Classifying Motor Movements from EEG Data Using a Spiking Neural Network

Nicole Heimbach, Vladimir Ivanov, Sepehr Janghorbani, Sten Knutsen, Charles Shvartsman

May 8, 2017

1 Introduction

Currently in the world of Artificial Intelligence (AI), the best classification systems are deep neural networks that use artificial neurons. However, these systems employ methods of learning - back propagation - that have not been discovered in the brain. As such, spiking neural networks (SNNs) by themselves offer little capability to learn and perform classification tasks. The current state-of-the-art in creating sophisticated, results-driven SNNs is to train deep artificial neural networks with back propagation. Then, once the network is functional, the artificial neurons are replaced with spiking neurons making the network a functional SNN. Unfortunately, such SNNs no longer have any learning capabilities.

In Lee, Delbruck and Pfeiffer (2016) researchers developed a form of back propagation that is implemented on a SNN employing the traditional deep network structure using spiking neurons [1]. Recent research has shown that neuronal dendrites do in fact produce their own membrane potential waves independent of the soma [2]. Moreover, there is clear evidence of some amount of back propagation of waves from the soma as a result of spike generation [3]. As such, a case can be made that dendrites not only receive back propagating signals from the soma, but also produce their own, resulting in some sort of computing possibly similar to the well established back propagation algorithm. Therefore, the approach outlined in Lee et al. has potential biological plausibility. And as the main goal of our project was use a SNN to classify simple limb movements in EEG data, we felt that the methods used here were likely the best approach to this complex problem.

2 Methods

2.1 Implementing Back-propagation in a Spiking Neural Network

Artificial neural networks implement back propagation as a linear combination function of inputs projected onto differentiable nonlinearities. Since the nonlinear functions are differentiable, the error at the top(output) layer can be propagated down to the input layer using the derivative of the activation function of the neurons. Usually, artificial neurons use a sigmoid activation function, however others can also be used. The main aspect is that the function is continuous and hence differentiable. This allows an algorithm like gradient descent to calculate the the change in slope of the error landscape and using the weight matrix proportionally propagate a neuron's error to its inputs. Once the error is known for each neuron in a layer, the strengths of the inputing connections can be adjusted to minimize the error of any particular neuron. The general formulaic form of this method is captured in the following expressions for error propagation and weight change:

$$\delta^{(l)} = \left((W^{(l+1)^T}) \delta^{(l+1)} \right) f^{'}(z^{(l)})$$

$$\Delta W^{(l)} = \delta^{(l+1)} (f(z^{(l)})^T$$

where:

- $\delta^{(l)}$ is the error in layer l
- $W^{(l+1)}$ is the weight matrix of layer l+1
- $f^{'}(z^{(l)})$ is the derivative of the activation function with inputs $z^{(l)}$

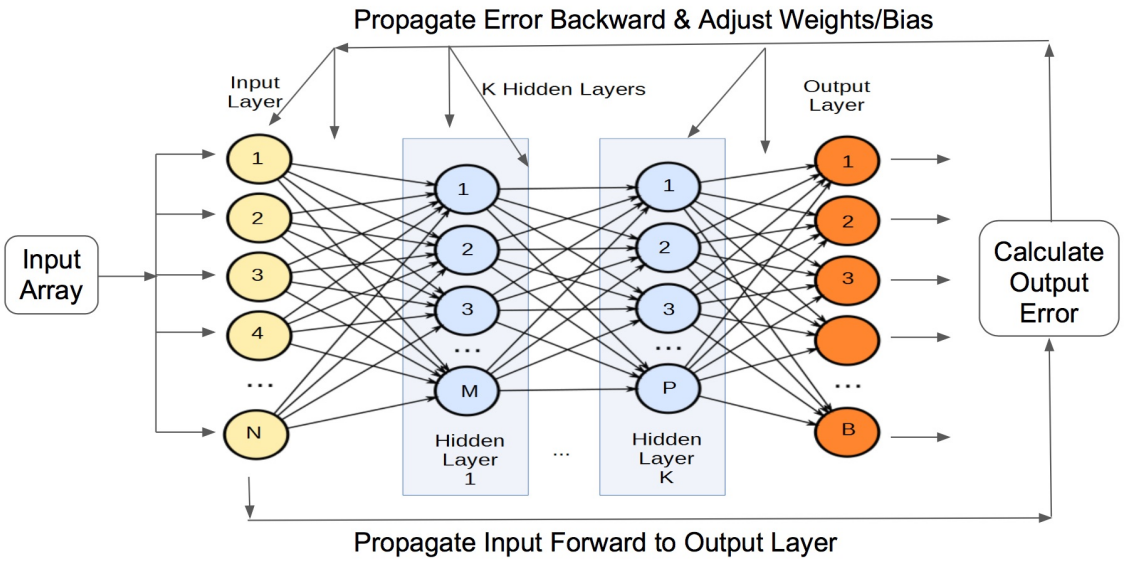

Figure 1: Overview of a neural network with backpropagation algorithm.

In the case of SNNs, the problem is that there is no continuous activation function that describes a neuron's output. Spiking neurons are complex dynamical systems that produce discrete spike out, which is not differentiable. This has been a large hurdle of the applicability of SNNs to practical applications in machine learning. However, one possible way to apply an algorithm like back propagation to an SNN is to treated for what it is, a dynamical system evolving and reacting to input over time. Thus, instead of a highly structured and linear combination form of propagating input through the network, a recent paper proposes to treat the dynamics of the network over time as the continuous signals over which inputs and errors can be propagated.[1] Therefore, a neuron's set of discrete spike responses to its set of discrete spike inputs is treated as as a continuous differentiable signal over time. Once the network has settled after witnessing an input signal for a brief period of time, the discrete responses over time are converted into continuous differentiable functions using the following formulas:

$$x_k(t) = \sum_p \exp\left(\frac{t_p - t}{\tau_{mp}}\right)$$

$$a_i(t) = \sum_q \exp\left(\frac{t_q - t}{\tau_{mp}}\right)$$

The above two functions allow us to write the membrane potential of a neuron in terms of our continuous variables:

$$V_{mp,i}(t) = \sum_{k=1}^{m} w_{ik} x_k(t) - V_{th,i} a_i(t)$$

where:

- t_p is the time of a spike arriving at synapse k
- t_q is the time of a spike generation by neuron i
- τ_{mp} is the time constant(20ms)
- $x_k(t)$ represents the entire spike transmission history of synapse k as a function of t
- $a_i(t)$ represents the entire spike generation history of neuron i
- $V_{th,i}$ is neuron i's threshold value

The above formulas were used in the back propagation part of our algorithm where t is the entire duration of activity for the network after it was presented with input. Using the above framework of a differentiable membrane potential, the paper proposes the following error propagation formula which we used in our algorithm:

$$\delta_i^{(l)} = N g_i^{(l)} \sum_j^{n^{(l+1)}} w_{ji}^{(l+1)} \delta^{(l+1)}$$

where:

- $\delta_i^{(l)}$ is the propagated error to neuron i in layer l
- $g_i^{(l)} = V'_{mp,i}(t)$ which is the derivative of activation function

This error propagation formula now corresponds to the traditional error propagation rule used in artificial neural networks. Similarly, an equivalent weight update rule is easily obtained now that there is a way to assign an error to each neuron in the network. Also, a threshold update is also performed in place of the usual bias weight update performed in artificial neural networks. Below are the formulas that describe this and which we implemented to train our networks.

$$\Delta w_{ij}^{(l)} = (-\eta_w N_w)\, \delta_i^{(l)} x_j^{(l)}$$
$$\Delta V_{th,i}^{(l)} = (-\eta_{th} N_{th})\, \delta_i^{(l)} a_i^{(l)}$$

where:

- η_w, η_{th} are the learning rates ($\eta_w = 0.003$,$\eta_{th} = 0.1\eta_w$)
- N_w, N_{th} are the normalization scalars
- $\delta_i^{(l)}$ is the error allocated to neuron i in layer l by back propagation
- $x_j^{(l)}$ spike history in synapse x_{ij}
- $a_i^{(l)}$ spike history of neuron i in layer l

And lastly, the weights and threshold are updates as follows:

$$w_{ij}^{(l)} = w_{ij}^{(l)} + \Delta w_{ij}^{(l)}$$
$$V_{th,i}^{(l)} = V_{th,i}^{(l)} + \Delta V_{th,i}^{(l)}$$

The above expressions all pertain to the back propagation part of the algorithm, which arguable is the most crucial part of both the paper we presented and our ability to get an SNN to classify EEG signal data. However, the feedforward of input data is also an important aspect of the algorithm. Namely, we used a modified version of leaky integrate and fire neurons with exponential leak and a refractory scaling factor applied to input weights:

$$V_{mp}(t_p) = V_{mp}(t_{p-1}) \exp\left(\frac{t_{p-1} - t_p}{\tau_{mp}}\right) + w_i^{(p)} w_{dyn}$$

- $V_{mp}(t)$ is the membrane potential of an arbitrary neuron
- t_p and t_{p-1} are most current and one before spike times arriving at arbitrary neuron
- $w_i^{(p)}$ is the weight of the synapse through which t_p spike has arrived
- w_{dyn} is a refractory weight
- V_{th} is the threshold of said arbitrary neuron

And the following reset rule, which actually does not reset the neuron value back to reset potential, but rather instead subtracts the threshold value of the membrane potential. This forces the network to often times run for much longer duration than the duration of the presentation of the input stimulus. This has a positive effect on classification capability of the network by amplifying the input data features.

$$V_{mp}(t_p^+) = V_{mp}(t_p) - V_{th}$$

Regarding actual implementation, we had three main functions corresponding the main aspects of training:

- A feed forward function to present and propagate input from the input layer to the output layer until all spiking ceased to occur.
- An error calculation function, which normalized the output of the output layer into an o vector. We used a scaled squared difference for error to compare the o vector to the y label, which is one of the most basic loss functions in machine learning. Below is the exact function used for training:
$$Loss = .5 * (o - y)^2$$
- A back propagation function, which took the error from the output layer and applies the procedure and formulas above to derived new weights and thresholds.

The network architecture we used consisted of the following:

- Input of size 9920
- First hidden layer of size 10,000
- Second hidden layer of size 8000
- Third hidden layer of size 3000
- Output layer of size 2

The input layer and output layers correspond the needed input and output dimensions. Since we were attempting to classify between only two movements, out two output neurons were required. Three hidden layers were chosen because a three layer network can express all logical connectives, making it most powerful with least amount of layers in terms of representation capacity.

2.2 Dataset

The dataset used in this project was recorded using the BCI2000 instrumentation system by its developers [4] and is made publically available by PhysioNet [5]. The dataset can be found at https://www.physionet.org/pn4/eegmmidb/. The reason for using such a dataset is that although we obtained our own data on one subject using a 128- channels EEG device, the environment was not isolated and controlled enough.

As per PhysioNet, the dataset contains one- to two-minute EEG recordings from a total of 109 volunteers. There were 4 main tasks that each subject completed:

1. A target appears on either the left or the right side of the screen. The subject opens and closes the corresponding fist until the target disappears. Then the subject relaxes.
2. A target appears on either the left or the right side of the screen. The subject imagines opening and closing the corresponding fist until the target disappears. Then the subject relaxes.
3. A target appears on either the top or the bottom of the screen. The subject opens and closes either both fists (if the target is on top) or both feet (if the target is on the bottom) until the target disappears. Then the subject relaxes.
4. A target appears on either the top or the bottom of the screen. The subject imagines opening and closing either both fists (if the target is on top) or both feet (if the target is on the bottom) until the target disappears. Then the subject relaxes.

The experimental protocol included 14 experimental runs in total as follows:

1. Baseline, eyes open
2. Baseline, eyes closed
3. Task 1 (open and close left or right fist)
4. Task 2 (imagine opening and closing left or right fist)
5. Task 3 (open and close both fists or both feet)
6. Task 4 (imagine opening and closing both fists or both feet)
7. Task 1
8. Task 2
9. Task 3
10. Task 4
11. Task 1
12. Task 2
13. Task 3
14. Task 4

The duration of each baseline was approximately 1 minute and the duration of each task was approximately 2 minutes.

The EEG recordings were sampled at 160 Hz and the recordings were saved in EDF+ format. There were 64 channels used for recordings and one additional channel for annotations ("EDF Annotations-1") that contains information about the events. The different event ids are listed below:

- T0 corresponds to rest
- T1 corresponds to onset of motion (real or imagined) of
  - the left fist (in runs 3, 4, 7, 8, 11, and 12)
  - both fists (in runs 5, 6, 9, 10, 13, and 14)
- T2 corresponds to onset of motion (real or imagined) of
  - the right fist (in runs 3, 4, 7, 8, 11, and 12)
  - both feet (in runs 5, 6, 9, 10, 13, and 14)

For the purpose of our project, we chose to use data from 104 of the 109 subjects. We excluded data from S088, S089, S092, S100, S106 due to oddities in the data (i.e., irregular event duration and/or mislabeled events in the annotation channel).

Since we were mainly interested in classifying between hand and foot movement, we used only the data for Task 3 (experimental runs R05, R09, R013), in which the subject moved either both hands or both feet. Each recording was about 2 minutes in total with each movement or rest period lasting for 4.0 - 4.2s. Rest periods occurred between each movement period such that there were never two consecutive movement nor rest periods.

```
for run_index in range (0,3):
# READ IN RAW DATA
    filepath = "/mnt/d/Documents (D)/homework/brain inspired computing/eeg_test_data/www.physionet.org/pn4/eegmmidb/"
    raw = get_raw_data(filepath=filepath, subject=subject, experimental_run=experimental_runs[run_index])
    ch_names=raw.ch_names
    print("Printing channels: " + ch_names[8] + ", " + ch_names[10] + ", " + ch_names[12])
    sys.exit(0)
# END READ IN RAW DATA

    #====== FIND EVENTS IN DATA ======#
    #By default, find_events returns all samples at which the value of the stim channel increases
    events=mne.find_events(raw, stim_channel='EDF Annotations-1', shortest_event=1) # 2 is default, but didn't work.
    #print events

#====== TIME-FREQUENCY ANALYSIS ======#
    n_cycles =  frequencies/4.0 # SCALING FACTOR FOR EACH FREQUENCY (I think...)
```

Figure 2: Portion of the code for reading the data.

2.3 Randomly Generated Dataset for Testing

When testing the neural network, we realized that it was not wise to initially test with the EEG data since we were not sure if the preprocessing we did to the data would allow a neural network to understand it properly. Thus, if, when debugging the network, we noticed that the network was not converging to the optimal classifier, we would have no way to determine whether the problem was in the network, or with our preprocessing of the data.

Thus, we devised a way to randomly generate a dataset that a decent classifier should be able to classify. The data was generated by first randomly generating two "prototypical" vectors of a given size that would represent the two distinct classes that should be classified. Then, to generate the actual vectors that comprise the data set, for each instantiation, we perturbed one of the the prototypical vectors by a random value generated based on a Gaussian distribution.

We created a generator that generates random variables from an approximate Gaussian Distribution by the following method. We took the sum of 12 randomly generated numbers, subtracted 6, and divided that by 6. Then, we generated each indexed value of a vector in the data set by starting with the value in the corresponding prototypical vector and perturbing it in the following way. Let v be the prototype value and g be the gaussian random variable. If $g < 0$, we would decrease the value in the prototypical vector; We would set $data = v + (1 - v) * -g$. If $g > 0$, we would increase the value in the prototypical vector; We would set $data = v * (1 - g)$. This method ensures that the generated values are between 0 and 1 and the vectors generated are not far different from the protoypical vectors that define their class.

2.4 Signal Analysis using MNE-python

One of the most famous and easy-to-use libraries for processing EEG data is the MNE-Python, a publicly available set of functions providing the tools necessary for analyzing and even classifying brain data [6]. In this regard, in the first phase of the project MNE-python was used in order to read, preprocess and analyze the raw data which would be fed into the Spiking Neural Nets in the next phase . Our process was as follows:

1. The first step was reading the raw data from .edf files. Next, using the annotation channel, we had to extract the event ids corresponding to of the experiments done. It is interesting to mention that the event ids were encoded as 12372, 12627, and 12884 (T0, T1, and T2 in ASCII format, respectively), so some decoding was also necessary.

2. The second step was using the event ids to determine the starting and end point of an event. The interval between these times is simply called an Epoch. In our processing, we used epochs that began at the time marked by the event id and that ended 4.0 seconds later. 4.0 seconds was chosen because events lasted durations of 4.0 - 4.2 seconds in total, so 4 seems to be a reasonable interval.

3. After determining the epochs, we resampled the data at 40 Hz. This was done in order to decrease the amount of data we were feeding into our network, so that the computation time does not grow exponentially.

```
if resampled_sfreq == 160:
    rest_epochs = mne.Epochs(raw, events=events, event_id= [12372], tmin=-0.0, tmax=4.0, picks=[8,10,12], preload=True)
    hand_epochs = mne.Epochs(raw, events=events, event_id= [12628], tmin=-0.0, tmax=4.0, picks=[8,10,12], preload=True)
    foot_epochs = mne.Epochs(raw, events=events, event_id= [12884], tmin=-0.0, tmax=4.0, picks=[8,10,12], preload=True)
else:
    rest_epochs = mne.Epochs(raw, events=events, event_id= [12372], tmin=-0.0, tmax=4.0, picks=[8,10,12], preload=True).resample(s
    hand_epochs = mne.Epochs(raw, events=events, event_id= [12628], tmin=-0.0, tmax=4.0, picks=[8,10,12], preload=True).resample(s
    foot_epochs = mne.Epochs(raw, events=events, event_id= [12884], tmin=-0.0, tmax=4.0, picks=[8,10,12], preload=True).resample(s
```

Figure 3: Portion of the code for separating the epochs.

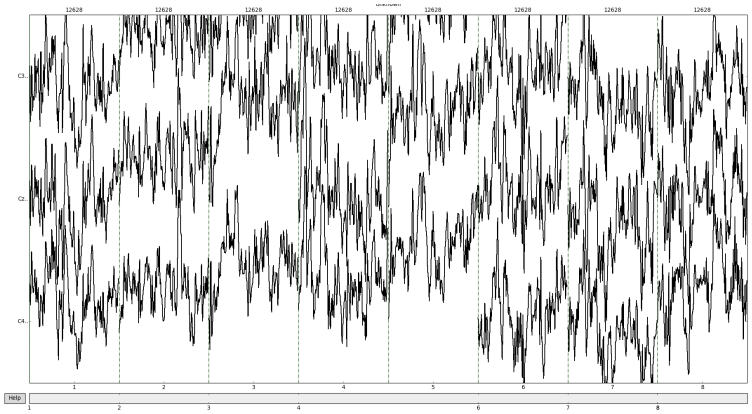

Figure 4: Diagram of hand movement epochs showing neural activity recorded from channels C3, Cz, and C4.

Channel Selection According to [7] neural activity related to hand or foot movements can be determined almost exclusively from activity in channels C3, Cz, and C4. As such, in order to reduce the amount of data the network would need to analyze and feed data into our network more efficiently, we chose to use only data from these three channels. In fact, these choices are biologically plausible since we know that most of hand-foot motor activity is associated with this region.

Frequency Band Selection Another challenge was to select a specific frequency band to focus on. Although human brainwaves seem to emit frequencies ranging from approximately 4 Hz - 100 Hz, human adults are only awake at frequencies above 4 Hz and neurological activity regarding simple motor movements seems to stay below 35 Hz [8][7][9]. Furthermore, [7] indicates that hand movement seems to peak just below 20 Hz while foot movement seems to peak just above 20 Hz. Aside from this, the fact that the frequencies above 35 Hz are not even considered, makes the data even more plausible, since most of electrical noise is present at 50 - 60 Hz.

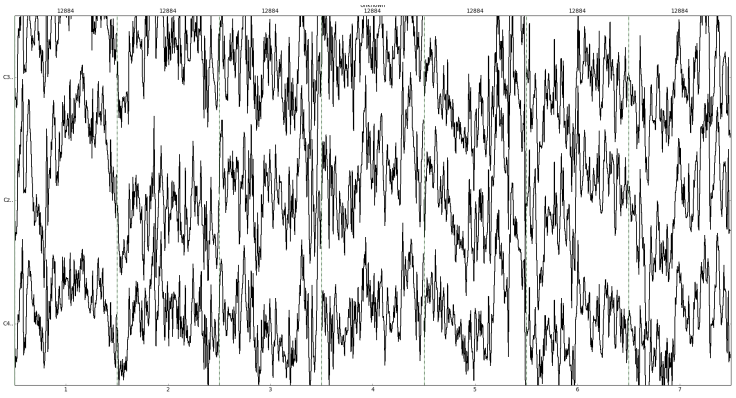

Figure 5: Diagram of foot movement epochs showing neural activity recorded from channels C3, Cz, and C4.

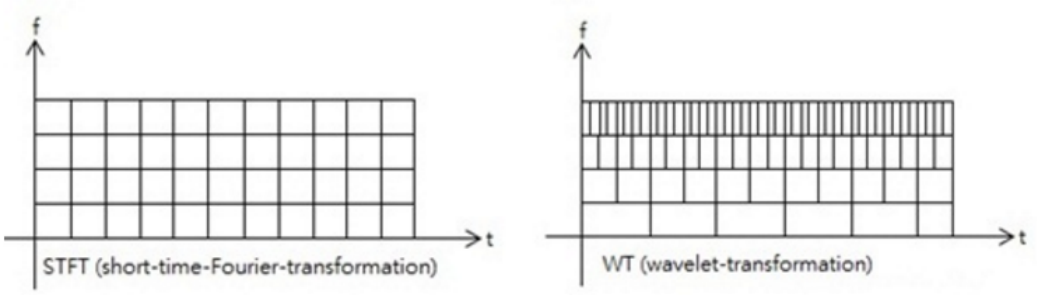

Figure 6: STFT vs. wavelet [11]

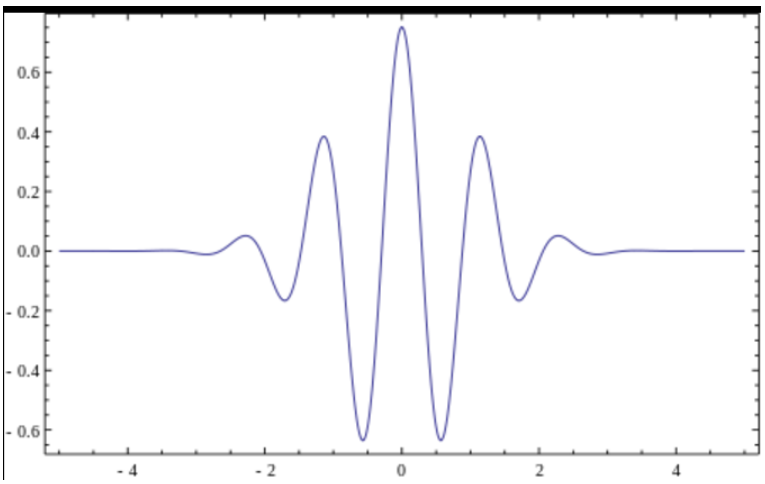

Figure 7: Morlet Wavelet mother function[12]

2.5 Time-Frequency Representation using Continuous Wavelet Transform

To get the data into a time-frequency representation, we used a continuous wavelet transform (CWT) with a Morlet wavelet as the mother wavelet function. Although we considered using Fourier Transform (FT) and Short Time Fourier Transform (STFT), neither of these transformations was ideal for the analysis of our data. Since we are looking at EEG signals, we are interested in the time localized components of the signal as well as the frequency information. As such, FT will not work because it sacrifices the time-domain altogether. Moreover, when losing the time-domain, it assumes stationarity within the signal. STFT, on the other hand, compromises between the time and frequency domains by computing FT on smaller, fixed-size windows within the raw signal. While this is more suited to our purposes than FT on its own, still assumes stationarity within its fixed windows. However, since we are working with biologically-based data, we cannot assume stationarity within fixed-size windows. Furthermore, by the Heisenberg principle, while a smaller window would give better time resolution it would also give lower frequency resolution, and similarly, a larger window would give better frequency resolution but lower time resolution [10]. The trade-offs of STFT would make it difficult to find an ideal window size to analyze our data with.

In comparison to FT and STFT, CWT using Morlet wavelets allows for a better compromise between frequency and time representation for our data without such a strong emphasis on stationarity [10]. While the Heisenberg Principle still applies, CWT has a better trade-off for our purposes. At high frequencies, CWT returns better time resolution and poorer frequency resolution, while at low frequencies, CWT returns better frequency resolution and poorer time resolution.

The choice of Morlet as the mother wavelet, then, is beneficial to us because the irregularity of the Morlet Wavelet makes it more biologically plausible [10].

It is also necessary to briefly discuss the mathematical foundation behind the Morlet- wavelet transformation. Basically, Morlet- wavelet transform is a mathematical transformation used to

approximate functions using a Morlet mother function:

Wavelet transformation equation: [11] [12]

$$X(a,b) = \frac{1}{\sqrt{a}} \int \psi(\overline{\frac{t-a}{b}}) x(t) dt \tag{1}$$

$$\psi(t) = c\pi^{\frac{-1}{4}} e^{\frac{-1}{2}t^2} (e^{i\sigma t} - \kappa) \tag{2}$$

2.6 Some technical notes on the implementation

We encountered some problems with 32-bit version of python interpreter, since memory capacity of 32-bit python (max 2 GB). As a result, we had to switch to the 64-bit python interpreter in order for Numpy library to work.

Also in order to compress the data and make the algorithm run faster, the data was fed in tor the network as the form of a single vector, which is made by concatinating the two original vectors containing the EEG data.

3 Results and Discussion

For testing the performance of our algorithm on real data, we performed training on three different data sets and for each one measured the classification accuracy. Specifically, as mentioned before, we singled out 3 channels for classificaiton due to their spatial relevance to the motor cortex.

Our results are as follows for 3 types of input, namely differing by channel:

- Input type 1 consisted of a combined vector representing data from C3 and Cz channels. With this type of input data our network was able to reach a classification accuracy of as high as 70%.
- Input type 2 consisted of data from CZ and C4 channels. This type of input resulted in a classification accuracy of 40%.
- Input type 3 consisted of data from C3 and C4. This type of input resulted in a classification accuracy of 60%.

The obtained classification results in each settings are briefly as follows:

C3 and Cz	**C4 and Cz**	**C3 and C4**
70%	40%	60%

To interpret and potentially understand our preliminary results, it is necessary to mention that C3 electrode is spatially located on the left side of the brain and associated with contralateral right hand movement. Similarly, C4 electrode is spatially located on the right and associated with left hand contralateral movement. Lastly, the Cz electrode is spatially located on the center vertial hyperplane going through the brain and spatially is most closely associated with leg movement.

Therefore, it is very interesting that our results have confirmed the hypothesis that due to fact that right hand (and the right side of the body as a whole)is being more dominant in most people, the contralateral signals of the left side of the brain may contain signals that are strongest and hence most reliable for classifying hand movement. That is to say that the right side of the brain is less dominant, it contains less uniquely identifible features due to less amount of activity.

Furthermore, our results show that having two electrodes closest to the left brain area associated with contralateral right hand movement provides best results in terms of movement classification, even though one of the electrodes(Cz) is really meant to extract foot movement. This makes sense since two independent observers of the most critical area of classification yields better classification results than only one observer. However, what this also implies is that foot movement related activity is too weak and far away to have any significant effect on classification, resulting in dominance of arm signals. Perhaps, better fine tuning our network or using more high resolution EEG equipment would result more optimistic prospects of classifying foot movement.

References

[1] Jun Haeng Lee, Tobi Delbruck, and Michael Pfeiffer. Training deep spiking neural networks using backpropagation. *Frontiers in Neuroscience*, 10:508, 2016.

[2] Michael London and Michael Hausser. Dendridic computation. *Annual Review of Neuroscience*, 28:503–32, 2005.

[3] Anne-Marie M. Oswald, Maurice J. Chacron, Brent Doiron, Joseph Bastian, and Leonard Maler. Parallel processing of sensory input by bursts and isolated spikes. *Journal of Neuroscience*, 24(18):4351–4362, 2004.

[4] G. Schalk, D.J. Mcfarland, T. Hinterberger, N. Birbaumer, and J.R. Wolpaw. Bci2000: A general-purpose brain-computer interface (bci) system. *IEEE Transactions on Biomedical Engineering*, 51(6):1034–1043, 2004.

[5] A. L. Goldberger, L. A. N. Amaral, L. Glass, J. M. Hausdorff, P. Ch. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley. PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals. *Circulation*, 101(23):e215–e220, 2000 (June 13). Circulation Electronic Pages: http://circ.ahajournals.org/content/101/23/e215.full PMID:1085218; doi: 10.1161/01.CIR.101.23.e215.

[6] Alexandre Gramfort. Meg and eeg data analysis with mne-python. *Frontiers in Neuroscience*, 7, 2013.

[7] C Neuper and G Pfurtscheller. Evidence for distinct beta resonance frequencies in human eeg related to specific sensorimotor cortical areas. *Clinical Neurophysiology*, 112(11):2084–2097, 2001.

[8] J. Sleight, P. Pillai, and S. Mohan. Classification of executed and imagined motor movement eeg signals. *Ann Arbor: University of Michigan*, pages 1–10, 2009.

[9] A Tzelepi, T Bezerianos, and I Bodis-Wollner. Functional properties of sub-bands of oscillatory brain waves to pattern visual stimulation in man. *Clinical Neurophysiology*, 111(2):259–269, 2000.

[10] Vincent J. Samar, Ajit Bopardikar, Raghuveer Rao, and Kenneth Swartz. Wavelet analysis of neuroelectric waveforms: A conceptual tutorial. *Brain and Language*, 66(1):7–60, 1999.

[11] Wavelet Transform. Wikipedia, the free encyclopedia. 2017. [Online; accessed May 08, 2017].

[12] Morlet wavelet. wikipedia, the free encyclopedia, 2017. [Online; accessed May 08, 2017].

A Python scripts

Listing 1: Core

```
from __future__ import division
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import math as m
import random as rnd
import pickle
import os


def print_save_output(str_output):
    print (str_output)
    current_path = os.getcwd()
```

```
    output_log = open( current_path + "/output_log4", 'a+') #save
        output here
    output_log.write(str_output+"\n")
    output_log.close()


##################Construct Network Structure

#layers is a list of numbers representing number of neurons in each
    layer, with first # being input layer, last being output layer
def createNetwork(tNumNeurons,layers,alpha,wta=False):

    # calculate initialization parameter a:
    # used to specify range from which to initialize weights
    # used to initialize membrane threshold
    a = np.ones(len(layers),dtype=np.float64)
#initialize input layer a independently
    a[0] = m.sqrt(3 / layers[0])
#initialize a for all other layers based on number of neurons in
    inputting layer
    for i in range(1,len(a)):
        a[i] = m.sqrt(3 / layers[i-1])

#create index dictionarry marking the layers out in Vmp, Vth
    indexList = []
    currentIndex = 0
    for i in range(0,len(layers)):
        indexList.append((i,(currentIndex,currentIndex+layers[i])))
        currentIndex = currentIndex + layers[i]
    layerIndDict = dict(indexList)

#create initial membrane potential and thresholding arrays
    Vmp = np.ones(tNumNeurons, dtype=np.float64)
    Vth = np.ones(tNumNeurons, dtype=np.float64)
    for i in range(0,len(a)):
        Vmp[layerIndDict[i][0]:layerIndDict[i][1]] = 0
        Vth[layerIndDict[i][0]:layerIndDict[i][1]] = alpha[i]*a[i]

#create dictionary of numpy arrays for storing spike times of each
    neuron, used in backprop equations
    tempDictList = []
    for i in range(0,tNumNeurons):
        tempDictList.append((i,np.zeros(0,dtype=np.float64)))
    #spikeTimesDict = dict(tempDictList)




# creates dictionary of connection matricies for each layer
    connectMatrixList = []
    for i in range(1,len(layers)):
        connectMatrixList.append((i,np.random.uniform(-a[i],a[i],layers
            [i]*layers[i-1]).reshape((layers[i],layers[i-1]))))
    conMatDict = dict(connectMatrixList)
```

```
    return Vmp,Vth,layerIndDict,conMatDict,tempDictList



################## Updating functions

def feedSample(inputArray,timeLimit,Vmp,Vth,layerIndDict,conMatDict,
    spikeTimesDict,train=True,tao=20,Tref=1,row=.00001):

    numLayers = len(layerIndDict)
    tNumNeurons = len(Vmp)
    BoolSpike = np.zeros(tNumNeurons, dtype=np.int64)
# creates 2 x tNumNeurons numppy array to store last two spike times of
     each neuron, used for feeding forward signal
    sT = np.zeros((2, tNumNeurons), dtype=np.float64)

    outputCounter = np.zeros((layerIndDict[numLayers-1][1]-layerIndDict
        [numLayers-1][0]), dtype=np.int64)
    neuronCounter = np.zeros(len(Vmp),dtype=np.int64)


######### delete after testing
    Ft = -1 * np.ones(1, dtype=np.int64)
    Fn = -1 * np.ones(1, dtype=np.int64)


    t = 0
    settled = False
    while(settled == False):

########obsolete
#    #calcs exp((t_p-1 - t_p)/tao) for all neurons
#        #E = np.exp(np.divide(sT[0,:]-sT[1,:],tao))
#    #cancel out E values for neurons that didn't spike in previous
    time step
#        #presentE = np.einsum('...,...',E,BoolSpike)
#    #multiplies piecewise V x E
#        #VE = np.einsum('...,...',Vmp,presentE)
######################

    #feed input layer
        if t <= timeLimit:
            Vmp[layerIndDict[0][0]:layerIndDict[0][1]] = np.add(Vmp[
                layerIndDict[0][0]:layerIndDict[0][1]],inputArray)

# must still address input neuron update
        for i in range(1,numLayers):
        #calc t_p
            tp = np.amax(sT[1][layerIndDict[i-1][0]:layerIndDict[i
                -1][1]])
        #calc exponential expression, same scalar for entire layer
            e = m.exp((np.amax(sT[0][layerIndDict[i-1][0]:layerIndDict[
                i-1][1]]) - tp)/tao)
        #calc dynamical W array, each value corresponding to particular
             neuron in layer
            Wdyn = np.square(np.divide(np.subtract(sT[1][layerIndDict[i
                ][0]:layerIndDict[i][1]], tp),Tref))
```

```
            tempInd = (Wdyn > 1).nonzero()[0]
            Wdyn[tempInd] = 1

        #calcs vector of sum of Wi's for each layer
            WW = np.einsum('...,...',Wdyn,np.einsum('ij,j',conMatDict[i
                ],BoolSpike[layerIndDict[i-1][0]:layerIndDict[i-1][1]])
                )

        #adds weighted spike input from lower layer to each neuron in
            this layer
        #finally updates Vmp
            Vmp[layerIndDict[i][0]:layerIndDict[i][1]] = np.add(np.
                einsum('...,...',e,Vmp[layerIndDict[i][0]:layerIndDict[
                i][1]]),WW)


#resetting Vmp
    ********************************************************************************

        dThresh = np.subtract(Vmp,Vth)
        fired = (dThresh > 0).nonzero()[0]
        neuronCounter[fired] += 1
        Vmp[fired] = dThresh[fired]
        dTresh2 = np.add(Vmp,Vth)
        belowNeg = (dTresh2 < 0).nonzero()[0]
        Vmp[belowNeg] = np.einsum('...,...',-1,Vth[belowNeg])


### Apply threshold regularization
        if train == True:
            #print('ttttttttttt',t)
            tempArr = np.arange(len(Vth), dtype=np.int64)
            for i in range(0,numLayers):
                #print('i',i)
                firedNeurons = np.intersect1d(tempArr[layerIndDict[i
                    ][0]:layerIndDict[i][1]],fired)
                notfiredNeurons = np.setdiff1d(tempArr[layerIndDict[i
                    ][0]:layerIndDict[i][1]],fired)
                #print('NOT FIRED: ',row*len(notfiredNeurons))
                Vth[firedNeurons] = np.add(Vth[firedNeurons],row*len(
                    notfiredNeurons))
                #print('FIRED: ', row*len(firedNeurons))
                Vth[notfiredNeurons] = np.subtract(Vth[notfiredNeurons
                    ],row*len(firedNeurons))
                #print('AVE for' , i,' : ', np.mean(Vth[layerIndDict[i
                    ][0]:layerIndDict[i][1]]))
#update sT array
        sT[0][fired] = sT[1][fired]
        sT[1][fired] = t

#update spikeTimesDict
        for i in fired:
            spikeTimesDict[i] = np.append(spikeTimesDict[i],t)

#set binary spike vector
        BoolSpike[:] = 0
        BoolSpike[fired] = 1
```

```
#reevaluate conditional whil3 loop variable
        if t > timeLimit and len(fired) == 0:
            settled = True



################### temp aggregates arrays of all neurons that spiked at
    which t
        tvec = t * np.ones(len(fired), dtype=np.int64)
        Ft = np.concatenate((Ft, tvec))
        Fn = np.concatenate((Fn, fired))

        t = t + 1

#check which output neurons fired in this time step
        #outputFired = (fired > (layerIndDict[numLayers-1][0]-1)).
            nonzero()[0]
        #outputCounter[np.mod(fired[outputFired], layerIndDict[numLayers
            -1][0])] += 1
    outputCounter = neuronCounter[layerIndDict[numLayers-1][0]:
        tNumNeurons]

    return outputCounter,t-1,Ft,Fn,neuronCounter,spikeTimesDict

def outputError(outputCounter,y):

    maxSpikes = np.amax(outputCounter)
#normalize outputCounter by maximum output neuron
    if maxSpikes > 0:
        oNrmd = np.divide(outputCounter,maxSpikes)
    else:
        oNrmd = np.zeros(len(outputCounter))

    lossArray = np.multiply(.5,np.square(np.subtract(oNrmd,y)))

    return lossArray

def backprop(outputLoss,tMax,Vth,layerIndDict,conMatDict,spikeTimesDict
    ,tao=200,muW=.003,bet=.001,lam=.003):

    numLayers = len(layerIndDict)
#initializes errors for output layer
    error = outputLoss

#calculate x_k for each neuron
    XA = np.zeros(len(Vth),dtype=np.float64)
    for k in range(0,len(Vth)):
        XA[k] = np.sum(np.exp(np.divide(np.subtract(spikeTimesDict[k],
            tMax),tao)))
        #print_save_output ("X_K TIME DIFFFEREENCE!!!!!" + str(np.exp(
            np.divide(np.subtract(spikeTimesDict[k],tMax),tao))))
        for j in range(0,len(spikeTimesDict[k])):
            if(spikeTimesDict[k][j] - tMax > 0):
                print_save_output ("CRIMES!!!!CRIMES\nCRIMES\nCRIMES\
                    nCRIMES")

#extract x_k, a_i for each neuron
    for i in range(numLayers-1,0,-1):
```

```
            A = XA[layerIndDict[i][0]:layerIndDict[i][1]]
            X = XA[layerIndDict[i-1][0]:layerIndDict[i-1][1]]


        #propagates errors from previous layer to this onef
            if i != (numLayers -1):
                ratioError = np.divide(1,np.multiply(Vth[layerIndDict[i
                    ][0]:layerIndDict[i][1]],np.sqrt(np.mean(np.square(np.
                    divide(1,Vth[layerIndDict[i][0]:layerIndDict[i][1]])))
                    ))
                error = np.multiply(ratioError,np.dot(np.transpose(
                    conMatDict[i+1]),error))

        #calculate weight regularization vector
            sumInputs = np.square(conMatDict[i])
            sumInputs = np.sum(sumInputs, 1)
            sumInputs = np.subtract(sumInputs,1)
            coef = .5*lam
            wReg = np.multiply(coef,np.exp(np.multiply(bet,sumInputs)))
            error = np.add(error,wReg)

        #calculates dw matrix and dVth array
            ratioW = np.sqrt(np.divide((layerIndDict[i][1]-layerIndDict[i
                ][0]),(layerIndDict[i-1][1]-layerIndDict[i-1][0])))
            ratioV = np.sqrt(np.divide((layerIndDict[i][1]-layerIndDict[i
                ][0]),np.square(layerIndDict[i-1][1]-layerIndDict[i-1][0]))
                )
            dW = np.multiply((-1*muW),np.multiply(ratioW,np.outer(error,X))
                )
            dVth = np.multiply((-.1*muW),np.multiply(ratioV,np.multiply(
                error,A)))

        #change the weights and v for layer
            conMatDict[i] = np.add(conMatDict[i],dW)
            Vth[layerIndDict[i][0]:layerIndDict[i][1]] = np.add(Vth[
                layerIndDict[i][0]:layerIndDict[i][1]],dVth)

        # update threshold for first layer, no weights for first layer
        A = XA[layerIndDict[0][0]:layerIndDict[0][1]]
        ratioError = np.divide(1, np.multiply(Vth[layerIndDict[0][0]:
            layerIndDict[0][1]], np.sqrt(np.mean(np.square(np.divide(1, Vth
            [layerIndDict[0][0]:layerIndDict[0][1]]))))))
        error = np.multiply(ratioError, np.dot(np.transpose(conMatDict[0 +
            1]), error))
        ratioV = np.sqrt(layerIndDict[0][1] - layerIndDict[0][0])
        dVth = np.multiply((-0.1 * muW), np.multiply(ratioV, np.multiply(
            error, A)))

        return conMatDict,Vth
```

Listing 2: Run Net

```
from __future__ import division
from core import *
import numpy as np
import os
import time
import matplotlib
import matplotlib.pyplot as plt
```

```
import shutil
import pdb

def print_save_output(str_output):
    print (str_output)
    current_path = os.getcwd()
    output_log = open( current_path + "/output_log4", 'a+') #save
        output here
    output_log.write(str_output+"\n")
    output_log.close()
def read_files(data_path, data_path_out):
    data_array=[]

    file = data_path
    f = open(file)
    data_array = pickle.load(f)
    f.close()

    np.random.shuffle(data_array)

    return data_array

### VARS

trainingDuration = 1000

###directories
current_path = os.getcwd()
data_path = current_path + '/fakedata' #current_path + '/HFDPoTT'
data_path_out = current_path + '/outpath'
## SAVE NET TO BASE NAME:
trndNt = 'trndNt_power_Net4'

## READ NET:_____________________________________
network = 'NET_4'
connections = 'NET_4_mat.npz'


networkData = open(network, 'rb')
#Vmp = pickle.load(networkData) #membrane potentials
Vth = pickle.load(networkData) #thresholds for firing
layerIndDict = pickle.load(networkData)
spikeTimesList = pickle.load(networkData)
networkData.close()
#-----------------------------------------------

# Assembles conMatDict ___________________________

npz = np.load(connections)
ll = npz['arr_0']
conMatList = []
for i in range(1, ll):
    key = 'arr_'+str(i)
    conMatList.append((i,npz[key]))
conMatDict = dict(conMatList)
#-----------------------------------------------
data_array=read_files(data_path=data_path, data_path_out=data_path_out)
```

```
accuracyList = []
timeList = []
for iteration in range(0, len(data_array)):
#### ___TRAINING PROCEDURES:
    ------------------------------------------------------------------------

    start_time = time.time()
    Vmp = np.zeros(len(Vth),dtype = np.float64)
    spikeTimesDict = dict(spikeTimesList)
    print_save_output('STARTING ITERATION: '+ str(iteration))
    print_save_output('Feeding Input for '+ str(trainingDuration)+ ' ms
        ')
    counter,tMax,Ft,Fn,neuronCounter,std = feedSample(data_array[
        iteration][2:],trainingDuration,Vmp,Vth,layerIndDict,conMatDict
        ,spikeTimesDict)
    print_save_output('output counter: '+ str(counter)) #print('output
        counter: ',counter)
    print_save_output ('neuronCounter: ' + str(neuronCounter))
    label = data_array[iteration][0:2]
    lossArray = outputError(counter,label)
    print_save_output('Correct Label: '+str(label))
    print_save_output('lossArray: '+str(lossArray))
    print_save_output('Backprop Started.')
    conMatDict,Vth = backprop(lossArray,tMax,Vth,layerIndDict,
        conMatDict,std)
    print_save_output('Backprop Done.')
    end_time = time.time()
    ##### meansures running accuracy
    if np.sum(counter) != 0 and counter[0] != counter[1]:
        if (counter[0] > counter[1] and label[0] > label[1]) or (
            counter[0] < counter[1] and label[0] < label[1]):
            accuracyList.append(1)
        else:
            accuracyList.append(0)

    if len(accuracyList) > 10:
        runAcc = (sum(accuracyList[(len(accuracyList) - 10):len(
            accuracyList)]))/10
    else:
        if len(accuracyList) == 0:
            runAcc = -2
        else:
            runAcc = (sum(accuracyList[0:len(accuracyList)]))/len(
                accuracyList)

    ##### measures running time
    run_time = end_time-start_time
    timeList.append(run_time)
    aveRunTime = sum(timeList)/len(timeList)

    print_save_output('END OF ITERATION: '+str(iteration))
    print_save_output('RUNNING ACCURACY OVER LAST 10 RUNS: _____
        '+str(runAcc))
    print_save_output('AVERAGE RUNNING TIME PER ITERATION: _____
        '+str(aveRunTime))

#
    ------------------------------------------------------------------------
```

```
    # fig1 = plt.figure(1)
    # plt.scatter(Ft[1:len(Ft)-1],Fn[1:len(Fn)-1],s=1)
    # plt.show()
## ______SAVING PROCEDURES EVERY SEVERAL ITERATIONS:
    ____________________________________________________
    if iteration % 100 == 0:
        print_save_output('Saving Network on Iteration: '+str(iteration
            ))
        networkT = trndNt + '_itr_' + str(iteration)
        networkTmat = trndNt + '_mat_itr_' + str(iteration)
        networkData = open(networkT, 'wb')
        pickle.dump(Vmp, networkData)
        pickle.dump(Vth, networkData)
        pickle.dump(layerIndDict, networkData)
        #pickle.dump(conMatDict, networkData)
        pickle.dump(spikeTimesDict, networkData)
        pickle.dump(accuracyList,networkData)
        np.savez(networkTmat, ll, conMatDict[1], conMatDict[2],
            conMatDict[3], conMatDict[4])
        print_save_output('NETWORK_SAVED')
        networkData.close()
#
    ____________________________________________________________________________________________


    ### trained on file to trained directory


print_save_output('SAVING_NETWORK_ON_FINAL_ITERATION')
networkT = trndNt + '_FINAL'
networkTmat = trndNt + '_mat_FINAL'
networkData = open(networkT, 'wb')
pickle.dump(Vmp, networkData)
pickle.dump(Vth, networkData)
pickle.dump(layerIndDict, networkData)
#pickle.dump(conMatDict, networkData)
pickle.dump(spikeTimesDict, networkData)
pickle.dump(accuracyList,networkData)
np.savez(networkTmat, ll, conMatDict[1], conMatDict[2], conMatDict[3],
    conMatDict[4])
print_save_output('NETWORK_SAVED')
networkData.close()
```

Listing 3: Build Net

```
from core import *
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import pickle




layers = [10,20,20,10,2]
totalNeurons = sum(layers)
#unique per layer, depends on size of layer, smaller layer needs
    smaller alpha
```

```
# normal range is from 3 - 10
alpha = [6,7,5,5,3]

print('Creating Network')
Vmp, Vth, layerIndDict, conMatDict, spikeTimesList = createNetwork(
    totalNeurons, layers, alpha)
print('Network Created')


#print(conMatDict[len(layers)-1])
networkName = 'NET_4'
connections = 'NET_4_mat'


print('Saving Network:')
output = open(networkName, 'wb')
print('Saving Vmp')
#pickle.dump(Vmp, output)
print('Saving Vth')
pickle.dump(Vth, output)
print('Saving layer index dictionary')
pickle.dump(layerIndDict, output)
#pickle.dump(conMatDict, output)
print('Saving spike times dictionary')
pickle.dump(spikeTimesList, output)
output.close()

#output2 = open(connections, 'wb')

print('Saving Connection Matrices')
np.savez(connections,len(layers),conMatDict[1],conMatDict[2],conMatDict
    [3],conMatDict[4])

#pickle.dump(a,output2)
#for i in range(0,len(layers)+1):
#    if i ==0:
#        pickle.dump(len(layers),output2)
#    else:
#        np.save(conMatDict[i], output2)
#output2.close()
print('Network Saved to: ',connections)
for i in range(1,len(conMatDict)):
    print(np.shape(conMatDict[i]))
```

Listing 4: Random Data Generator

```
import random, decimal
import numpy as np
import pickle

vectorsize = 10
data = np.empty([3000,vectorsize+2], dtype=float)
rvp1 = np.empty(vectorsize+2, dtype=float)
rvp2 = np.empty(vectorsize+2, dtype=float)

for i in range (0, len(rvp1)):
        rvp1[i]= random.random()
        rvp2[i]= random.random()
```

```
for i in range (0, len(data)):

        # if i%3==0:
        #       for j in range(2, len(data[i])):
        #               data[i][j]=random.random()
        #       data[i][0]=0
        #       data[i][1]=0

        if i%2==0:
                gauss_RV=0.0
                for j in range(2, len(data[i])):
                        for k in range(0, 12):
                                r = random.random()
                                gauss_RV += r
                        gauss_RV=(gauss_RV-6.0)/6.0

                        if gauss_RV < 0:
                                data[i][j]=rvp1[j-2]+(1-rvp1[j-2]*abs(
                                    gauss_RV))
                        if gauss_RV>=0:
                                data[i][j]=rvp1[j-2]*(1-gauss_RV)
                data[i][0]=1
                data[i][1]=0
        if i%2==1:
                gauss_RV=0.0
                for j in range(2, len(data[i])):
                        for k in range(0, 12):
                                r = random.random()
                                gauss_RV += r
                        gauss_RV=(gauss_RV-6.0)/6.0

                        if gauss_RV < 0:
                                data[i][j]=rvp1[j-2]+(1-rvp2[j-2]*abs(
                                    gauss_RV))
                        if gauss_RV>=0:
                                data[i][j]=rvp2[j-2]*(1-gauss_RV)
                data[i][0]=0
                data[i][1]=1
#counter = 1
#for i in range (0, 30):
        #for j in range(0, len(data)):
                #file = open("data"+str(counter), "w+")



file=open('fakedata', "w+")
pickle.dump(data, file)
file.close()
                #counter++
```

Listing 5: EEG Signal Analysis

```
# System modules
import sys

# MNE modules
import mne
import numpy as np
import pickle
```

```
#DATA VER 1: 40 Hz, frequencies=np.arange(4,35). This is what we are
    using.

#returns resampling rate based on data version
def get_ver_sfreq(data_ver):
        if data_ver==0:
                return 160
        elif data_ver==1 or data_ver==2:
                return 40
        elif data_ver==3 or data_ver==4:
                return 80
        else: return 160 # default value

#returns frequencies of interest based on data version
def get_ver_frequencies(data_ver):
        if data_ver==0 or data_ver==1 or data_ver==4:
                return np.arange(4,35)

        elif data_ver==2 or data_ver==3: # exponentially increase
            frequencies of interest by 4 * (1.13^i)
                max_exponent = 17
                freq_array = np.empty(max_exponent+1, dtype=float)
                for i in range (0, max_exponent+1):
                        freq_array[i]=(4.0* (1.13**i))
                return freq_array

        else: return [] # default value

#returns raw data
def get_raw_data(filepath, subject, experimental_run):
        if subject<10:
                str_subject="S00"+str(subject)
        elif subject < 100:
                str_subject="S0"+str(subject)
        else : str_subject="S"+str(subject)
        if experimental_run < 10:
                str_experimental_run="R0"+str(experimental_run)
        else: str_experimental_run = "R"+str(experimental_run)

        print ("\nReading data for "+str_subject+str_experimental_run+"
            ...")

        return mne.io.read_raw_edf( filepath+ str_subject+"/" +
            str_subject+str_experimental_run+".edf",
                                                        stim_channel
                                                            =65,

                                                        preload
                                                            =

                                                            True
                                                            )

#prints epoch size of the array so we know what size network will be
    needed
def print_morlet_array_epoch_size(morlet_array):
```

```
        print (str(len(morlet_array)) #n_epochs
                        + " x " + str(len(morlet_array[0])) #
                            n_channels
                        + " x " + str(len(morlet_array[0][0])) #
                            n_freqs
                        + " x " + str(len(morlet_array[0][0][0])) #
                            n_times
                        + " \n --> Epoch size: " + str(len(morlet_array
                            [0])*len(morlet_array[0][0])*len(
                            morlet_array[0][0][0])))

def save_epochs_individually(morlet_array,filepath, type, counter):
        for i in range (0, len(morlet_array)):
                file=open(""+filepath+type+str(counter+i+1), "wb+")
                pickle.dump(morlet_array[i],file)
                file.close()

        return counter+len(morlet_array)


#MAIN VARIABLES TO MODIFY FOR DIFFERENT OUTPUTS AND SUCH.
rest_counter = 0 #used for file names (e.g., rest1, rest2, rest3, ...).
hand_counter = 0 #used for file names
foot_counter = 0 #used for file names
data_versions = [0,1,2,3]
data_ver = data_versions[1]
morlet_output="power" #what type of output to use for morlet

experimental_runs = [5, 9, 13] #Task 3: Open and close both fists or
    both feet

#Paper 1: http://ac.els-cdn.com/S1388245701006617/1-s2.0-
    S1388245701006617-main.pdf?_tid=e265fe96-29b5-11e7-aacb-00000
    aab0f6b&acdnat=1493124862_de4fbf6f4c9ea8a8ed9b0ae0a631fd4f
#Paper 2: http://web.eecs.umich.edu/~cscott/past_courses/eecs545f09/
    projects/MohanPillaiSleight.pdf
resampled_sfreq = get_ver_sfreq(data_ver=data_ver)
frequencies = get_ver_frequencies(data_ver=data_ver)


for subject in range (1, 110):
        if subject==88 or subject==89 or subject==92 or subject==100 or
            subject==106:
                continue
        for run_index in range (0,3):
        # READ IN RAW DATA
                filepath = "/mnt/d/Documents (D)/homework/brain
                    inspired computing/eeg_test_data/www.physionet.org/
                    pn4/eegmmidb/"
                raw = get_raw_data(filepath=filepath, subject=subject,
                    experimental_run=experimental_runs[run_index])
                ch_names=raw.ch_names
                # print("Printing channels: " + ch_names[8] + ", " +
                    ch_names[10] + ", " + ch_names[12])
                # sys.exit(0)
        # END READ IN RAW DATA

                #====== FIND EVENTS IN DATA ======#
```

```
        #By default, find_events returns all samples at which
            the value of the stim channel increases
        events=mne.find_events(raw, stim_channel='EDF␣
            Annotations-1', shortest_event=1) # 2 is default,
            but didn't work. changed to 1.
        #print events

#====== TIME-FREQUENCY ANALYSIS ======#

#event ids 12372, 12628, 12884 correspond to T0, T1, T2,
    respectively
#T0 is rest     T1 is both hands        T2 is both feet
#tmax is the offset from the event (i.e., the end of the epoch)
        if resampled_sfreq == 160:
                rest_epochs = mne.Epochs(raw, events=events,
                    event_id= [12372], tmin=-0.0, tmax=4.0,
                    picks=[8,10,12], preload=True)
                hand_epochs = mne.Epochs(raw, events=events,
                    event_id= [12628], tmin=-0.0, tmax=4.0,
                    picks=[8,10,12], preload=True)
                foot_epochs = mne.Epochs(raw, events=events,
                    event_id= [12884], tmin=-0.0, tmax=4.0,
                    picks=[8,10,12], preload=True)
        else:
                rest_epochs = mne.Epochs(raw, events=events,
                    event_id= [12372], tmin=-0.0, tmax=4.0,
                    picks=[8,10,12], preload=True).resample(
                    sfreq=resampled_sfreq)
                hand_epochs = mne.Epochs(raw, events=events,
                    event_id= [12628], tmin=-0.0, tmax=4.0,
                    picks=[8,10,12], preload=True).resample(
                    sfreq=resampled_sfreq)
                #hand_epochs.plot(block=True) #use export
                    DISPLAY=:0.0 to execute this
                foot_epochs = mne.Epochs(raw, events=events,
                    event_id= [12884], tmin=-0.0, tmax=4.0,
                    picks=[8,10,12], preload=True).resample(
                    sfreq=resampled_sfreq)
                #foot_epochs.plot(block=True)

        rest_morlet_array=mne.time_frequency.tfr_array_morlet(
            epoch_data=rest_epochs.get_data(), sfreq=
            resampled_sfreq, frequencies=frequencies,  n_cycles
            =n_cycles, use_fft=True, output=morlet_output)#
            print zip(*rest_morlet_array)[0]
        hand_morlet_array=mne.time_frequency.tfr_array_morlet(
            epoch_data=hand_epochs.get_data(), sfreq=
            resampled_sfreq, frequencies=frequencies,  n_cycles
            =n_cycles, use_fft=True, output=morlet_output)#
            print hand_morlet_array[0][0][0]
        foot_morlet_array=mne.time_frequency.tfr_array_morlet(
            epoch_data=foot_epochs.get_data(), sfreq=
            resampled_sfreq, frequencies=frequencies,  n_cycles
            =n_cycles, use_fft=True, output=morlet_output)
        print
        print ("\nRest:␣")
        print_morlet_array_epoch_size(rest_morlet_array)
        print ("\nHand:␣")
```

```
print_morlet_array_epoch_size(hand_morlet_array)
print ("\nFoot: ")
print_morlet_array_epoch_size(foot_morlet_array)
print

#====== SAVE DATA ======#
data_filepath="/mnt/d/Documents (D)/homework/brain
    inspired computing/data for vlad/data_v1_power/"
rest_counter=save_epochs_individually(morlet_array=
    rest_morlet_array, filepath=data_filepath, type="
    rest", counter=rest_counter)
hand_counter=save_epochs_individually(morlet_array=
    hand_morlet_array, filepath=data_filepath, type="
    hand", counter=hand_counter)
foot_counter=save_epochs_individually(morlet_array=
    foot_morlet_array, filepath=data_filepath, type="
    foot", counter=foot_counter)
```