Neural Signal Processing to Identify Human Emotions

Pradeep Roy Yadlapalli (MS CS), Eric Jing (Phd CS), Annalie Swanepoel (BS CS)

5/3/23
525 Brain Inspired Computing
Prof. Konstantinos Michmizos

Abstract

Neural signal processing is a field of study that involves the analysis and interpretation of neural signals generated in the human brain, nervous systems or other biological systems. These signals, such as electroencephalograms (EEGs) and electromyograms (EMGs), provide insight into brain activity, neural communication, and muscle function. A low computational model was necessary for real time analysis of these data. We explored the Spiking Neural Network (SNN) model as a possible solution for this. So, a spiking neural network model was used to estimate the emotional state of a person into positive, neutral and negative using EEG brain waves as input.

Traditional machine learning and deep-learning techniques have achieved an accuracy of 98% on the test dataset. Our SNN model achieves an accuracy of 88.56% on our test dataset. The predictions result of our model are on par with some of the best available models. We conclude that spiking neural networks work very well and can serve as an alternative solution for real time data analysis when there are limitations on power usage and computational capabilities.

Introduction

Neural Signal Processing is a specialized area of signal processing aimed at extracting information and decoding intent from neural signals recorded to better understand how the brain represents and conveys information, through neuronal ensembles from the central or peripheral nervous system. This has significant applications in the area of neuroscience and neural engineering. With the complexity and size of brain recordings growing, the use of neural signal processing has gained significance in the area of neuroscience. Interpreting these recorded signals such as electroencephalograms (EEGs) and electromyograms (EMGs), can provide insight into brain activity, neural communication, and muscle function. Electroencephalography is a non-invasive technique used to track the electrical activity of the brain through the placement of electrodes on the scalp. Analyzing EEG signals in real-time is becoming increasingly important in various applications such as neurofeedback, brain-computer interfaces, and monitoring of patients with neurological disorders. All these applications are possible through wearable technology, but the only drawback of these wearable tech is that they have less battery and low computational power. Hence, it is high time that we need machine learning models that can not only process the EEG signals in real time, but also are energy efficient. Spiking Neural Networks (SNN's) best fit this role and have shown significant promise in recent years to be widely used. Since the scope of analyzing EEG signals is massive and there is very less EEG data available, we tried to address a smaller problem of estimating the emotional state of an individual into positive, neutral and negative using a Spiking Neural Network model on EEG brainwaves dataset.

Autonomous non-invasive detection of emotional states is potentially useful in multiple domains such as human robot interaction and mental healthcare. It can provide an extra dimension of interaction between user and device, as well as enabling tangible information to be derived that does not depend on verbal communication [1]. With the increasing availability of low-cost EEG devices, brainwave data is becoming affordable for the consumer industry as well as for research, introducing the need for autonomous classification without the requirement of an expert on hand[2]. Spiking neural networks can here act as an autonomous classification model.

Spiking neural networks are a type of artificial neural network that model the behavior of neurons more accurately than traditional artificial neural networks (ANNs), which makes them better suited for accurate modelling of applications that involve processing time-varying signals, such as EEG

data. SNNs can process data more efficiently than traditional ANNs as they are more biologically realistic in their approach, which allows for better utilization of computational resources and more efficient processing of large datasets. Spiking neural networks can also be used for control tasks, such as controlling prosthetic limbs or computer cursors.

The above-mentioned smaller problem will aid in the development of many other real-world applications for better diagnostic capability of conditions such as epilepsy, depression, dementia, Alzheimer's and Parkinson's disease as well as improved monitoring and treatment options. Moreover, further along in the future, neural prosthetics (BMIs) can be used to restore muscle and cognition function to patients with spinal cord injuries and other neurological conditions.

2. Background (or Theory)

2.1 Electroencephalography

Electroencephalography is the process using applied electrodes to derive electrophysiological data and signals produced by the brain. Electrodes can be subdural ie. under the skull, placed on and within the brain itself. Noninvasive techniques require either wet or dry electrodes to be placed around the cranium. Raw electrical data is measured in Microvolts (uV) at observed time t producing wave patterns from t to t+n.

2.2 Human Emotion:

Human emotions are varied and complex but can be generalized into positive and negative categories. Some emotions overlap such as 'hope' and 'anguish', which are considered positive and negative respectively but that are often experienced.

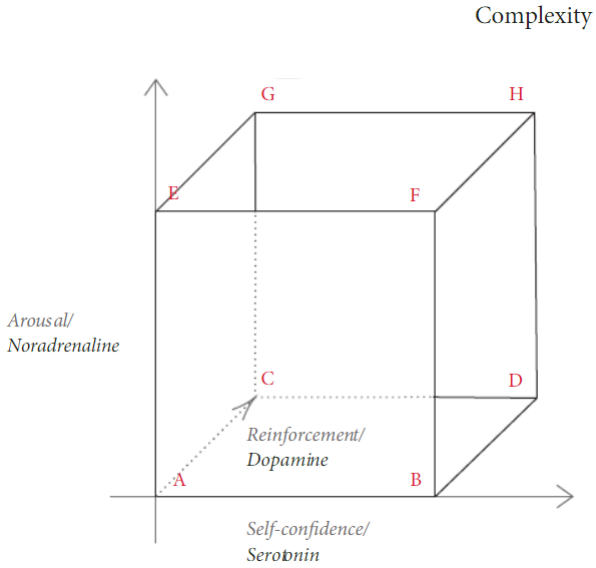

Figure 1: L¨ovheim's cube: mapping levels of noradrenaline, dopamine, and serotonin to human emotion.

Emotion Category	Emotion/Valence
A	Shame (Negative) Humiliation (Negative)
B	Contempt (Negative) Disgust (Negative)
C	Fear (Negative), Terror (Negative)
D	Enjoyment (Positive) Joy (Positive)
E	Distress (Negative) Anguish (Negative)
F	Surprise (Negative) (Lack of Dopamine)
G	Anger (Negative) Rage (Negative)
H	Interest (Positive) Excitement (Positive)

Table 1. Table to show Lövheim categories and their encapsulated emotions with a valence label.

Lövheim's three-dimensional emotional model maps brain chemical composition to generalized states of positive and negative valence. This is shown in Fig. 1 with emotion categories A-H from each of the model's vertices, further detailed in Table I. Each vertex of the cube represents a centroid of an emotional category. It is worth noting that categories are not completely concrete, and that emotions are experienced in gradient, as well as overlapping between categories It is this chemical composition that causes certain nervous oscillation and thus electrical brainwave activity. These chemical compositions can be mapped to emotions with positive and negative classes. Since emotions are encoded within chemical composition that directly influence electrical brain activity, the above study proposes that they can be classed using statistical features of the produced brainwaves.

The works [2][3] explores single and ensemble methods to classify emotional experiences based on EEG brainwave data. A commercial MUSE EEG headband is used with a resolution of four (TP9, AF7, AF8, TP10) electrodes. Positive and negative emotional states are invoked using film clips with an obvious valence, and neutral resting data is also recorded with no stimuli involved, all for one minute per session.

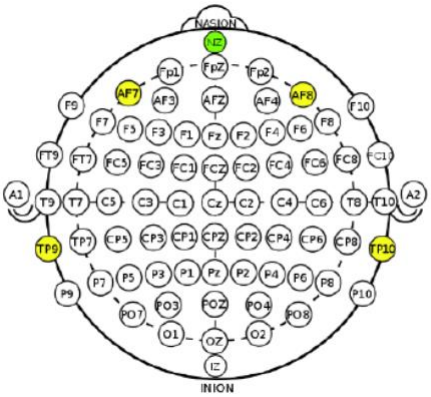

Figure 2: EEG sensors TP9, AF7, AF8, TP10 of the Muse headband on the international standard EEG placement system

2.3 Data Acquisition:

The dataset (emotional state) was based on whether a person was feeling positive, neutral, or negative emotions (https://www.kaggle.com/birdy654/eeg-brainwave-datasetfeeling-emotions). The study [2][3] employs four dry extra-cranial electrodes via a commercially available MUSE EEG headband. Micro voltage measurements are recorded from the TP9, AF7, AF8, and TP10 electrodes, as seen in figure 2. Because the signals are quite weak in nature, signal noise is a major issue due to it effectively masking useful information. The EEG headband employs various artefact separation techniques to best retain the brainwave data and discard unwanted noise. Sixty seconds of data were recorded from two subjects (1 male, 1 female, aged 20-22) for each of the 6 film clips found in Table II producing 12 minutes (720 seconds) of brain activity data (6 minutes for each emotional state). Six minutes of neutral brainwave data were also collected resulting in a grand total of 36 minutes of EEG data recorded from subjects. With a variable frequency resampled to 150Hz, this resulted in a dataset of 324,000 data points collected from the waves produced by the brain. Activities were exclusively stimuli that would evoke emotional responses from the set of emotions found in Table I and were considered by their valence labels of positive and negative rather than the emotions themselves. Neutral data were also collected, without stimuli and before any of the emotions data (to avoid contamination by the latter), for a third class that would be the resting emotional state of the subject. Three minutes of data were collected per day to reduce the interference of a resting emotional state.

Stimulus	Valence	Studio	Year
Marley and Me	Neg	Twentieth Century Fox, etc.	2008
Up	Neg	Walt Disney Pictures, etc.	2009
My Girl	Neg	Imagine Entertainment, etc.	1991
La La Land	Pos	Summit Entertainment, etc.	2016
Slow Life	Pos	BioQuest Studios	2014
Funny Dogs	Pos	MashupZone	2015

Table 2. Source of Film Clips used as Stimuli for EEG Brainwave Data Collection

Participants were asked to watch the film without making any conscious movements (eg. drinking coffee) to prevent the influence of Electromyographic (EMG) signals on the data due to their prominence over brainwaves in terms of signal strength. Observations of the experiment showed a participant smile for a short few second during the 'funny dogs' compilation clip, as well as become visibly upset during the 'Marley and Me' film clip (death scene). These facial expressions will influence the recorded data but are factored into the classification model because they accurately reflect behavior in the real world, where these emotional responses would also

occur. Hence, to accurately model realistic situations, both EEG and facial EMG signals are considered informative. To generate a dataset of statistical features, an effective methodology from a previous study [4] was used to extract 2400 features through a sliding window of 1 second beginning at t=0 and t=0.5. The down sampling was set to the minimum observed frequency of 150Hz.

2.4 Full Set of Features (Preselection):

This section describes the reasoning behind the necessity of performing statistical extraction, as well as the method to perform the process. The EEG sensor used for the experiments, the Muse headband, communicates with the computer using Bluetooth Low energy (BLE). The use of this protocol improves the autonomy of the sensor at the expense of a nonuniform sampling rate. The first step applied to normalize the dataset is using a Fourier-based method to resample the data to a fixed frequency of 200Hz. Brainwave data is nonlinear and nonstationary in nature, and thus single values are not indicative of class. That is, mental classification is based on the temporal nature of the wave, and not the values specifically. For example, a simplified concentrative and relaxed wave can be visually recognized due to the fact that wavelengths of concentrative mental state class data are far shorter, and yet, a value measured at any one point might be equal for the two states (i.e., x Microvolts). Additionally, the detection of the natures that dictate alpha, beta, theta, delta, and gamma waves also require analysis over time. It is for these reasons that temporal statistical extraction is performed. For temporal statistical extraction, sliding time windows of total length 1s are considered, with an overlap of 0.5 seconds. That is, windows run from [0s−1s), [1.5s − 2.5s), [2s – 3s), [2.5s – 3s), continuing until the experiment ends [3]. The remainder of this subsection describes the different statistical feature types which are included in the initial dataset:

(i) A set of values of signals within a sequence of temporal windows $x_1, x_2, x_3 \ldots x_n$ are considered and mean values are computed:

$$\mu \frac{1}{N} \sum_{N}^{i} x_i.$$

(ii) The standard deviation of values is recorded:

$$\sigma = \sqrt{\frac{1}{N} \sum_{N}^{i} (x_i - \mu)^2}.$$

(iii) Asymmetry and peakedness of waves are statistically represented by the skewness and kurtosis via the statistical moments of the third and fourth order.

Skewness:

$$y = \frac{\mu^k}{\sigma^k}$$

Kurtosis:

$$\mu^k = \frac{1}{N} \sum_{N}^{i} (x_i - \mu)^k$$

are taken where k=3rd and k=4th moment about the mean.

(iv) Max value within each particular time window {max $_1$, max $_2$ …… max $_n$}

(v) Minimum value within each time window {min $_1$, min $_2$ ..... min $_n$}.

(vi) Derivatives of the minimum and maximum values by dividing the time window in half and measuring the values from either half of the window.

(vii) Performing the min and max derivatives a second time on the presplit window, resulting in the derivatives of every 0.25s time window.

(viii) For every min, max, and mean value of the four 0.25s time windows, the Euclidean distance between them is measured. For example, the maximum value of time window one of four has its 1D Euclidean distance measured between it and max values of windows two, three, and four of four.

(ix) From the 150 features generated from quarter-second min, max, and mean derivatives, the last six features are ignored and thus a 12x12 (144) feature matrix can be generated. Using the Logarithmic Covariance matrix model [51], a log-cov vector and thus statistical features can be generated for the data as such.

$$lcM = U\left(\log m\left(\mathrm{cov}\left(M\right)\right)\right).$$

U returns the upper triangular features of the resultant vector and the covariance matrix (cov(M)) is

$$\mathrm{cov}\left(M\right) = \mathrm{cov}_{ij} = \frac{1}{N\sum_N^k \left(x_{ik} - \mu_i\right)\left(x_{kj} - \mu_j\right)}.$$

(x) For each full 1s time window, the Shannon Entropy is measured and considered as a statistical feature:

$$h = -\sum_j S_j \times \log\left(S_j\right).$$

The complexity of the data is summed up as such, where h is the statistical feature and S relates to each signal within the time window after normalization of values.

(xi) For each 0.5s time window, the log-energy entropy is measured as

$$\log e = \sum_i \log\left(S_i^2\right) + \sum_j \log\left(S_i^2\right),$$

w

where i is the first-time window n to n+0.5 and j is the second time window n+0.5 to n+1.

(xii) Analysis of a spectrum is performed by an algorithm to perform Fast Fourier Transform (FFT) [52] of every recorded time window, derived as follows:

$$X_k = \sum_{n=0}^{N-1} S_n^t e^{-i2\pi k(n/N)}, \quad k = 0, \ldots, N-1.$$

The above statistical features are used to represent the waves. With these features considered for each electrode and time window (including those formed by overlaps), this produces a total of 2147 scalars per measure.

2.5 Spiking Neural Network (SNNs)

SNNs are based on the idea that information in the brain is processed and communicated through the generation of electrical signals, or spikes, by individual neurons. These neurons generate output in the form of spikes, which are discrete electrical events that occur at specific points in time. When the input signal to a neuron exceeds a certain threshold, the neuron generates a spike, which is then transmitted to other neurons in the network. Neurons in an SNN are connected by synapses, which are the connections between neurons that transmit electrical signals. In SNNs, synapses are modeled as variable strength connections that can be modified by the arrival of spikes from other neurons. SNNs can learn to recognize patterns in the input data by modifying the strengths of the synapses between neurons. When a neuron generates a spike, it can cause the strength of the synapse between it and the receiving neuron to increase or decrease, depending on the timing of the spike. SNNs represent the behavior of each neuron and synapse using mathematical equations and simulating the flow of electrical spikes through the network over time.

3. Experimental/Modeling Design

Due to the complexity, randomness, and non-stationary aspects of brainwave data, classification is very difficult with a raw EEG stream. For this reason, stationary techniques such as time windowing must be introduced alongside feature extraction of the data within a window. There are many statistics that can be derived from such EEG windows, each of which has varying classification efficacy depending on the goal. Feature engineering must be performed to identify useful statistics from the EEG signals.

As mentioned in the Preselection section of the background, temporal statistical extraction was performed on the EEG signals using the above mentioned 12 feature engineering techniques from the sliding time windows of length 1 second with an overlap of 0.5 seconds. This resulted in 2548 individual features.

3.1 Feature Selection:

The resulting number of features is too large to be used in real time (i.e., it would be computationally intensive) and would not yield good classification results because of the large dimensionality. Attribute selection is therefore performed to overcome these limitations and, additionally, make the training process significantly faster. Attribute selection or feature selection is a method that identifies a significant subset of features by eliminating redundant features. This can improve the model performance, reduce model overfitting, reduce computational processing costs and can also speed up the model training and processing time.

The data was min-max normalized, and a Random Forest (RF) based feature selection technique was used on the 2548 features. 255 significant features were identified using the forest's feature importance.

3.2 Spiking Neural Network propagation:

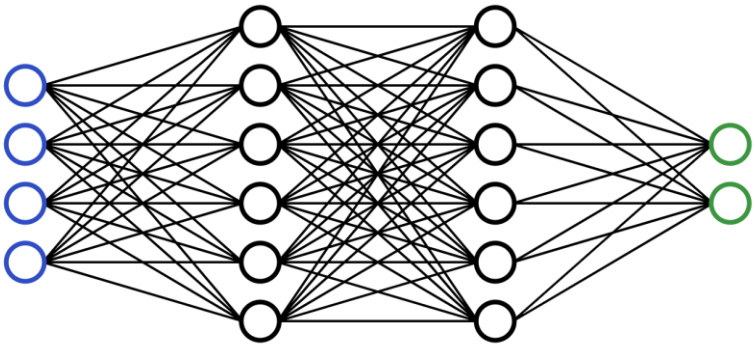

Figure 3. [5] Structure of the proposed brain-inspired spiking neural network architecture for emotion classification. The responses from the 255 significant features are encoded into spiking data and presented to an 255 × 80 × 30 X 3 spiking neural networks (SNN).

There are many different ways to implement spiking neural networks. One way to approach this is to follow the model that faithfully recreates the system of variables and equations that govern a biological neuron. While such a model may be most accurate to real life, it is also the most computationally intensive way to implement. Additionally, for a classification problem, it is difficult to translate a spiking output from time series data into discrete categories.

Another approach to spiking neural networks focuses less on fidelity and more on capturing the general idea of a neuron without sacrificing performance. The leaky integrate-and-fire neuron hardly qualifies as biologically accurate [Brunel]. However, it is good enough to be useful in practical applications such as classification.

While backpropagation does not directly translate to any biological process, it is so prevalent in machine learning that many implementations of spiking neural networks use it to train supervised models. This mismatch is especially apparent when considering the difficulty of obtaining a gradient for spiking neural networks, since spiking neurons are not differentiable. Therefore, a pseudo gradient is used to aid in backpropagation.

In our work, we tried to recreate such a faithful model. In the process, the above normalized data from the significant features were encoded into spike trains with 50 timesteps. A Leaky integrate-and-Fire(LIF) neuron model was used to model the neurons of our Spiking Neural Network with an input layer of dimension 255, two hidden layers and one output layer containing 3 output nodes representing the emotional states was built. These encoded spike trains are propagated through the network and the cross-entropy loss is calculated. The loss was used to optimize the network weights using stochastic gradient descent optimization technique as a process of back propagation. The synaptic connections between neurons will be modeled as conductance-based synapses. The strength of the synapses will be adjusted during training using a spike-timing-dependent plasticity (STDP) rule, such as the one proposed by Song et al. (2000)[6]. The STDP rule will ensure that the synaptic weights are adjusted in a way that strengthens connections between neurons that fire together and weakens connections between neurons that fire out of sync.

4. Results and Discussion

There are many neural signal processing studies done with various datasets using traditional high computational models but spiking neural network though being a low computational model has proven to be a reliable model. Our Random Survival Forest model used for feature selection has also shown significant results. Below are some of the alternative studies in this field.

Study	**Method**	**Accuracy**
Bos, et al. [7]	Fisher's Discriminant	94.9
Li, et al. [8]	Common Spatial Patterns	93.5
Li, et al	Linear SVM	93
Zheng, et al. [9]	Deep Belief Network	87.62
Koelstra, et al. [10]	Common Spatial Patterns	58.8
This study	Random Forest model	98.7
This study	Spiking Neural Network	88.56

Table 5. An Indirect Comparison of this Study to Similar Works Performed on Different Dataset

Using the spiking neural network model, the testing accuracy reached 88.56%, which is lower than the 98% baseline using the original random forest classifier. One reason for this performance could be that the EEG dataset has already been heavily filtered and processed such that the spiking neural network could not benefit from the time series nature of the data. Indeed, the paper that originally put forward the dataset mentioned that the data was preprocessed to render it more suitable to a conventional machine learning algorithm. If the original time series data were available, a spiking network may have more success using it. Other alternative approaches to improve the accuracy could be to

(i) Using additional datasets to train as SNNs typically require a larger dataset to achieve high accuracy compared to traditional machine learning algorithms. This is because SNNs learn by adjusting the strengths of connections between neurons, which can require more data to achieve accurate representations of the input.

(ii) Adjust the architecture of your network: There are many different architectural choices to make when building an SNN, including the number of layers, the number of neurons in each layer, and the connectivity pattern between neurons. Experimenting with different architectures can help you find the optimal design for your dataset.

(iii) Adjust the parameters of your network: There are many different parameters to tune in an SNN, including the learning rate, weight initialization, and regularization

strength. Experimenting with different values for these parameters can help you find the optimal settings for your dataset.

(iv) Using different neuron models and different neuron activations

Many further improvements could be made to the existing models, which could give promising results. In order to assess which improvements could be made, we shall implement this architecture using other EEG datasets to understand the limitations and capabilities of this model.

5. Conclusions

In conclusion, our study explored the potential of using a Spiking Neural Network model to estimate the emotional state of a person into positive, neutral and negative using EEG brain waves as input. The results showed that the SNN model achieved an accuracy of 88% on our test dataset, which is comparable to some of the best available models. The use of spiking neural networks has shown great promise in real-time data analysis, especially in situations where power usage and computational capabilities are limited. This study demonstrates the potential of using SNNs for other real-world applications, such as monitoring and treating patients with neurological disorders and restoring muscle and cognitive function through neural prosthetics. Further research can be conducted to improve the accuracy of the SNN model and explore other potential applications for neural signal processing. Overall, the results of this study contribute to the growing body of research on spiking neural networks and their potential use in real-time data analysis for various applications.

Some additional future research directions would involve having an enhanced understanding of brain activity and its relationship to neurological conditions and disorders. Finding ways to improve the Diagnostic capability of conditions such as epilepsy, depression, dementia, Alzheimer’s and Parkinson's disease. Improved treatment outcomes to optimize designs and advancements in brain computer interfaces developing neural prosthetics to restore function to patients with spinal cord injuries and other neurological conditions.

Acknowledgments

We would like to take this opportunity to express my sincere gratitude to my class professor, Prof. Konstantinos Michmizos, for his guidance, support, and encouragement throughout this term. Prof. Konstantinos’s insightful feedback and valuable suggestions have been instrumental in shaping this project. We have learned so much from his expertise and dedication, and we will carry these lessons with me in my academic and professional endeavors.

References

[1] M. S. El-Nasr, J. Yen, and T. R. Ioerger, “Flame - fuzzy logic adaptive model of emotions,” Autonomous Agents and Multi-agent systems, vol. 3, no. 3, pp. 219–257, 2000.------

[2] Bird, Jordan & Ekart, Aniko & Buckingham, Christopher & Faria, Diego. (2019). Mental Emotional Sentiment Classification with an EEG-based Brain-machine Interface.

[3] Bird, Jordan & Faria, Diego & Manso, Luis & Ekárt, A. & Buckingham, Christopher. (2019). A Deep Evolutionary Approach to Bioinspired Classifier Optimisation for Brain-Machine Interaction.

[4] Ding, W. and Marchionini, G. 1997. A Study on Video Browsing Strategies. Technical Report. University of Maryland at College Park.

[5] Vanarse, A., Espinosa-Ramos, J. I., Osseiran, A., Rassau, A., & Kasabov, N. (2020). Application of a brain-inspired spiking neural network architecture to odor data classification. Sensors, 20(10), 2756.

[6] Song S, Miller KD, Abbott LF. Competitive Hebbian learning through spike-timing-dependent synaptic plasticity. Nat Neurosci. 2000 Sep;3(9):919-26. doi: 10.1038/78829. PMID: 10966623.

[7] J. J. Bird, L. J. Manso, E. P. Ribiero, A. Ekart, and D. R. Faria, “A study on mental state classification using eeg-based brain-machine interface,” in 9th International Conference on Intelligent Systems, IEEE, 2018.

[8] S. Koelstra, A. Yazdani, M. Soleymani, C. Mühl, J.-S. Lee, A. Nijholt, T. Pun, T. Ebrahimi, and I. Patras, “Single trial classification of eeg and peripheral physiological signals for recognition of emotions induced by music videos,” in Int. Conf. on Brain Informatics, pp. 89– 100, Springer, 2010.

[9] X.-W. Wang, D. Nie, and B.-L. Lu, “Emotional state classification from eeg data using machine learning approach,” Neurocomputing, vol. 129, pp. 94–106, 2014.

[10] M. Abujelala, C. Abellanoza, A. Sharma, and F. Makedon, “Brainee: Brain enjoyment evaluation using commercial eeg headband,” in Proceedings of the 9th acm international conference on pervasive technologies related to assistive environments, p. 33, ACM, 2016.

Appendix:

Our project link:

https://colab.research.google.com/drive/1xnt-zQAWIoAM28UwtOUGJjgj80GBoi94?usp=sharing

code:

```
# # -*- coding: utf-8 -*-
# """Pradeep_2.ipynb

# Automatically generated by Colaboratory.

# Original file is located at
#     https://colab.research.google.com/drive/1xnt-zQAWIoAM28UwtOUGJjgj80GBoi94

# #### https://www.kaggle.com/datasets/birdy654/eeg-brainwave-dataset-feeling-emotions
# """

# import pandas as pd

# from google.colab import drive
# drive.mount('/content/drive')

# # Commented out IPython magic to ensure Python compatibility.
# # %cd /content/drive/MyDrive/BIC/Project

# data = pd.read_csv('emotions.csv')

# """# Train and Test Split"""

# data

# from sklearn.model_selection import train_test_split
# X_train, X_test, y_train, y_test = train_test_split(data.iloc[:,:-1], data.iloc[:,-1],
#                                                     stratify=data.iloc[:,-1],
#                                                     test_size=0.25)

# y_train.value_counts()

# sum = 537+531+531
# (537/sum),(531/sum),(531/sum)

# y_test.value_counts()
```

```
# sum = 179+177+177
# (179/sum),(177/sum),(177/sum)

# data.iloc[:,-1].value_counts()

# X_train

# X_test

# """# Normalizing data and Feature Selection"""



# from sklearn.preprocessing import MinMaxScaler
# scaler = MinMaxScaler()
# normalized_train = scaler.fit_transform(X_train)
# normalized_test = scaler.transform(X_test)

# normalized_train = pd.DataFrame(normalized_train)
# normalized_train.columns = X_train.columns
# normalized_train.index = X_train.index
# normalized_train

# normalized_test = pd.DataFrame(normalized_test)
# normalized_test.columns = X_test.columns
# normalized_test.index = X_test.index
# normalized_test

# import pandas as pd
# from sklearn.ensemble import RandomForestClassifier
# from sklearn.feature_selection import SelectFromModel

# sel = SelectFromModel(RandomForestClassifier(n_estimators = 1000 , n_jobs = -1  , oob_score= True))
# sel.fit(normalized_train, y_train)
# sel.get_support()

# selected_feat= normalized_train.columns[(sel.get_support())]
# len(selected_feat)

# print(selected_feat)

# model = RandomForestClassifier(n_estimators= 1000 , n_jobs= -1 , oob_score= True)
# model.fit(X_train,y_train)

# normalized_train = normalized_train[selected_feat]
```

```
# normalized_test = normalized_test[selected_feat]

# normalized_train

# model = RandomForestClassifier(n_estimators= 1000 , n_jobs= -1 , oob_score= True)
# model.fit(normalized_train,y_train)

# y_pred = model.predict(normalized_test)

# from sklearn.metrics import accuracy_score
# from sklearn.metrics import f1_score
# from sklearn.metrics import precision_score
# from sklearn.metrics import recall_score
# from sklearn.metrics import roc_auc_score
# from sklearn.metrics import confusion_matrix

# accuracy = accuracy_score(y_test, y_pred)
# f1 = f1_score(y_test, y_pred, average=None)
# precision = precision_score(y_test, y_pred, average=None)
# recall = recall_score(y_test, y_pred, average=None)
# cf = confusion_matrix(y_test, y_pred)

# print("Accuracy :{} \nF1-Score : {} \nPrecisoin : {} \nRecall : {}  \nConfusion Matrix \n {}
".format(accuracy,f1,precision,recall,cf))



# '''
# import numpy as np
# #dropping correlated features
# # Create correlation matrix
# corr_matrix = normalized_train.corr().abs()
# upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))
# to_drop = [column for column in upper.columns if any(upper[column] > 0.95)]

# normalized_train = normalized_train.drop(to_drop, axis=1)
# normalized_test = normalized_test.drop(to_drop , axis =1)
# '''

# imp_feat = normalized_train.columns

# imp_feat = pd.DataFrame(imp_feat)
# imp_feat.to_csv('imp_feat.csv')

# # model = RandomForestClassifier(n_estimators= 1000 , n_jobs= -1 , oob_score= True)
```

```
# # model.fit(normalized_train,y_train)

# # y_pred = model.predict(normalized_test)

# # from sklearn.metrics import accuracy_score
# # from sklearn.metrics import f1_score
# # from sklearn.metrics import precision_score
# # from sklearn.metrics import recall_score
# # from sklearn.metrics import roc_auc_score
# # from sklearn.metrics import confusion_matrix

# # accuracy = accuracy_score(y_test, y_pred)
# # f1 = f1_score(y_test, y_pred, average=None)
# # precision = precision_score(y_test, y_pred, average=None)
# # recall = recall_score(y_test, y_pred, average=None)
# # cf = confusion_matrix(y_test, y_pred)

# # print("Accuracy :{} \nF1-Score : {} \nPrecisoin : {} \nRecall : {}  \nConfusion Matrix \n {}
".format(accuracy,f1,precision,recall,cf))

# Y_Train = y_train.replace( to_replace= ['POSITIVE','NEUTRAL','NEGATIVE'] , value = [0,1,2])
# Y_Test = y_test.replace( to_replace= ['POSITIVE','NEUTRAL','NEGATIVE'] , value = [0,1,2])

# train_data = pd.concat([normalized_train,Y_Train] , axis = 1 , join = 'inner')
# test_data = pd.concat([normalized_test,Y_Test] , axis = 1 , join = 'inner')

# train_data

# """# Setup for Spiking Neural Network with Back Propogation"""

# import torch
# import torch.nn as nn
# import torchvision
# import torchvision.transforms as transforms
# from torch.utils.data import DataLoader, sampler
# import torch.optim as optim
# import os
# import time
# import pickle
# import numpy as np

# import torch
# from torch.utils.data import Dataset, DataLoader

# class MyDataset(Dataset):
```

```
#     def __init__(self, data):
#         self.X = data.iloc[:, :-1].values  # get all columns except the last one
#         self.y = data.iloc[:, -1].values   # get the last column as target labels

#     def __len__(self):
#         return len(self.X)

#     def __getitem__(self, idx):
#         return torch.tensor(self.X[idx]), torch.tensor(self.y[idx])

# class PseudoSpikeRect(torch.autograd.Function):
#     """ Rectangular Pseudo-grad function """

#     @staticmethod
#     def forward(ctx, input, vth, grad_win, grad_amp):
#         """
#         Args:
#             input (Torch Tensor): Input tensor containing voltages of neurons in a layer
#             vth (Float): Voltage threshold for spiking
#             grad_win (Float): Window for computing pseudogradient
#             grad_amp (Float): Amplification factor for the gradients

#         Returns:
#             output (Torch Tensor): Generated spikes for the input

#         Write the operation for computing the output spikes from the input. The operation should be vectorized, i.e.
no loops.
#         """

#         #Saving variables for backward pass. Nothing to do here
#         ctx.save_for_backward(input)
#         ctx.vth = vth
#         ctx.grad_win = grad_win
#         ctx.grad_amp = grad_amp

#         #Compute output from the input. No loops. Hint: Use Pytorch "greater than" function.
#         output = torch.gt(input , vth).float()

#         return output

#     @staticmethod
#     def backward(ctx, grad_output):
#         """
#         Args:
#             grad_output (Torch Tensor): Gradient of the output
```

```
#         Returns:
#             grad (Torch Tensor): Gradient of the input

#         Write the operation for computing the output spikes from the input. The operation should be vectorized, i.e. no loops.
#         """

#         #Retrieving variables from forward pass. Nothing to do here.
#         input, = ctx.saved_tensors
#         vth = ctx.vth
#         grad_win = ctx.grad_win
#         grad_amp = ctx.grad_amp
#         grad_input = grad_output.clone()

#         #compute the gradient of the input using rectangular pseudograd function
#         spike_pseudo_grad = torch.lt(torch.abs(torch.sub(input , vth)) , grad_win)

#         #Multiplying by gradient amplifier. Nothing to do here
#         grad = grad_amp * grad_input * spike_pseudo_grad.float()
#         return grad, None, None, None

# class LinearIFCell(nn.Module):
#     """ Leaky Integrate-and-fire neuron layer"""

#     def __init__(self, psp_func, pseudo_grad_ops, param):
#         """
#         Args:
#             psp_func (Torch Function): Pre-synaptic function
#             pseudo_grad_ops (Torch Function): Pseudo-grad function
#             param (tuple): Cell parameters (Voltage Threshold, gradient window, gradient amplitude)

#         This function is complete. You do not need to do anything here.
#         """
#         super(LinearIFCell, self).__init__()
#         self.psp_func = psp_func
#         self.pseudo_grad_ops = pseudo_grad_ops
#         self.vdecay, self.vth, self.grad_win, self.grad_amp = param

#     def forward(self, input_data, state):
#         """
#         Forward function
#         Args:
#             input_data (Tensor): input spike from pre-synaptic neurons
#             state (tuple): output spike of last timestep and voltage of last timestep
```

```
#         Returns:
#             output: output spike
#             state: updated neuron states

#         Write the operation for integrating the presynaptic spikes into voltage.
#         """
#         pre_spike, pre_volt = state

#         #Compute the voltage from the presynaptic inputs. This should be a vectorized operation. No loops.
#         volt = pre_volt * self.vdecay *  (1. - pre_spike) + self.psp_func(input_data)

#         #Compute the spike output by using the pseudo_grad_ops function. This should be a vectorized operation. No loops.
#         output = self.pseudo_grad_ops(volt , self.vth  , self.grad_win , self.grad_amp)
#         return output, (output, volt)

# class SingleHiddenLayerSNN(nn.Module):
#     """ SNN with single hidden layer """

#     def __init__(self, input_dim, output_dim, hidden_dim1, hidden_dim2 , param_dict):
#         """
#         Args:
#             input_dim (int): input dimension
#             output_dim (int): output dimension
#             hidden_dim (int): hidden layer dimension
#             param_dict (dict): neuron parameter dictionary for each layer (Voltage Threshold, gradient window, gradient amplitude)

#         Create hidden and output layers using implementation of the layer in Q2. and using nn.Linear as the psp function.
#         """
#         super(SingleHiddenLayerSNN, self).__init__()
#         self.input_dim = input_dim
#         self.output_dim = output_dim
#         self.hidden_dim1 = hidden_dim1
#         self.hidden_dim2 = hidden_dim2
#         pseudo_grad_ops = PseudoSpikeRect.apply

#         #Create the hidden layers. Assume that the hidden layer neuron parameters are in param_dict['hid_layer']. Set bias=False for nn.Linear.
#         self.hidden_cell1 = LinearIFCell(nn.Linear(input_dim , hidden_dim1 , bias = False), pseudo_grad_ops , param_dict['hid_layer1'] )
#         self.hidden_cell2 = LinearIFCell(nn.Linear(hidden_dim1 , hidden_dim2 , bias = False), pseudo_grad_ops , param_dict['hid_layer2'] )
```

```
#         #Create the output layer. Output layer params are in param_dict['out_layer']. Set bias=False for nn.Linear.
#         self.output_cell = LinearIFCell(nn.Linear(hidden_dim2, output_dim , bias = False), pseudo_grad_ops , 
param_dict['out_layer'] )


#     def forward(self, spike_data, init_states_dict, batch_size, spike_ts):
#         """
#         Forward function
#         Args:
#             spike_data (Tensor): spike data input (batch_size, input_dim, spike_ts)
#             init_states_dict (dict): initial states for each layer- 'hid_layer' for hidden layer; 'out_layer' for output layer.
#             batch_size (int): batch size
#             spike_ts (int): spike timesteps
#         Returns:
#             output: number of spikes of output layer

#         Write the operations for propagating the input through the network and computing the spike outputs.
#         """
#         hidden_state1, hidden_state2 , out_state = init_states_dict['hid_layer1'],init_states_dict['hid_layer2'] , 
init_states_dict['out_layer']
#         spike_data_flatten = spike_data.view(batch_size, self.input_dim, spike_ts)
#         output_list = [] #List to store the output at each timestep
#         for tt in range(spike_ts):
#             #Retrieve the input at time tt
#             input_tt = spike_data_flatten[:,:,tt]

#             #Propagate through the hidden layer
#             hidden_output1, hidden_state1 = self.hidden_cell1.forward(input_tt, hidden_state1)
#             hidden_output2, hidden_state2 = self.hidden_cell2.forward(hidden_output1, hidden_state2)
#             #Propagate through the output layer
#             output, out_state = self.output_cell.forward(hidden_output2, out_state)

#             #Append output spikes to output list
#             output_list.append(output)

#         #Sum the outputs to compute spike count for each output neuron. Torch.stack and Torch.sum might be useful 
here. No loops
#         output = torch.sum(torch.stack(output_list), dim=0)

#         return output

# class WrapSNN(nn.Module):
#     """ Wrapper of SNN """

#     def __init__(self, input_dim, output_dim, hidden_dim1 , hidden_dim2, param_dict, device):
```

```
#         """
#         Args:
#             input_dim (int): input dimension
#             output_dim (int): output dimension
#             hidden_dim (int): hidden layer dimension
#             param_dict (dict): neuron parameter dictionary
#             device (device): device
#         """
#         super(WrapSNN, self).__init__()
#         self.input_dim = input_dim
#         self.output_dim = output_dim
#         self.hidden_dim1 = hidden_dim1
#         self.hidden_dim2 = hidden_dim2
#         self.device = device
#         self.snn = SingleHiddenLayerSNN(input_dim, output_dim, hidden_dim1 , hidden_dim2 , param_dict)

#     def forward(self, spike_data):
#         """
#         Forward function
#         Args:
#             spike_data (Tensor): spike data input
#         Returns:
#             output: number of spikes of output layer
#         """
#         batch_size = spike_data.shape[0]
#         spike_ts = spike_data.shape[-1]
#         init_states_dict = {}
#         # Hidden layer 1
#         hidden_volt1 = torch.zeros(batch_size, self.hidden_dim1, device=self.device)
#         hidden_spike1 = torch.zeros(batch_size, self.hidden_dim1, device=self.device)
#         init_states_dict['hid_layer1'] = (hidden_spike1, hidden_volt1)

#         # Hidden layer 2
#         hidden_volt2 = torch.zeros(batch_size, self.hidden_dim2, device=self.device)
#         hidden_spike2 = torch.zeros(batch_size, self.hidden_dim2, device=self.device)
#         init_states_dict['hid_layer2'] = (hidden_spike2, hidden_volt2)

#         # Output layer
#         out_volt = torch.zeros(batch_size, self.output_dim, device=self.device)
#         out_spike = torch.zeros(batch_size, self.output_dim, device=self.device)
#         init_states_dict['out_layer'] = (out_spike, out_volt)
#         # SNN
#         output = self.snn(spike_data, init_states_dict, batch_size, spike_ts)
#         return output
```

```
# #def sample_to_event_sample(sample, Y_Train , snn_timestep):
# def sample_to_event_sample(sample , device , snn_timestep):
#     """
#     Transform data sample to spikes, also called an event sample
#     Args:
#         sample (Tensor): sample of shape batch_size x imp_feat_size x 1
#         device (device): device (can be either CPU or GPU)
#         snn_timestep (int): spike timestep
#     Returns:
#         event_sample : event sample - spike encoding of the sample

#     Complete the expression for converting the sample to spikes (event sample)
#     """

#     batch_size = sample.shape[0]
#     sample_size = sample.shape[1]
#     sample = sample.view(batch_size , sample_size , 1 )
#     #Generate a random sample of the shape batch_size x imp_feat_size x  snn_timestep.
#     random_sample = torch.rand([batch_size, sample_size, snn_timestep] , device = device)
#     #Generate the event sample
#     event_sample  = torch.gt(sample , random_sample).float()
#     return event_sample

# def stbp_snn_training(network, train_dataset , test_dataset , spike_ts, device, batch_size=40, test_batch_size=40,
epoch=50):
#     """
#     STBP SNN training
#     Args:
#         network (SNN): STBP learning SNN
#         spike_ts (int): spike timestep
#         device (device): device
#         batch_size (int): batch size for training
#         test_batch_size (int): batch size for testing
#         epoch (int): number of epochs
#     Returns:
#         train_loss_list: list of training loss for each epoch
#         test_accuracy_list: list of test accuracy for each epoch
#     """

#     # Train and test dataloader
#     train_dataloader = DataLoader(train_dataset, batch_size=batch_size,
#                                   shuffle=False, num_workers=4)
#     test_dataloader = DataLoader(test_dataset, batch_size=test_batch_size,
#                                  shuffle=False, num_workers=4)
```

```
#     # Next we need to define a criteria for computing the loss.
#     # Refer to https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html on how to define the cross entropy loss.
#     # Note that you just need to define the loss here (and not compute it)

#     criterion = nn.CrossEntropyLoss()

#     #Define an optimizer that will perform the weight updates. You can find more about the optimizers in PyTorch here: https://pytorch.org/docs/stable/optim.html
#     #Taking the help of the documentation above, create an optimizer for stochastic gradient descent (SGD).
#     optimizer = torch.optim.SGD(network.parameters(), lr=0.0003, momentum=0.05)


#     # List for saving loss and accuracy
#     train_loss_list, test_accuracy_list, test_loss_list = [], [],[]
#     test_num = len(test_dataset)

#     # Start training

#     #Put the network on the device (typically GPU but can also be cpu)
#     network.to(device)

#     #Loop for the epochs
#     for ee in range(epoch):
#         #Keep track of running loss
#         running_loss = 0.0
#         running_batch_num = 0
#         running_test_batch_num = 0
#         train_start = time.time()

#         #Iterate over the training data in train dataloader
#         for data in train_dataloader:

#             #Retrieve the sample and label from data
#             sample, label = data

#             #Put the sample and labels on the device
#             sample, label = sample.to(device), label.to(device)

#             #Convert sample to event samples
#             event_sample = sample_to_event_sample(sample, device, spike_ts)

#             #Before we backprop, we need to set the gradients for each tensor to zero. This is done using the zero_grad function in Pytorch
```

```
#             optimizer.zero_grad()

#             #Compute the network output for the event sample
#             output = network(event_sample)

#             #Compute the loss using the criterion defined previously. Store in a variable called loss
#             loss = criterion(output, label)

#             #Backpropagate the loss through the network. Use Pytorch backward() function:
https://pytorch.org/docs/stable/generated/torch.Tensor.backward.html
#             loss.backward()

#             #Update the network weights by taking an optimizer 'step'.
#             #You can learn how to do that here: https://pytorch.org/docs/stable/optim.html#taking-an-optimization-step
#             optimizer.step()

#             #Updating tracking variables. Nothing to do here
#             running_loss += loss.item()
#             running_batch_num += 1
#         train_end = time.time()
#         train_loss_list.append(running_loss / running_batch_num)
#         print("Epoch %d Training Loss %.4f" % (ee, train_loss_list[-1]), end=" ")


#         #This ends one training iteration. After every training iteration, we can evaluate how well the network does
on data that it has not seen before.
#         #This step is called testing and is done on test dataset.

#         #Counter to keep track of the number of correct predictions
#         test_correct_num = 0
#         test_start = time.time()
#         with torch.no_grad():
#             for data in test_dataloader:

#                 #Retrieve the sample and label from test data
#                 sample, label = data

#                 #Put the sample and labels on the device
#                 sample, label = sample.to(device), label.to(device)

#                 #Convert the sample into event samples
#                 event_sample = sample_to_event_sample(sample, device, spike_ts)

#                 #Compute the network predictions and store in a variable called outputs
#                 outputs = network(event_sample)
```

```
#                 loss = criterion(outputs, label)
#                 running_loss += loss.item()
#                 #Get the class label as the largest activation. This is complete. Nothing to do here.
#                 _, predicted = torch.max(outputs, 1)

#                 #Compare the network predictions against the true labels and update the counter for correct predictions.
No loops.
#                 test_correct_num += (predicted == label).sum().item()
#                 running_test_batch_num += 1

#         #Updating tracking variables. Nothing to do here
#         avg_test_loss = running_loss / running_test_batch_num
#         test_loss_list.append(avg_test_loss)
#         test_end = time.time()
#         test_accuracy_list.append(test_correct_num / test_num)
#         print("Test Accuracy %.4f Training Time: %.1f Test Time: %.1f" % (
#             test_accuracy_list[-1], train_end - train_start, test_end - test_start))

#     #Return the loss and accuracies. Nothing to do here.
#     print("End Training")
#     network.to('cpu')
#     return train_loss_list, test_accuracy_list , test_loss_list

# """# Running the model """



# # Define the device on which training will be performed
# device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# torch.cuda.empty_cache()
# #Define the input dimensions in a variable
# input_dim = train_data.shape[1] -1

# #Define the output dimensions in a variable
# output_dim = 3

# #Define the hidden layer dimension in a variable
# hidden_dim1 = 80
# hidden_dim2 = 30

# #Create a dictionary of the neuron parameters for the hidden and output layer. The keys should be 'hid_layer' and
'out_layer'.
# #The values of the dictionary should be a list of the neuron parameters for each layer where the list elements are
[vdecay, vth, grad_win, grad_amp]
# neuron_params = {
```

```
#     'hid_layer1': [0.5, 0.5, 0.5, 1.0],
#     'hid_layer2': [0.5, 0.5, 0.5, 1.0],
#     'out_layer': [0.5, 0.5, 0.5, 1.0]
# }

# #Define snn timesteps in a variable
# spike_ts = 50

# #Create the SNN using the class definition in 3b and the arguments defined above
# snn = WrapSNN(input_dim, output_dim, hidden_dim1, hidden_dim2 , neuron_params, device)

# #Define the following training parameters
# #Batch size for training
# train_batch_size = 60

# #Batch size for testing
# test_batch_size = 60

# #Epochs
# epochs = 200

# train_dataset = MyDataset(train_data)
# test_dataset =  MyDataset(test_data)
# #Train the snn using the above arguments and the definition in Q5.
# train_loss_list, test_accuracy_list, test_loss_list = stbp_snn_training(snn, train_dataset, test_dataset, spike_ts,
device, train_batch_size, test_batch_size, epochs)

# test_data.shape

# test_dataloader = DataLoader(test_dataset, batch_size= test_data.shape[0],
#                              shuffle=False)

# for data in test_dataloader:
#     #Retrieve the sample and label from test data
#     sample, label = data
#     #Put the sample and labels on the device
#     sample, label = sample.to(device), label.to(device)
#     #Convert the sample into event samples
#     event_sample = sample_to_event_sample(sample, device, spike_ts)
#     #Compute the network predictions and store in a variable called outputs
#     outputs = snn(event_sample)

# y_pred = torch.argmax(outputs , dim = 1)

# y_pred = y_pred.numpy()
```

```
# y_test = Y_Test.values

# from sklearn.metrics import accuracy_score
# from sklearn.metrics import f1_score
# from sklearn.metrics import precision_score
# from sklearn.metrics import recall_score
# from sklearn.metrics import roc_auc_score
# from sklearn.metrics import confusion_matrix

# accuracy = accuracy_score(y_test, y_pred)
# f1 = f1_score(y_test, y_pred, average=None)
# precision = precision_score(y_test, y_pred, average=None)
# recall = recall_score(y_test, y_pred, average=None)
# cf = confusion_matrix(y_test, y_pred)

# print("Accuracy :{} \nF1-Score : {} \nPrecisoin : {} \nRecall : {}  \nConfusion Matrix \n {}
".format(accuracy,f1,precision,recall,cf))
```