[:, 1]
        self.F3 = self.matrix[:, 2]
        self.FC5 = self.matrix[:, 3]
        self.T7 = self.matrix[:, 4]
        self.P7 = self.matrix[:, 5]
        self.O1 = self.matrix[:, 6]
        self.O2 = self.matrix[:, 7]
        self.P8 = self.matrix[:, 8]
        self.T8 = self.matrix[:, 9]
        self.FC6 = self.matrix[:, 10]
        self.F4 = self.matrix[:, 11]
        self.F8 = self.matrix[:, 12]
        self.AF4 = self.matrix[:, 13]

    def plotRawChannel(self, channel, start, stop):
        """
        plot raw channel for trainingdatamacro object
        :param channel: channel to be plotted
        :param start: (in seconds) where in the recording to start plotting
        :param stop: (in seconds) where in the recording to stop plotting
        :return:
        """
        channel = channel[start*self.sampleRate:stop*self.sampleRate]

        # define time axis
        tAxis = np.arange(0, len(channel))  # create time axis w same length as the data matrix
        tAxis = tAxis / self.sampleRate  # adjust time axis to 256 sample rate

        # use eeg matrix as y axis
        yAxis = channel

        # plot figure
        plt.figure()
        plt.plot(tAxis, yAxis)
        plt.ylim(-1000, 1000)
        plt.show(block=True)
```

```
    def createChunks(self):
        """
        creates chunks that meet the threshold and backtrack criteria. Basically, if a certain
channel's voltage passes a
        certain threshold, a chunk of samples will be saved to self.trainingChunks
        :return:
        """
        self.nChunks = 0
        x, y = self.matrix.shape
        i = 0
        while i < len(self.F7):
            if ((abs(self.F7[i]) >= self.threshold) or (abs(self.AF3[i]) >= self.threshold) or
(abs(self.T7[i]) >= self.threshold)) and (i > eegData.backTrack) and (i < (len(self.F7) -
eegData.chunkSize)):
                chunkStart = i - eegData.backTrack
                chunkEnd = chunkStart + eegData.chunkSize
                # print(self.matrix[chunkStart:chunkEnd,:])
                # print(self.emotivChannels)
                self.trainingChunks.append(pandas.DataFrame(self.matrix[chunkStart:chunkEnd, :],
columns=self.emotivChannels))
                self.nChunks = self.nChunks + 1
                i = chunkEnd + 1
            i += 1

    def plotChunk(self, num):
        """
        plots a single chunk from self.trainingChunks (not curated). to be used in evalChunk
method
        """
        fig = plt.figure(num)
        plt.plot(self.trainingChunks[num].values)
        plt.xlabel('sample number ' + str(num) )
        plt.ylim(-1500, 1500)
        plt.pause(0.001)

        return fig

    def evalChunk(self):
        """
                evaluates chunks
                goes through self.matrix and picks out chunks for you to evaluate
                if you think the chunk is good, it will save it to self.curatedChunk list that
contains all good chunks

                :return:
        """
        self.curatedChunkCount = 0
        for i in range(0, len(self.trainingChunks)):
            fig = self.plotChunk(i)
            prompt = input('would you like to use sample number ' + str(i) + '? ')
            if prompt == 'y':
                self.curatedChunk.append(self.trainingChunks[i])
                self.curatedChunkCount += 1

            plt.close(fig)

    def saveChunksToCSV(self, subdir='smallChunks'):
        """
        saves curated chunks to csv
        :param obj:
        :return:
        """
```

```
        for i in range(len(self.curatedChunk)):
            self.curatedChunk[i].to_csv(os.path.join(parentDir, 'data', 'savedChunks', subdir,
self.tag + '_' + str(i) + '.csv'))

    def saveTrainingObject(self, filename, address=os.path.join(parentDir, 'data',
'savedTrainingObjects')):
        filehandle = open(os.path.join(address, filename), 'w')
        pickle.dump(self, filehandle)

    @staticmethod
    def loadFromTrainingObject(filename, address=os.path.join(parentDir, 'data',
'savedTrainingObjects')):
        file = open(os.path.join(address, filename), 'r')
        object = pickle.load(file)
        return object

    def plotRawEEG(self, matrix, offset=200):
        """
        note: the only difference between this and the parent method is that this one displays
the title of the thing
        being plotted
        :param matrix:
        :param offset: DC offset between eeg channels
        :return: plot with all 14
        """
        # define time axis
        tAxis = np.arange(0, len(matrix))  # create time axis w same length as the data matrix
        tAxis = tAxis / self.sampleRate  # adjust time axis to 256 sample rate

        # use eeg matrix as y axis
        yAxis = matrix + offset * 13

        # add offset to display all channels
        for i in range(0, len(matrix[0, :])):
            yAxis[:, i] -= offset * i

        # plot figure
        plt.figure()
        plt.plot(tAxis, yAxis)
        plt.title(self.tag)
        plt.ylim(-300, offset * 20)
        plt.legend(["EEG.AF3", "EEG.F7", "EEG.F3", "EEG.FC5", "EEG.T7", "EEG.P7", "EEG.O1",
                    "EEG.O2", "EEG.P8", "EEG.T8", "EEG.FC6", "EEG.F4", "EEG.F8", "EEG.AF4"],
                   loc='upper right')
        plt.xlabel('time')
        plt.show(block=True)
        # plt.pause(0.01)

# todo: rebuild tensor flow with AVX2 FMA for faster performance
```

music module

```
import mido
import time
from audiolazy.lazy_midi import str2midi
from MusEEG import parentDir, port, closePort, resetPort


class music:
    midiChannel = 0
    tempo = 120

    def set_tempo(self, tempo):
        self.tempo = tempo

    def panic(self):
        port.panic()

    def pause(self,nQuarterNotes):
        time.sleep((nQuarterNotes)*1/self.tempo*60)

class chord(music):
    def __init__(self, notelist, name):
        self.notelist = notelist
        self.name = name

    def playchord(self, vel=64, qtrnotes=2):
        self.play(vel)
        self.pause(qtrnotes)
        self.stop()


    def play(self, vel=64):
        for notes in self.notelist:
            msg = mido.Message('note_on', note=str2midi(notes), velocity=vel,
channel=self.midiChannel)
            port.send(msg)

    def stop(self):
        for notes in self.notelist:
            msg = mido.Message('note_off', note=str2midi(notes), channel=self.midiChannel)
            port.send(msg)

    def stopAllNotes(self):
        for notes in range(0,128):
            msg = mido.Message('note_off', note=notes, channel=self.midiChannel)
            port.send(msg)

    def arpeggiate(self, notelength, vel, numTimes):
        for i in range(numTimes):
            for notes in self.notelist:
                msg = mido.Message('note_on', note=str2midi(notes), velocity=vel,
channel=self.midiChannel)
                port.send(msg)
                self.pause(notelength)
                self.stop()

class melody(music):
    currentTime = 0

    def __init__(self, midiname):
        self.midi = mido.MidiFile(midiname)
```

```
        self.track = mido.MidiTrack
        self.midi.tracks.append(self.track)

    def addnote(self, note, duration, vel=64):
        msgon = mido.Message('note_on', note=str2midi(note), velocity=vel,
channel=self.midiChannel, time=self.currentTime)
        port.send(msgon)
        self.pause(mido.second2tick(duration/60*self.tempo))
        msgoff = mido.Message('note_off', note=str2midi(note), velocity=vel,
channel=self.midiChannel, time=self.currentTime+duration)
        port.send(msgoff)


#todo: add melody class
```

classifier module

```
import os
import platform
import pandas
import numpy as np
import tensorflow as tf
from tensorflow import keras
from keras import regularizers
from MusEEG import parentDir
from sklearn.metrics import confusion_matrix

class classifier:
    hiddenNeurons = 20
    numberOfTargets = 10
    inputShape = 350

    def __init__(self):
        print('welcome')

    def loadTrainingData(self, percentTrain=0.75,
                         address=os.path.join(parentDir, 'data', 'training'),
                         subdir='bigChunks',
                         inputFilename='inputs.csv',
                         targetFilename='targets.csv'):
        inputsAll = pandas.read_csv(os.path.join(address, subdir,  inputFilename)).values
        targetsAll = pandas.read_csv(os.path.join(address, subdir,  targetFilename)).values
        # use index from 1 on bc index 0 is just a counter for some reason.
        inputsAll[:, 0] = targetsAll[:, 1]
        # first column of inputsAll will now be the targets (sorry programming gods, I'm going
crazy over this one)
        trainingData = pandas.DataFrame(inputsAll)
        #shuffle the data!!!
        trainingData = trainingData.reindex(np.random.permutation(trainingData.index))
        trainingData = trainingData.values

        #slice it up, baby
        slice = round(len(trainingData[:, 0])*percentTrain)
        train_inputs = trainingData[0:slice, 1:]
        train_targets = trainingData[0:slice, 0]
        test_inputs = trainingData[slice:, 1:]
        test_targets = trainingData[slice:, 0]

        return train_inputs, train_targets, test_inputs, test_targets

    # build the model
    def build_model(self, inputShape, hiddenNeurons, numberOfTargets, hiddenActivation='relu',
                    outputActivation='softmax', regularization='l2_l2', optimizer='adam',
                    loss='sparse_categorical_crossentropy'):
        if regularization == 'l1':
            reg = regularizers.l1(0.001)
            self.model = keras.Sequential([
                keras.layers.Dense(hiddenNeurons,
                                   activation=hiddenActivation,
                                   activity_regularizer=reg,
                                   input_dim=inputShape),
                keras.layers.Dense(numberOfTargets, activation=outputActivation),
            ])
        elif regularization == 'l2':
            reg = regularizers.l2(0.001)
            self.model = keras.Sequential([
                keras.layers.Dense(hiddenNeurons,
                                   activation=hiddenActivation,
```

```
                                activity_regularizer=reg,
                                input_dim=inputShape),
                    keras.layers.Dense(numberOfTargets, activation=outputActivation),
                ])
            elif regularization == 'l1_l2':
                reg = regularizers.l1_l2(0.001)
                self.model = keras.Sequential([
                    keras.layers.Dense(hiddenNeurons,
                                activation=hiddenActivation,
                                activity_regularizer=reg,
                                input_dim=inputShape),
                    keras.layers.Dense(numberOfTargets, activation=outputActivation),
                ])
            else:
                self.model = keras.Sequential([
                    keras.layers.Dense(hiddenNeurons, activation=hiddenActivation,
                                input_dim=inputShape),
                    keras.layers.Dense(numberOfTargets, activation=outputActivation)])

            self.model.compile(optimizer=optimizer,
                               loss=loss,
                               metrics=['accuracy'])
            self.hiddenNeurons = hiddenNeurons
            self.numberOfTargets = numberOfTargets
            self.inputShape = inputShape
            return self.model

        # train the model
        def train_model(self, train_inputs, train_targets, nEpochs, verbose=0):
            self.model.fit(train_inputs, train_targets, epochs=nEpochs, verbose=verbose)

        def evaluate_model(self, test_inputs, test_targets, verbose=2):
            test_loss, test_acc = self.model.evaluate(test_inputs, test_targets, verbose)
            print('\nTest accuracy:', test_acc)
            return test_acc

        def print_confusion(self, test_inputs, test_targets):
            prediction = self.model.predict(test_inputs)
            prediction = np.array([np.argmax(row) for row in prediction])
            cm = confusion_matrix(test_targets, prediction)
            print(cm)
            return cm

        def classify(self, inputVector):
            prediction = self.model.predict(inputVector)
            output = np.argmax(prediction)
            return output

        def clear(self):
            keras.backend.clear_session()

        def savemodel(self, filename, address=os.path.join(parentDir, 'data', 'savedModels')):
            self.model.save(os.path.join(address, filename), save_format='tf')

        def loadmodel(self, filename, address=os.path.join(parentDir, 'data', 'savedModels')):
            """
            load a saved keras model
            :param filename: name of the savedModel
            :param address: address (relative to the parent directory) where your model is stored.
    defaults to /data/SavedModels
            :return:
            """
            self.model = keras.models.load_model(os.path.join(address, filename))
```

```
        if platform.uname()[1] == 'raspberrypi':
            litemodel = self.createLiteModel(self.model)
            self.model = litemodel

    def createLiteModel(self, model):
        """
        create tflite model from saved keras model
        :param filename: name of the savedModel
        :param address: address (relative to the parent directory) where your model is stored.
defaults to /data/SavedModels
        :return: tensorflow lite model
        """
        converter = tf.lite.TFLiteConverter.from_keras_model(model)
        return converter.convert()
```

cerebro module

```
from .classifier import *
from .eegData import *
from .music import *
from .client import *

from MusEEG import parentDir, resetPort, closePort
import threading


class cerebro:
    """
    hello message to display in UI
    """
    demomsg = (
        'Hello! welcome to the MusEEG demo. This demo will: \n'
        '- send a pre-recorded brain signal of your choice when you click on any of the gesture
buttons\n'
        '- process it using a 4-level, db2 wavelet transform\n'
        '- extract the first four statistical moments of the wavelet decompositions (mean,
variance, skewness, kurtosis)\n'
        '- classify it using a deep neural network\n'
        '- using the results from the DNN, play the chord that is referenced to the gesture
using MIDI\n'
        '- to change a chord, press the "update chord dictionary" button after youve changed the
notes\n')
    eeg = eegData()

    gestures = ['smile', 'bitelowerlip', 'eyebrows', 'hardblink', 'lookleft', 'lookright',
                'neutral', 'scrunch', 'tongue']

    """
    chord objects are defined here. the chord() class takes any set of notes as an input.
    """
    cmaj7sharp11add13 = chord(['C4', 'E4', 'G4', 'B4', 'D5', 'F#4', 'A5'],
                              name='cmaj7sharp11add13')
    fminmaj7 = chord(['F4', 'Ab4', 'C5', 'E5'], name='fminmaj7')
    fmaj7 = chord(['F4', 'A4', 'C5', 'E5', 'G5'], name='fmaj7')
    ab69 = chord(['Ab4', 'C5', 'F5', 'Bb5', 'C6'], name='ab69')
    dmin7b5 = chord(['D4', 'F4', 'Ab4', 'C5', 'E5'], name='dmin7b5')
    g7b9 = chord(['G4', 'B4', 'D5', 'F5', 'Ab5'], name='g7b9')
    c5 = chord(['C3', 'G3'], name='c5')
    noChord = chord([], name='nochord')
    polychordcde = chord(
        ['C3', 'E3', 'G3', 'D4', 'F#4', 'A4', 'E5', 'G#5', 'B5'], name='E/D/C')  # todo: add a
polychord(chord) method
    dbmaj7 = chord(['Db4', 'F4', 'Ab4', 'C5', 'Eb5'], name='dbmaj7')
    margaretsmagicchord = chord(['D4', 'F4', 'A#4'], name='margschord')

    defaultchordlist = [cmaj7sharp11add13.notelist, fminmaj7.notelist,
                        fmaj7.notelist, ab69.notelist, dmin7b5.notelist,
                        c5.notelist, noChord.notelist,
                        polychordcde.notelist, dbmaj7.notelist]

    """
    this dictionary is where chords are referenced to facial gestures.
    """

    def __init__(self):
        #default mididict. it will be updated everytime the user presses the update chord button
        self.mididict = {'smile': self.cmaj7sharp11add13,
                         'bitelowerlip': self.fmaj7,
```

```
                        'hardblink': self.fminmaj7,
                        'eyebrows': self.ab69,
                        'lookleft': self.g7b9,
                        'lookright': self.c5,
                        'neutral': self.noChord,
                        'scrunch': self.polychordcde,
                        'tongue': self.dbmaj7}

        # open and reset midiport
        resetPort()

        # list of gestures to be used in classifier
        self.gestures = ['smile', 'bitelowerlip', 'eyebrows', 'hardblink', 'lookleft',
'lookright',
                         'neutral', 'scrunch', 'tongue']

        # load the DNN classifier (bigbrain for whole eeg chunks, small brain for small chunks)
        self.bigBrain = classifier()
        self.bigBrain.loadmodel(os.path.join(parentDir, 'data', 'savedModels', 'bigBrain_v2'))
        self.smallBrain = classifier()
        self.smallBrain.loadmodel(os.path.join(parentDir, 'data', 'savedModels',
'smallBrain_v1'))

        # define chords and tempo to be used
        music.tempo = 60  # bpm
        music.midiChannel = 0  # add 1

    def setupClient(self):
        self.client = client()
        self.client.setup()

    def updateChordList(self, chordlistlist):
        for c in chordlistlist:
            index = chordlistlist.index(c)
            gestureBeingDefined = self.gestures[index]
            self.mididict[gestureBeingDefined] = chord(notelist=chordlistlist[index],
name=gestureBeingDefined)
            print(self.mididict)

    def loadFromDataSet(self, name):
        # subdirectory where sample chunks are located and load a random chunk from trianing
dataset
        SUBDIR = os.path.join('bigChunks', 'hugo_facialgestures')
        self.eeg.loadChunkFromTraining(subdir=SUBDIR, filename=name + '_' +
str(np.random.randint(0, 60)) + '.csv')

    def processAndPlay(self, arp, tempo, arpDurationFromGUI, noteDurationFromGUI):
        print('performing wavelet transform')
        brainInput = self.eeg.process()

        self.arpDurationFromGUI = arpDurationFromGUI
        self.noteDurationFromGUI = noteDurationFromGUI

        # classify facial gesture in DNN
        brainOutput = self.bigBrain.classify(brainInput.reshape(1, 350))
        print('the neural network has taken the brain signal and classified it.')
        self.gestureResult = self.gestures[brainOutput]
        print('classification result: ' + self.gestureResult)

        # refer classification to midi dictionary and refer chord object to musician
        musician = self.mididict[self.gestureResult]
        musician.set_tempo(tempo=tempo)
```

```
    #with threading
    musicianProcess = threading.Thread(target=self.perform, args=[musician, arp])
    musicianProcess.start()

def perform(self, musician, arp):
    if arp:
        print('arpeggiate!')
        musician.arpeggiate(notelength=self.arpDurationFromGUI, vel=30, numTimes=8)

    else:
        musician.panic()
        musician.playchord(qtrnotes=self.noteDurationFromGUI, vel=30)
```

client module

```
from MusEEG import eegData
from MusEEG import TrainingDataMacro
import numpy as np
from numpy import array
import threading
import time
# -*- coding: utf8 -*-
#
# Cykit Example TCP - Client
# author: Icannos
# modified for MusEEG by: hugo flores garcia
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import matplotlib.animation as animation

import socket
import queue

class client:
    BUFFER_SIZE = eegData.chunkSize
    host = "127.0.0.1"
    port = 5555
    sampleRate = eegData.sampleRate
    streamIsSimulated = False

    # Named fields according to Warren doc !
    FIELDS = {"COUNTER": 0, "DATA-TYPE": 1, "AF3": 4, "F7": 5, "F3": 2, "FC5": 3, "T7": 6, "P7":
7, "O1": 8, "O2": 9,
              "P8": 10, "T8": 11, "FC6": 14, "F4": 15, "F8": 12, "AF4": 13, "DATALINE_1": 16,
"DATALINE_2": 17}

    def data2dic(self, data):
        field_list = data.split(b',')

        if len(field_list) > 17:
            return {field: float(field_list[index]) for field, index in self.FIELDS.items()}
        else:
            return -1

    def setup(self):
        self.s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.s.connect((self.host, self.port))
        self.s.send(b"\r\n")

        # To read the header msgs about cykit etc...
        self.s.recv(168, socket.MSG_WAITALL)

        # Local buffer to store parts of the messages
        self.buffer = b''

        # If when when split by \r, \r was the last character of the message, we know that we
have to remove \n from
        # the begining of the next message
        self.remove_newline = False

    def stream(self):
        self.q = queue.LifoQueue()
        def workerjob():
            try:
                while True:
```

```
                    # We read a chunk
                    data = self.s.recv(self.BUFFER_SIZE)

                    # If we have to remove \n at the begining
                    if self.remove_newline:
                        data = data[1:]
                        self.remove_newline = False

                    # Splitting the chunk into the end of the previous message and the begining
of the next message
                    msg_parts = data.split(b'\r')

                    # If the second part ends with nothing when splitted we will have to remove
\n next time
                    if msg_parts[-1] == b'':
                        self.remove_newline = True
                        # Therefore the buffer for the next step is empty
                        self.n_buffer = b''
                    else:
                        # otherwise we store the begining of the next message as the next buffer
                        self.n_buffer = msg_parts[-1][1:]

                    # We interprete a whole message (begining from the previous step + the end
                    fields = self.data2dic(self.buffer + msg_parts[0])

                    # We setup the buffer for next step
                    self.buffer = self.n_buffer

                    # Print all channel
                    self.q.put(fields, block=False)
            except Exception:
                self.q.join()
                self.s.close()

        worker = threading.Thread(target=workerjob, args=())
        worker.setDaemon(True)
        worker.start()

    def getChunk(self, chunkSize=eegData.chunkSize):
        chunk = []
        while len(chunk) < chunkSize:
            try:
                data = self.q.get()
                ## this conditional is to differentiate between a simulated stream and the
actual server
                if not self.streamIsSimulated:
                    chunk.append([data["AF3"], data["F7"], data["F3"], data["FC5"], data["T7"],
data["P7"], data["O1"],
                                  data["O2"], data["P8"], data["T8"], data["FC6"], data["F4"],
data["F8"], data["AF4"]])
                else:
                    chunk.append(data + 4100)
            except TypeError:
                pass

        return array(chunk) - 4100

    def simulateStream(self, gesture):
        self.streamIsSimulated = True
        eeg = TrainingDataMacro()
        eeg.importCSV(subdir='hugo_facialgestures', filename=gesture+'.csv', tag=gesture)
        self.q = queue.LifoQueue()
        def worker():
```

```
            for i in range(0,len(eeg.matrix)):
                packet = eeg.matrix[i][:]
                self.q.put(item=packet)
                time.sleep(1/eegData.sampleRate/10)

        simulationWorker = threading.Thread(target=worker)
        simulationWorker.setDaemon(True)
        simulationWorker.start()



if __name__ == "__main__":
    client = client()
    client.setup()
```

demoApp

```
import tkinter as tk
from MusEEG import cerebro

import matplotlib
matplotlib.use("TkAgg")
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg

buttonRow = 2

#instantiate a cerebro object
cerebro = cerebro()

class demoApp(tk.Frame):
    availableGestures = list(cerebro.mididict.keys())
    defaultGesture = availableGestures[6]
    gestureButtonStartRow=0
    buttonRow=11

    def __init__(self, master=None):
        super().__init__(master)
        self.master = master
        self.pack()
        self.create_widgets()

    def welcomeMessage(self):
        self.welcomemsg = tk.Message(self, text=cerebro.demomsg, relief=tk.RIDGE)
        self.welcomemsg.anchor('nw')
        # self.welcomemsg.pack(side="left")
        self.welcomemsg.grid(row=11, column=0, rowspan=5, columnspan=2, padx=5, pady=5)

    def tempoBox(self):
        self.templbl = tk.Label(self, text='tempo (bpm)').grid(row=self.buttonRow+1, column=2)
        self.tempobx = tk.Entry(self)
        self.tempobx.insert(10, '120')
        self.tempobx.grid(row=self.buttonRow, column=2)

    def arpeggioDuration(self):
        self.arpeglbl = tk.Label(self, text='arpeggio note duration (in qtr
notes)').grid(row=self.buttonRow+1, column=3)
        self.arpegbx = tk.Entry(self)
        self.arpegbx.insert(10, '0.5')
        self.arpegbx.grid(row=self.buttonRow, column=3)

    def chordDuration(self):
        self.sustainlbl = tk.Label(self, text='sustain duration (in qtr
notes)').grid(row=self.buttonRow + 3, column=3)
        self.sustainbx = tk.Entry(self)
        self.sustainbx.insert(10, '8')
        self.sustainbx.grid(row=self.buttonRow+2, column=3)

    def buttonProcessAndSend(self):
        def processFunction():
            cerebro.processAndPlay(arp=self.arpVar.get(), tempo=int(self.tempobx.get()),
                                   arpDurationFromGUI=float(self.arpegbx.get()),
                                   noteDurationFromGUI=float(self.sustainbx.get()))

            # update plot window todo: make it not have to redefine entire plot window for
faster processing
            self.canvas.flush_events()
            self.canvas = FigureCanvasTkAgg(cerebro.eeg.plotWavelets(1), self)
            self.canvas.draw()
```

```
            self.say_hi()
            self.canvas.get_tk_widget().grid(row=0, column=2, rowspan=11, columnspan=3, padx=5,
pady=5)
            # self.wavcanvas.flush_events()
            # self.wavcanvas = FigureCanvasTkAgg(cerebro.eeg.plotWavelets(1), self)
            # self.wavcanvas.draw()
            # self.say_hi()
            # self.wavcanvas.get_tk_widget().grid(row=0, column=6, rowspan=11, columnspan=3,
padx=5, pady=5)

        self.processAndSendBttn = tk.Button(self, command=processFunction)
        self.processAndSendBttn["text"] = "Process and Send to Musician"
        self.processAndSendBttn.grid(row=self.buttonRow+3, column=2, padx=5, pady=5)

    def gestureButtons(self):
        def gestBttnCommand(gestureToLoad):
            #load from dataset
            cerebro.loadFromDataSet(name=gestureToLoad)



        self.gesturebttn = list()
        for GESTURES in cerebro.gestures:
            index = cerebro.gestures.index(GESTURES)
            self.gesturebttn.append(tk.Button(self, text=GESTURES, command=lambda
GESTURES=GESTURES: gestBttnCommand(GESTURES)))
            self.gesturebttn[index].grid(row=self.gestureButtonStartRow+index, column=0)

    def plotWindow(self):
        self.canvas = FigureCanvasTkAgg(cerebro.eeg.plotWavelets(1), self)
        self.canvas.draw()
        # self.canvas.get_tk_widget().pack(side="right", expand=True)
        self.canvas.get_tk_widget().grid(row=0, column=2, rowspan=11, columnspan=3, padx=5,
pady=5)

    # def wavPlotWindow(self):
    #     self.wavcanvas = FigureCanvasTkAgg(cerebro.eeg.plotWavelets(1), self)
    #     self.wavcanvas.draw()
    #     # self.canvas.get_tk_widget().pack(side="right", expand=True)
    #     self.wavcanvas.get_tk_widget().grid(row=0, column=6, rowspan=11, columnspan=3, padx=5,
pady=5)

    def checkboxArpeggiate(self):
        self.arpVar = tk.BooleanVar()
        self.checkboxArp = tk.Checkbutton(self, text="arpeggiate?", variable=self.arpVar)
        self.checkboxArp.grid(row=self.buttonRow+2, column=2, padx=5, pady=5)

    def defineChordEntry(self):
        def listToString(s):
            # initialize an empty string
            str1 = " "
            # return string
            return (str1.join(s))

        def stringToList(s):
            return s.split()

        self.chordEntryLbl = list()
        self.chordEntrybx = list()

        self.chordlist = cerebro.defaultchordlist

        for gesture in cerebro.gestures:
```

```
            index = cerebro.gestures.index(gesture)
            # self.chordEntryLbl.append(tk.Label(self, text=gesture))
            # self.chordEntryLbl[index].grid(row=self.gestureButtonStartRow+index, column=0)

            #create entry box and set defaultchordlist as default
            self.chordEntrybx.append(tk.Entry(self))
            self.chordEntrybx[index].insert(0, listToString(cerebro.defaultchordlist[index]))
            self.chordEntrybx[index].grid(row=self.gestureButtonStartRow+index, column=1)

        #retrieve chords from list
        def defineChordList():
            for items in range(len(self.chordlist)):
                self.chordlist[items] = stringToList(self.chordEntrybx[items].get())

            cerebro.updateChordList(self.chordlist)

        #button to update chords
        self.updateChords = tk.Button(self, command=defineChordList)
        self.updateChords["text"] = "update chord dictionary"
        #place the button under all the entry boxes
        self.updateChords.grid(row=self.gestureButtonStartRow+len(self.chordlist)+1, column=1)



    def classificationResult(self):
        permanentText = 'classification result: '
        self.classificationResultVar = tk.StringVar(self)
        self.classificationResultVar.set('none')
        self.classPrint = tk.Label(self, text=permanentText+self.classificationResultVar.get())
        self.classPrint.grid(row=self.buttonRow+3, column=2)


    def create_widgets(self):
        self.winfo_toplevel().title("MusEEG")

        self.welcomeMessage()
        self.tempoBox()
        self.arpeggioDuration()
        self.chordDuration()
        self.gestureButtons()
        self.plotWindow()
        # self.wavPlotWindow()
        self.checkboxArpeggiate()
        # self.classificationResult()
        self.buttonProcessAndSend()
        self.defineChordEntry()



    def say_hi(self):
        print("\nhi there, everyone!")


cerebro.loadFromDataSet(name=demoApp.defaultGesture)

root = tk.Tk()
app = demoApp(master=root)

#todo: the app isnt quitting properly and it ruins ur computer
while True:
    try:
        app.mainloop()
        break
```

```
except UnicodeDecodeError:
    pass
```

example.py

```
import os
import MusEEG
from MusEEG import eegData, classifier
from MusEEG.music import chord
import numpy as np
import threading

#todo: this is currently not working, due to an update to the perform and chord methods.

# open and reset midiport
MusEEG.resetPort()

# list of gestures to be used in classifier
gestures = ['smile', 'bitelowerlip', 'eyebrows', 'hardblink', 'lookleft', 'lookright',
            'neutral', 'scrunch', 'tongue']

# load the DNN classifier (bigbrain for whole eeg chunks)
brain = classifier()
brain.loadmodel(os.path.join(MusEEG.parentDir, 'data', 'savedModels', 'bigBrain_v2'))

# define chords and tempo to be used
chord.tempo = 60  # bpm
chord.midiChannel = 0  # add 1

"""
chord objects are defined here. the chord() class takes any set of notes as an input.
"""
cmaj7sharp11add13 = chord(['C4', 'E4', 'G4', 'B4', 'D5', 'F#4', 'A5'], name='cmaj7sharp11add13')
fminmaj7 = chord(['F4', 'Ab4', 'C5', 'E5'], name='fminmaj7')
fmaj7 = chord(['F4', 'A4', 'C5', 'E5', 'G5'], name='fmaj7')
ab69 = chord(['Ab4', 'C5', 'F5', 'Bb5', 'C6'], name='ab69')
dmin7b5 = chord(['D4', 'F4', 'Ab4', 'C5', 'E5'], name='dmin7b5')
g7b9 = chord(['G4', 'B4', 'D5', 'F5', 'Ab5'], name='g7b9')
c5 = chord(['C3', 'G3'], name='c5')
noChord = chord([], name='nochord')
polychordcde = chord(
    ['C3', 'E3', 'G3', 'D4', 'F#4', 'A4', 'E5', 'G#5', 'B5'], name='E/D/C')  # todo: add a
polychord(chord) method
dbmaj7 = chord(['Db4', 'F4', 'Ab4', 'C5', 'Eb5'], name='dbmaj7')
margaretsmagicchord = chord(['D4', 'F4', 'A#4'], name='margschord')

chordlist = [cmaj7sharp11add13.name, fminmaj7.name, fmaj7.name, ab69.name, dmin7b5.name,
c5.name, noChord.name,
             polychordcde.name, dbmaj7.name, margaretsmagicchord.name]
# refer gestures to chords
"""
this dictionary is where chords are referenced to facial gestures.
"""
mididict = {'smile': cmaj7sharp11add13,
            'bitelowerlip': fmaj7,
            'hardblink': fminmaj7,
            'eyebrows': ab69,
            'lookleft': g7b9,
            'lookright': c5,
            'neutral': noChord,
            'scrunch': polychordcde,
            'tongue': dbmaj7}

mididictstr = dict(zip(gestures, chordlist))

def dothething(eeg, verbose=True, arp=None):
```

```
    if verbose:
        # plot raw eeg data
        eeg.plotRawEEG(title=eeg.filename)

        # process eegdata: wavelet transform, statistical extraction
        print('performing wavelet transform')
        brainInput = eeg.process()

        # plot wavelet transform of channel 2
        eeg.plotWavelets(channel=1)

        # classify facial gesture in DNN
        brainOutput = brain.classify(brainInput.reshape(1, 350))
        print('\nthe neural network has taken the brain signal and classified it.')
        gestureResult = gestures[brainOutput]
        print('classification result: ' + gestureResult + '\n')

        # refer classification to midi dictionary and refer chord object to musician
        musician = mididict[gestureResult]

        t1 = threading.Thread(target=musician.playchord(), args=[musician, arp])
        t1.start()


    if not verbose:
        print('performing wavelet transform')
        brainInput = eeg.process()

        # classify facial gesture in DNN
        brainOutput = brain.classify(brainInput.reshape(1, 350))
        print('\nthe neural network has taken the brain signal and classified it.')
        gestureResult = gestures[brainOutput]
        print('classification result: ' + gestureResult)

        # refer classification to midi dictionary and refer chord object to musician
        musician = mididict[gestureResult]

        t1 = threading.Thread(target=musician.playchord, args=[musician, arp])
        t1.start()


def demoComponent():
    firstTime = True
    while (True):
        try:
            # prompt user what gesture they'd like to demo and load a random sample from the
dataset to test
            eeg = eegData()
            print(
                '\nHello! welcome to the MusEEG demo. This demo will send a pre-recorded brain
signal, classify it into\n'
                'a facial gesture using a deep learning algorithm, and the turn it into a set of
pre-referenced\n'
                ' chords. if you are familiar with music and prorgamming, feel free to edit the
chord objects\n '
                'and the chord-facial gesture dictionary in the cerebro.py file\n\n')
            print('NOTE: make sure to run this script BEFORE you open your DAW/virtual
instrument due to MIDI port '
                  'reset purposes\n')
            print('available gestures: ')
            print(gestures)
            name = input('\n\nwhat gesture would you like to send to the neural network? please
enter one of the '
```

```
                    'gestures available above:  ')
            if name not in mididict:
                print('this gesture was not found. try again')
                continue

            # subdirectory where sample chunks are located and load a random chunk from trianing
dataset
            SUBDIR = os.path.join('bigChunks', 'hugo_facialgestures')
            eeg.loadChunkFromTraining(subdir=SUBDIR, filename=name + '_' +
str(np.random.randint(0, 60)) + '.csv')

            # only show the verbose version once
            if firstTime == True:
                arp = input('play the chords straight or arpeggiate?')
                dothething(eeg, verbose=True, arp=arp)
                firstTime = False
            else:
                dothething(eeg, verbose=False)

            cont = input('would you like to try another eeg signal? (y/n)')
            if cont == 'y':
                continue
            elif cont == 'n':
                break
            else:
                print('invalid command. exiting anyway')
                break

        except KeyboardInterrupt:
            break


demoComponent()
MusEEG.closePort()
```

mastermind.py

```
from MusEEG import eegData, classifier, client, cerebro
import matplotlib.pyplot as plt
import numpy as np
import threading

"""
this is set up rn to simulate an eeg stream, instead of getting data from the client
"""

cerebro = cerebro()

client = client()
# client.setup()
# client.stream()
#### TEST TEST
testgesture = 'scrunch'
numberofErrors = 0
client.simulateStream(testgesture)
#### TEST TEST

figure = plt.figure()

def mainProcessor():


    stopChunkGetter=False
    def getMoreChunks(chunk):
        while len(chunk) < eegData.chunkSize:
            chunk.extend(list(client.getChunk(chunkSize=eegData.smallchunkSize)))
            if stopChunkGetter:
                break

    while (True):
        try:
            activeGesture = False
            while not activeGesture:
                eeg = eegData()
                eeg.chunk = client.getChunk(chunkSize=eegData.smallchunkSize)
                fullchunk = list(eeg.chunk)
                chunkGetter = threading.Thread(target=getMoreChunks, args=(fullchunk,))
                chunkGetter.start()

                brainInput = eeg.process()
                brainOutput = cerebro.smallBrain.classify(brainInput.reshape(1, 350))

                if brainOutput == 0:
                    print('gesture found')
                    activeGesture = True
                    stopChunkGetter = False
                    chunkGetter.join()
                else:
                    print('.')
                    activeGesture = False
                    stopChunkGetter = True
                    chunkGetter.join()

            eeg = eegData()

            eeg.chunk = np.array(fullchunk)
            # eeg.plotRawEEG(figure=figure)
```

```
            if len(eeg.chunk) != eeg.chunkSize:
                raise RuntimeError('this chunk wasn\'t 384 samples. something went wrong')

            def processAndPlay(eeg):
                # print('performing wavelet transform')
                brainInput = eeg.process()

                # classify facial gesture in DNN
                brainOutput = cerebro.bigBrain.classify(brainInput.reshape(1, 350))
                # print('\nthe neural network has taken the brain signal and classified it.')
                gestureResult = cerebro.gestures[brainOutput]

                ####### TEST TEST
                global numberofErrors
                global testgesture
                if gestureResult is not (testgesture or 'neutral'):
                    numberofErrors = numberofErrors + 1
                    print(numberofErrors)
                ####### TEST TEST

                print('classification result: ' + gestureResult)

                resultingChord = cerebro.mididict[gestureResult]

                resultingChord.playchord()

            processor = threading.Thread(target=processAndPlay, args=(eeg,))
            processor.start()


        except KeyboardInterrupt:
            break

if __name__=="__main__":
    mainProcessor()
```

optimizeClassifier.py

```
import os
import MusEEG
import pandas as pd

brain = MusEEG.classifier()
train_inputs, train_targets, test_inputs, test_targets =
brain.loadTrainingData(percentTrain=0.75)

#define the different parameters to be tried
results = []
hiddenActivations = ['relu', 'sigmoid', 'tanh', 'elu' ]
outputActivations = ['softmax']
hiddenNeurons = range(50, 250, 25)
regularizers = ['l1', 'l2', 'l1_l2', 'no']
losses = ['sparse_categorical_crossentropy']

#perform exhaustive search of different combinations of network architectures
for a in hiddenActivations:
    for n in hiddenNeurons:
        for r in regularizers:
            for l in losses:
                brain.build_model(brain.inputShape, hiddenNeurons=n, hiddenActivation=a,
                                  numberOfTargets=(max(train_targets) + 1), regularization=r,
loss=l)
                name = a + '/ act ' + str(n) + ' neurons/ ' + \
                       r + ' reg/ ' + l + ' losses '
                hist = brain.train_model(train_inputs, train_targets, 80, verbose=0)
                test_accuracy = brain.evaluate_model(test_inputs, test_targets)
                brain.clear()
                results.append([name, test_accuracy])

print('stop')
print(results)

#save results
results = pd.DataFrame(results)
results.to_csv(os.path.join(MusEEG.parentDir, 'data', 'ClassifierOptimizations',
'forSmallChunks.csv'))
results.plot()
print('hi')

#it appears as if using elu, 175 neurons, l1_l2 regulation and sparse categorical crossentropy
losses is the best, with 93%
```

processTrainingData.py

```
# script meant to extract features from curated training data and put features into an input
vector for ANN as well as create
# a target vector for labels

import os
import sys

import MusEEG
from MusEEG import eegData
import pandas as pandas
import numpy as np


def createTargetVector(objarray, *argv):
    labels = []
    targets = [0 for row in range(len(objarray[0][:]) * len(objarray))]
    index = 0
    for i in range(0, len(objarray)):
        print(objarray[i][0].filename)
        for j in range(0, len(objarray[i])):
            labels.append(objarray[i][j].filename)
            for arg in argv:
                if arg in objarray[i][j].filename:
                    targets[index] = i
                    index = index + 1
    return targets


# create object lists, for easier handling
smile = [eegData() for i in range(0, 60)]
biteLowerLip = [eegData() for i in range(0, 60)]
eyebrows = [eegData() for i in range(0, 60)]
hardBlink = [eegData() for i in range(0, 60)]
lookLeft = [eegData() for i in range(0, 60)]
lookRight = [eegData() for i in range(0, 60)]
neutral = [eegData() for i in range(0, 60)]
scrunch = [eegData() for i in range(0, 60)]
tongue = [eegData() for i in range(0, 60)]

# load chunks for each of the objects
for i in range(0, 60):
    smile[i].loadChunkFromTraining('smile_' + str(i) + '.csv')
    biteLowerLip[i].loadChunkFromTraining('bitelowerlip_' + str(i) + '.csv')
    eyebrows[i].loadChunkFromTraining('eyebrows_' + str(i) + '.csv')
    lookLeft[i].loadChunkFromTraining('lookleft_' + str(i) + '.csv')
    lookRight[i].loadChunkFromTraining('lookright_' + str(i) + '.csv')
    neutral[i].loadChunkFromTraining('neutral_' + str(i) + '.csv')
    scrunch[i].loadChunkFromTraining('scrunch_' + str(i) + '.csv')
    tongue[i].loadChunkFromTraining('tongue_' + str(i) + '.csv')
    hardBlink[i].loadChunkFromTraining('hardblink_' + str(i) + '.csv')

# create single list with all of other gesture lists.
gestures = [smile, biteLowerLip, eyebrows, hardBlink, lookLeft, lookRight, neutral, scrunch,
tongue]
targets = createTargetVector(gestures, 'smile', 'bitelowerlip', 'eyebrows', 'hardblink',
'lookleft', 'lookright',
                             'neutral', 'scrunch', 'tongue')


inputs = np.ndarray([540, 350])
inputindex = 0
```

```
for i in range(0, len(gestures)):
    for j in range(0, len(gestures[0])):
        gestures[i][j].cutChunk()
        gestures[i][j].wavelet()
        gestures[i][j].extractStatsFromWavelets()
        gestures[i][j].flattenIntoVector()
        inputs[inputindex][:] = gestures[i][j].inputVector
        inputindex = inputindex + 1

print(targets)

inputs = pandas.DataFrame(inputs)
targets = pandas.DataFrame(targets)

inputs.to_csv(os.path.join(MusEEG.parentDir, 'data', 'training', 'smallChunks', 'inputs.csv'))
targets.to_csv(os.path.join(MusEEG.parentDir, 'data', 'training', 'smallChunks', 'targets.csv'))
```

saveModels.py

```
import MusEEG
import pandas
import numpy as np

save = False

brain = MusEEG.classifier()
train_inputs, train_targets, test_inputs, test_targets =
brain.loadTrainingData(subdir='bigChunks', percentTrain=0.75)
brain.build_model(inputShape=brain.inputShape,
                  hiddenNeurons=175,
                  hiddenActivation='elu',
                  numberOfTargets=max(train_targets) + 1,
                  regularization='l1_l2 ',
                  loss='sparse_categorical_crossentropy')

brain.train_model(train_inputs, train_targets, nEpochs=80, verbose=2)
brain.evaluate_model(test_inputs, test_targets)

cm = brain.print_confusion(test_inputs, test_targets)

cmdataframe = pandas.DataFrame(cm)

print('hi')
if save:
    prompt = input('should we save this model, ye great master?')
    if prompt == 'yes':
        name = input('what should we name it, ye great master?')
        brain.savemodel(name)
    else:
        print('oh ok')
```

sortTrainingData.py

```
"""
sortTrainingData
This is meant for u to look through each of the really long CSV files and cut the time series
EEG data into chunks that contain the desired gesture
"""
import os
import MusEEG
from MusEEG import TrainingDataMacro

#this is where the your training data is stored
subdir = 'hugo_facialgestures'

smile = TrainingDataMacro()
biteLowerLip = TrainingDataMacro()
hardBlink = TrainingDataMacro()
lookLeft = TrainingDataMacro()
lookRight = TrainingDataMacro()
neutral = TrainingDataMacro()
scrunch = TrainingDataMacro()
tongue = TrainingDataMacro()
eyebrows = TrainingDataMacro()

# create importCSV from gesturedata
smile.importCSV(subdir, "smile.csv", "smile")
biteLowerLip.importCSV(subdir, "bitelowerlip.csv", 'bitelowerlip')
hardBlink.importCSV(subdir, "hardblink.csv", 'hardblink')
lookLeft.importCSV(subdir, "lookLeft.csv", 'lookleft')
```

```
lookRight.importCSV(subdir, "lookRight.csv", 'lookright')
neutral.importCSV(subdir, "neutral.csv", 'neutral')
scrunch.importCSV(subdir, "scrunch.csv", 'scrunch')
tongue.importCSV(subdir, "tongue.csv", 'tongue')
eyebrows.importCSV(subdir, "eyebrows.csv", 'eyebrows')


# #sort through the data and cut chunks whenever it reaches the threshold
def evalAndPrep(obj):
    obj.plotRawEEG(obj.matrix, 400)
    obj.createChunks()
    obj.evalChunk()
    obj.saveChunksToCSV(subdir='smallChunks')


evalAndPrep(neutral)
```