The University of British Columbia

MANU 465

Capstone Project

Predicting Worker Fatigue Using Machine Learning

Anant Goyal 46894325

Alberto Mussali 50684182

Musa Habib 25899808

Sadul Bombuwala 76343292

December 3, 2021

Table of Contents

1. Abstract **4**

2. Introduction **5**

2.1. Motivation 5

2.2. Project Goal 5

3. Industry Applications **6**

4. Background and Overview **7**

4.1. Neural Oscillations 7

4.2. Electroencephalography (EEG) 8

5. Data Collection **8**

5.1. Muse 2 8

5.2. MindMonitor App 9

5.3. Obtaining Data 10

5.4. Data Output 10

6. Theoretical Framework **11**

6.1. State of Literature 11

6.2. Feature Extraction 12

7. Data Preprocessing **14**

7.1. Initial Preprocessing 14

7.2. Feature Generation 15

7.3. Further Processing Steps 17

7.4. Principal Component Analysis 17

7.5. Image Generation 19

8. Machine Learning Models **20**

8.1. Artificial Neural Networks 21

8.1.1. Theory 21

8.1.2. Architecture 22

8.2. Convolutional Neural Networks 23

8.2.1. Theory 23

8.2.2. Architecture 24

8.3. Random Forest 26

8.4. Logistic Regression 27

8.5. Model Training 27

9. Results **27**

10. Conclusions 29

11. References 30

12. Appendix A – Code 31

13. Appendix B – Sample RAW Data 57

1. Abstract

Machine learning is still in its early stages of adoption when it comes to real life applications, however, one place it can be useful is ensuring the safety of workers and their level of fatigue when performing their work. Using a Muse 2 device and the MindMonitor mobile application, EEG data from various fatigued (< 4 hours of sleep) and not fatigued individuals was recorded. The data then underwent several preprocessing steps, feature generation methods, and Principal Component Analysis (PCA) before being used to generate images. Both the images and the PCA-transformed dataset were then used to train Artificial and Convolutional Neural Networks. These machine learning models achieved near 100% accuracy as measured through a 3-fold Cross Validation procedure. However, the measured accuracy is not reflective of true model skill due to suspected data overfitting.

2. Introduction

2.1. Motivation

Possibly the most incredible aspect of machine learning and artificial intelligence is the ability to mimic human-like decision making. This study was motivated by a fascination with how Artificial Neural Networks (ANNs) are constructed and how they work. This project was also a means to further study and understand the complexities of our minds which contain their own biological neural networks. Collecting and analyzing brainwave data is a unique opportunity, and it is worthwhile to gain a valuable understanding of how brainwave data within humans can be used for research and applications in the world of artificial intelligence.

2.2. Project Goal

The objective of this study was to be able to use machine learning models to predict whether or not a person is fatigued, simply based on an individual's brainwave data. For this study, we defined “fatigued” to be an individual who had gotten less than 4 hours of sleep throughout the last 24 hours and a “not fatigued” or “normal” subject to have gotten a minimum of 6 hours of sleep and feeling well rested.

The Muse 2 is a multi-sensor meditation device that provides real-time feedback on brain activity, heart rate, breathing, and body movements to help users build a consistent meditation practice. When paired with the MindMonitor phone application, one can view and analyze the neural oscillation readings picked up by the Muse 2 in real-time, and use this headband for purposes beyond mediation. This study uses these tools to collect data on the variations in brain activity when an individual is in a sleep-deprived state of mind, and compares this to when they are well rested. Machine learning algorithms were then applied to the data, which accurately predicted the level of sleep deprivation of an individual.

3. Industry Applications

Predicting whether a person is fatigued or not through a quick analysis of their brainwave patterns has many useful applications to real world situations, including within the manufacturing industry.

One such manufacturing example would be to ensure worker safety in a factory setting. Oftentimes workers are handling dangerous, heavy machinery that requires their full attention in order to perform the work safely. In such a situation, machine learning models can be deployed to ensure workers are not actively working when they are fatigued, as sleep induced fatigue has been attributed to 13% of workplace injuries and costs US businesses more than $100 billion per year (National Safety Council, 2021).

Another situation where determining if a person is fatigued or not can be extremely beneficial is for ensuring driver safety. According to a study by the AAA Foundation for Traffic Safety, it is estimated that 328,000 drowsy driving crashes occur annually in the US alone (Tefft, 2014). By ensuring law enforcement officials can determine if a driver is likely to fall asleep behind the wheel, many of these accidents can be prevented.

Lastly, predicting fatigue can be useful in applications where decision making and motor skills are vital. Some examples are nurses, doctors and construction workers all of whom have critical roles when it comes to public safety. Such jobs are also more likely to see fatigued workers as they often require long hours and night shifts.

It is important to understand the relevance of such a study as it can go beyond purely research and be deployed to mitigate critical real-world problems.

4. Background and Overview

This section will explore the scientific background regarding neural oscillations, more popularly referred to as brainwaves.

4.1. Neural Oscillations

Neural oscillations are the repetitive patterns of neural activity that are observed in the Central Nervous System. These oscillations can be characterized by their amplitude, phase, and frequency by performing time-frequency analysis on a recording of neural activity.

There are several well-known "bands" of neural activity that can be measured via EEG. Each band is associated with different kinds of behaviors and pathologies. The following is a comparison of all the EEG bands:

Table 1: Comparison of EEG bands

Band	Frequency (Hz)	Location	Normally
Delta	< 4	frontally in adults, posteriorly in children; high-amplitude waves	adult slow-wave sleep in babies Has been found during some continuous-attention tasks
Theta	4–7	Found in locations not related to task at hand	higher in young children drowsiness in adults and teens idling Associated with inhibition of elicited responses (has been found to spike in situations where a person is actively trying to repress a response or action).
Alpha	8–15	posterior regions of head, both sides, higher in amplitude on dominant side. Central sites (c3-c4) at rest	relaxed/reflecting closing the eyes Also associated with inhibition control, seemingly with the purpose of timing inhibitory activity in different locations across the brain.
Beta	16–31	both sides, symmetrical distribution, most evident frontally; low-amplitude waves	range span: active calm → intense → stressed → mild obsessive active thinking, focus, high alert, anxious
Gamma	> 32	Somatosensory cortex	Displays during cross-modal sensory processing (perception that combines two different senses, such as sound and sight) Also is shown during short-term memory matching of recognized objects, sounds, or tactile sensations
Mu	8–12	Sensorimotor cortex	Shows rest-state motor neurons.

(Wikipedia contributors, Electroencephalography)

The aforementioned frequency bands are not standardized in literature, MindMonitor defines the frequencies as:

Table 2: Frequency band definitions

Band	Frequency Range (Hz)
Delta	1-4
Theta	4-8
Alpha	7.5-13
Beta	13-30
Gamma	30-44

(Clutterbuck, n.d.)

4.2. Electroencephalography (EEG)

Electroencephalography (EEG) is a powerful medical diagnosis tool that has been in use since the early 20th century. It allows the electrical activity of the brain to be monitored. Specifically, EEG measures the voltage fluctuations in the brain which result from neuronal activity. While the voltage fluctuations of a single neuron are too minute to be detected, an EEG device is capable of detecting the "summation of the synchronous activity of thousands or millions of neurons that have similar spatial orientation." (Wikipedia Contributors, Electroencephalography)

5. Data Collection

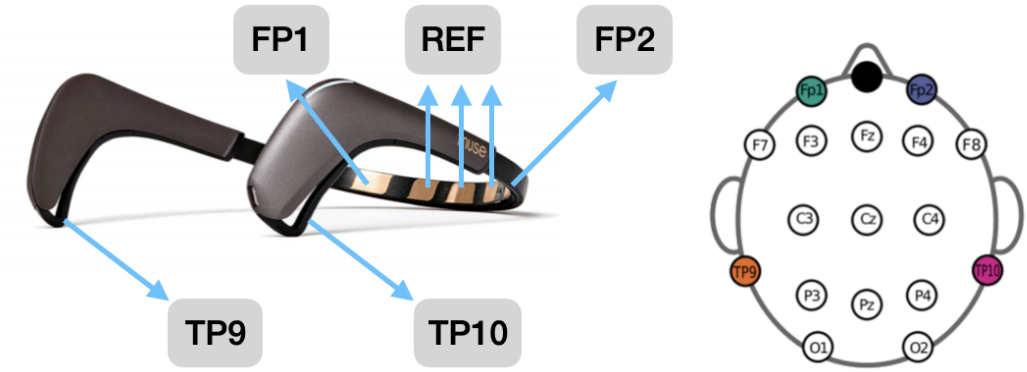
This section will explore how data was collected for the present work, as well as the apparatus used to collect said data and how it can be interpreted.

5.1. Muse 2

The Muse 2 device is a “multi-sensor meditation device” (Muse) in the form of a headband. The device contains four electrodes, with the capacity of adding an additional one through the micro-USB port. In addition, the device contains an accelerometer, gyroscope, and continuously monitors for jaw clenching and blinking via a combination of its sensors. The following figure is an image of the Muse 2 device with the electrodes labeled, as well as the locations for the electrodes in a

subject's head. TP and FP are acronyms for pre-temporal lobe and prefrontal lobe respectively, according to the *International 10-20 System*.

Figure 1: A Muse 2 device with the electrodes labeled and electrode position on subject's head

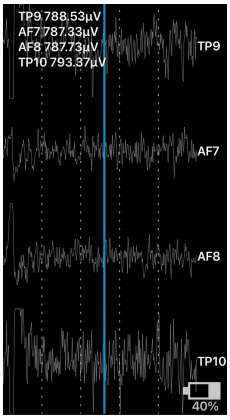


Each of the aforementioned electrodes record the immediate voltage magnitude in μV with a sampling frequency of 1 Hz. The quality of readings from the copper-coated electrodes can be improved by cleaning the subject's forehead and the copper plates with a damp cloth.

5.2. MindMonitor App

The MindMonitor application was used throughout this project as the primary data collection software. It allows for the Muse's electrode readings to be viewed and analyzed in real-time and to be saved to a .CSV file. The MindMonitor application also offers the capability to process the data; however, this functionality was not used since only the raw electrode, accelerometer, and gyroscope readings were used in this analysis. The following figure illustrates how the raw data is collected for each of the electrodes.

Figure 2: Raw electrode readings in the MindMonitor App



5.3. Obtaining Data

In order to collect the data, subjects' heads were first wiped with a damp cloth on the forehead and behind the ears. Similarly, the Muse 2 device was also cleaned with a wipe cloth as doing so allows the copper electrodes to retain some humidity for connection quality improvement. The following figure shows how the Muse 2 device was set up on a subject.

Figure 3: Data recording being performed on a subject

Data recording would go for as long as the subject was willing to, battery permitting. The data would then be exported as a .CSV file and stored for further analysis.

5.4. Data Output

As was explained in Section 4.1, neural oscillations are sorted into categories or "bands" based on the frequencies that make them up (see Tables 1 and 2). The Muse 2 processes and outputs data samples into their respective categories such that these bands can be differentiated from each other. In order to convert the raw electrical signals picked up by each electrode on the headband into this categorical form, the headband performs a Fast Fourier Transform (FFT) on the raw data before saving it. This computes the power spectral density of each frequency measured by each electrode on the headband. In essence, through Fourier analysis, each signal from each electrode on the headband is broken down into the discrete frequencies that it is composed of. (Clutterbuck, 2016)

By providing raw FFTs for each electrode, one can effectively view which frequencies are making up a signal from a particular electrode on the headband, as well as the percentage each

frequency contributes to making up that signal. Please refer to Appendix B for a sample table of the unabridged data as exported from MindMonitor.

6. Theoretical Framework

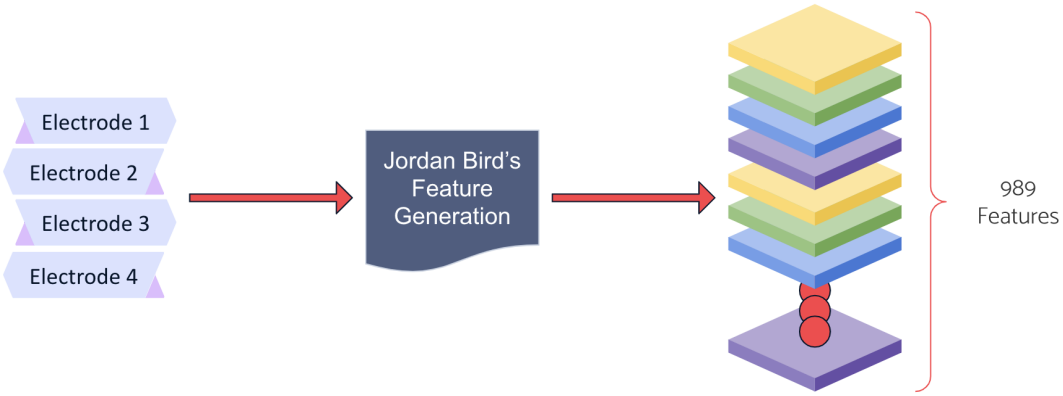
The following section will explore the current state of the literature on the field of neural oscillations and machine learning. Particular focus will be placed around the work of Jordan James Bird, a PhD. researcher at Nottingham Trent University. Bird's work was used substantially in this research; his approach to CNN models via converting neural oscillation data into images was of particular relevance to our work. Similarly, this section will explore all the theory revolving around neural oscillations, and the relevance to the present work.

6.1. State of Literature

The current world record for EEG signal classification models is held by Bird et al, with an accuracy of 89.38% set in 2020. Previously, this record was set at 87.20 and 79.80% by Bird et al in 2018 and 2019 respectively. These extremely high-accuracy models are due in part to ongoing research regarding the best features that ML models can leverage from EEG data. Moreover, the approach used in these different studies varies slightly, with the latest study using an image classification approach as will be expressed in following sections.

As part of his research, Bird published an open-source Python method which computes 989 features using only 4 electrode raw signals as input. These features make up the numerical and categorical data that machine learning models leverage in order to learn and make predictions. The 989 features arise from the work of previous researchers in the field, and have been shown to improve the accuracy of machine learning models when employed.

Figure 4: Feature generation process



6.2. Feature Extraction

This section will explain the significance of the 989 features generated by Bird's open-source software. The software is available publicly and draws from the work of Bird in 2018, 2019, and 2020.

As previously mentioned, the software uses the raw EEG readings as input, as well as the time when the measurements were recorded. The format that was used as input to the software was as follows (note that the rightmost column "AUX_RIGHT" is an unused sensor reading and is ignored in the feature extraction software):

Table 3: Data inputs to the feature generation software

	TimeStamp	RAW_TP9	RAW_AF7	RAW_AF8	RAW_TP10	AUX_RIGHT
0	1635812847.362	792.967033	811.904762	769.194139	788.937729	680.549451
1	1635812848.362	751.465201	770.805861	780.879121	803.846154	799.816850
2	1635812849.362	747.435897	827.619048	793.369963	802.234432	906.190476
3	1635812850.371	838.901099	803.040293	803.443223	795.787546	817.142857
4	1635812851.366	809.890110	780.476190	798.205128	743.406593	785.311355

It can be seen that the full time series of data is input into the software for a certain individual.

The following is the complete list of features generated by the software, extracted from Bird's 2020 study "*Classification of EEG Signals Based on Image Representation of Statistical Features*":

- Considering the full window:
    - The sample mean and sample standard deviation of each signal (8 features).
    - The sample skewness and sample kurtosis of each signal (8 features).
    - The maximum and minimum value of each signal (8 features).
    - The sample variances of each signal, plus the sample covariances of all signal pairs (10 features).
    - The eigenvalues of the covariance matrix (4 features).
    - The upper triangular elements of the matrix logarithm of the covariance matrix . (10 features)
    - The magnitude of the frequency components of each signal, obtained using a Fast Fourier Transform (FFT) (300 features).
    - The frequency values of the ten most energetic components of the FFT, for each signal (40 features).
- Considering the two half-windows:
    - The change in the sample means and in the sample standard deviations between the first and second half-windows, for all signals (8 features).
    - The change in the maximum and minimum values between the first and second half-windows, for all signals (8 features).
- Considering the quarter-windows:
    - The sample mean of each quarter-window, plus all paired differences of sample means between the quarter-windows, for all signals (56 features).
    - The maximum (minimum) values of each quarter-window, plus all paired differences of maximum (minimum) values between the quarter-windows, for all signals (112 features).

(Bird. et al, 2020)

7. Data Preprocessing

7.1. Initial Preprocessing

As described in Section 5.4, once data is recorded and saved from the MindMonitor app to a .CSV file, it is depicted in the form of raw FFTs for each electrode on the Muse 2. Every second of data recording generated 39 different features. The following figure shows some of these 39 features from a "Not Tired" data sample, which includes not only the Fourier-transformed data but also accelerometer and gyroscope data, timestamps, blinking and jaw clenching readings, as well as battery percentage and electrode connection quality:

Table 4: Raw EEG data

	TimeStamp	Delta_TP9	Delta_AF7	Delta_AF8	Delta_TP10	Theta_TP9	Theta_AF7	Theta_AF8	Theta_TP10	Alpha_TP9	...	Gyro_X	Gyro_Y	Gyro_Z	HeadBandOn	HSI_TP9	HSI_AF7	HSI_AF8	HSI_TP10	Battery	Elements
0	2021-11-01 17:54:38.045	1.110457	-0.382196	0.082630	0.743808	0.455723	-0.523256	0.086015	0.487615	0.493558	...	4.134674	-5.824432	-1.510315	1.0	1.0	2.0	1.0	1.0	70.0	NaN
1	2021-11-01 17:54:39.045	0.904642	-0.382196	0.236881	0.613098	0.313527	-0.523256	0.171247	0.546970	0.538756	...	4.329071	-2.990723	-1.644897	1.0	1.0	2.0	1.0	1.0	70.0	NaN
2	2021-11-01 17:54:39.187	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	/muse/elements/jaw_clench
3	2021-11-01 17:54:40.045	0.652124	-0.382196	0.462323	0.410327	0.293693	-0.523256	0.267178	0.466408	0.343593	...	5.622559	-5.099182	-0.732727	1.0	1.0	2.0	1.0	1.0	70.0	NaN
4	2021-11-01 17:54:41.043	0.558608	-0.382196	0.502156	0.877835	0.281408	-0.523256	0.337400	0.469669	0.381862	...	4.882355	-3.536530	-1.652374	1.0	1.0	2.0	1.0	1.0	70.0	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
156	2021-11-01 17:56:57.042	1.011459	-0.382196	0.502156	0.955036	0.456557	-0.523256	0.337400	0.439388	0.575279	...	5.510406	-7.880554	-2.257996	1.0	1.0	4.0	2.0	1.0	70.0	NaN
157	2021-11-01 17:56:57.111	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	/muse/elements/blink
158	2021-11-01 17:56:57.852	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	/muse/elements/blink
159	2021-11-01 17:56:58.042	1.011459	-0.382196	0.502156	0.955036	0.456557	-0.523256	0.337400	0.439388	0.575279	...	4.844971	-6.190796	-2.781372	1.0	1.0	4.0	4.0	1.0	70.0	NaN
160	2021-11-01 17:56:58.222	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	/muse/elements/blink

161 rows × 39 columns

While data of this form is useful for many neuroscience applications, 39 features is not enough for a machine learning algorithm to learn to differentiate between data as fluid and volatile as neural oscillations. As a result, a rigorous data cleaning process was performed in order to have the data be compatible as an input to Jordan B. Bird's feature generation Python method, and generate a larger number of features to be passed to the machine learning algorithm. (Refer Appendix A for full preprocessing details.)

Bird's method requires only timestamps and raw EEG data in the form of voltages read by each electrode. As a result, the first step was removing all Fourier-transformed data, as well as accelerometer, gyroscope, blinking, jaw-clenching, and battery information. A number of null EEG samples were also present due to period bluetooth connectivity issues or non-constant sampling rates

for different sensors, resulting in several empty rows of data which had to be removed. Furthermore, the Muse 2 took time measurements in the Datetime format which had to be converted to Timestamp format to be compatible with Bird's method.

Given that all the data was recorded in Vancouver, Canada, a notch filter around the 60 Hz frequency was applied. This ensures that noise from mains power is minimized. Specifically, Bird's feature generation software was modified in order to perform this filtering at 60 rather than at 50 Hz, as was the intention in the original version of the source code.

Upon completion of these data refinement steps, the preprocessed data was saved before being passed into Bird's feature generation method. The following figure shows the first 5 rows of the same "Not Tired" data sample, now processed to include only timestamps and raw voltage readings (in μV) from each of the electrodes on the Muse 2 headband:

Table 5: Data inputs to the feature generation software

	TimeStamp	RAW_TP9	RAW_AF7	RAW_AF8	RAW_TP10	AUX_RIGHT
0	1635812847.362	792.967033	811.904762	769.194139	788.937729	680.549451
1	1635812848.362	751.465201	770.805861	780.879121	803.846154	799.816850
2	1635812849.362	747.435897	827.619048	793.369963	802.234432	906.190476
3	1635812850.371	838.901099	803.040293	803.443223	795.787546	817.142857
4	1635812851.366	809.890110	780.476190	798.205128	743.406593	785.311355

7.2. Feature Generation

There are several parameters that can be set when using Bird's feature extraction software, the following table is a summary of these parameters, the values they were set to in this study, as well as their use:

Table 6: Summary of Feature Generation Software Parameters

Parameter	Description	Value
Filepath	Location of the input data to the program as .CSV	-

nsamp	Number of samples to resample for each time window on all signals	50
period	Duration of the time window in seconds	6
Slide_percent *	Amount to slide each window by as percentage of period.	1%
remove_redundant	Boolean, weather to remove redundant features from output	False
cols_to_ignore	List of Features to ignore or avoid computing	None

Note that the 'slide_percent' parameter was added in the present work, in order to maximize the amount of data generated by the software.

With the inputs and parameters now defined, the software produces 989 features and outputs them as a matrix for further analysis. In producing the features, the software divides the time window into half- and quarter-windows in order to calculate some of the features. Once the features are calculated for all windows, the time window *slides* to include new data on the right, and drop data on the left. This procedure is continued until the end of the file is reached, at which point there is only one full window remaining.

The following is an example of the data output from the software:

Table 7: Sample data output from feature generation software

TimeStamp	lag1_mean_0	lag1_mean_1	lag1_mean_2	lag1_mean_3	lag1_mean_4	lag1_mean_d_h2h1_0	lag1_mean_d_h2h1_1	lag1_mean_d_h2h1_2
788.131868	798.769231	789.018315	786.842491	797.802198	55.860118	-1.775760	12.502775	-30.339521
788.131868	798.769231	789.018315	786.842491	797.802198	55.860118	-1.775760	12.502775	-30.339521
788.131868	798.769231	789.018315	786.842491	797.802198	55.860118	-1.775760	12.502775	-30.339521
788.131868	798.769231	789.018315	786.842491	797.802198	55.860118	-1.775760	12.502775	-30.339521
782.571429	798.688645	794.336996	789.582418	827.377289	3.583346	-14.421597	0.365225	-17.210459

The size of the output matrix depends on both the input matrix size and the parameters used by the function. The number of features generated is always 989 as per the complete list shown in Section 6.2 above.

7.3. Further Processing Steps

Upon calling Bird's method, a coherency check was performed on each of the 'Tired' and 'Not Tired' datasets to ensure that both had successfully generated the 989 features as expected. The Timestamps were then removed as they were no longer necessary beyond the calling of the feature generation method. Labels were attached for each class ('Tired' = 1, 'Not Tired' = 0) before the datasets were combined and randomized to minimize bias. A Train and Test split of the combined dataset was then performed with the train set making up 80% of the data and the remainder forming the test set. The column of labels was then assigned as the output variable, with the input variable holding the remainder of the data. Feature scaling was performed on the input data so as to approximate the columnar data as if normally distributed with a mean of 0 and unity standard deviation. The image below shows a section of the train set following feature scaling:

Table 8: Train set sample after scaling

	0	1	2	3	4	5	6	7	8	9
0	0.152249	-0.261045	-0.258016	0.535385	0.263150	0.059903	-0.140409	-0.049241	0.915919	-0.365244
1	0.143130	-0.378017	-0.416003	-0.281708	-0.104084	-0.084495	0.016555	0.020376	0.450715	-0.171376
2	0.090031	-0.241925	-0.872286	1.557996	0.244509	0.253245	-0.088993	-0.203154	-1.164588	0.007921
3	0.134012	-0.338089	-0.163003	0.513839	0.006225	0.030942	0.035758	0.118816	-0.140858	-0.162809
4	-0.017241	-0.091212	-0.547474	-1.138579	-0.382105	-0.200236	0.255581	-0.844705	-1.992430	-0.334977

7.4. Principal Component Analysis

Following the general data processing, principal component analysis (PCA) was performed to reduce the number of features while retaining a majority of the information based on the information gain measure. 989 features was an excessive amount of features to pass to the machine learning programs, and by reducing this to 225 principal components approximately 100% of the information was still retained. The following figure shows the state of the dataset after performing PCA.

Table 9: Sample input data after performing PCA

```
Train set shape =  (270, 225)
Test set shape  =  (68, 225)
Using the first 225 Principal Components describes 100.0% of the data.
```

	0	1	2	3	4	5	6	7	8	9
0	-188799.714527	-23073.518325	-9801.856529	9566.113542	-14332.068735	-23187.043665	2995.226434	-1386.114505	-3396.073323	-5915.612791
1	-188799.762822	-23073.511101	-9801.836145	9566.094555	-14331.974606	-23187.555427	2995.211264	-1385.872033	-3396.202215	-5915.717482
2	-188799.538669	-23073.571970	-9801.963634	9566.308546	-14332.134694	-23186.135988	2995.084943	-1385.992396	-3395.312072	-5914.853783
3	-188799.763902	-23073.506342	-9801.835448	9566.090586	-14331.999563	-23187.476976	2995.212687	-1385.890715	-3396.137747	-5915.701104
4	-188799.012054	-23073.602369	-9802.169941	9566.541268	-14332.831484	-23180.039102	2995.548317	-1389.757445	-3396.123283	-5914.309797
...	...	...	...	...	...	...	...	...	...	...
333	-188789.845100	-23064.588811	-9804.542697	9569.568994	-14333.395024	-23188.534687	2995.778737	-1387.696793	-3392.594902	-5919.826044
334	-188799.688663	-23073.579169	-9801.823081	9566.098257	-14332.109641	-23186.989600	2995.149934	-1385.836490	-3395.773882	-5915.331444
335	-188799.024606	-23073.593786	-9802.170699	9566.550611	-14332.782422	-23180.225029	2995.545433	-1389.684965	-3396.304350	-5914.563509
336	-188799.701441	-23073.536908	-9801.860717	9566.113083	-14331.995001	-23187.420330	2995.217597	-1385.927799	-3396.125835	-5915.805956
337	-188798.872897	-23072.944347	-9801.793007	9565.975333	-14331.777909	-23187.911325	2995.117982	-1385.908342	-3395.828999	-5915.452494

338 rows × 225 columns

The number $15^2 = 225$ was chosen as this would allow us to generate square 15x15 pixel images during the image generation process (see Section 7.5). Prior to this, the principal components were scaled as per a normal distribution (μ=0, σ=1), and plotted to gain a visual understanding of the data. Below are plots comparing pairs of principal components with each other, clearly depicting the separation boundary between the 'Tired' and 'Not Tired' classes.

Figure 5: Comparing PCs #0 and #5

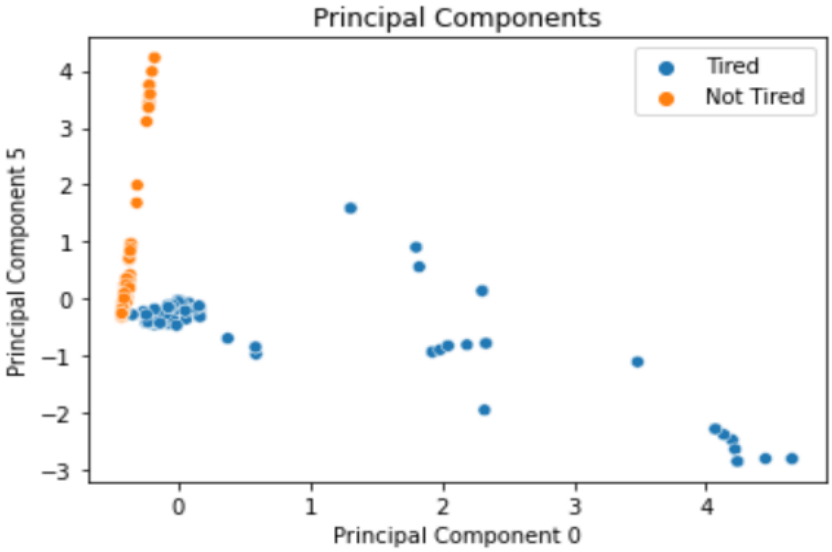
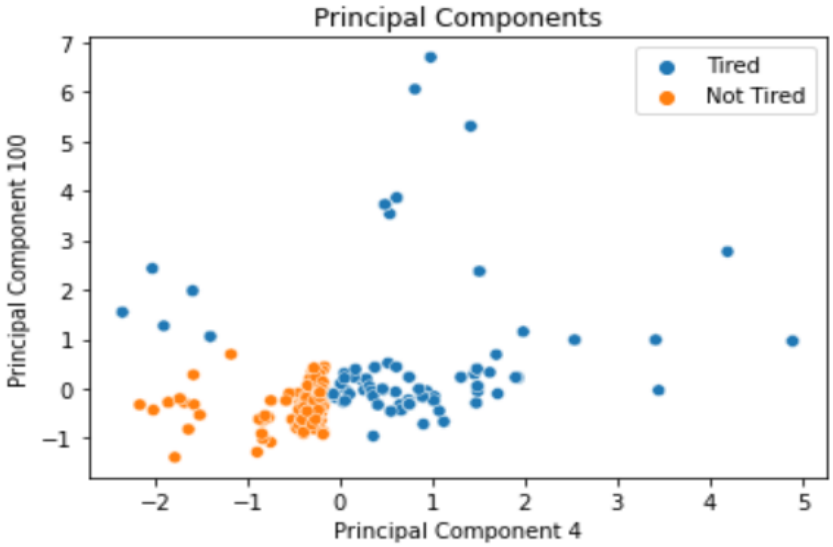


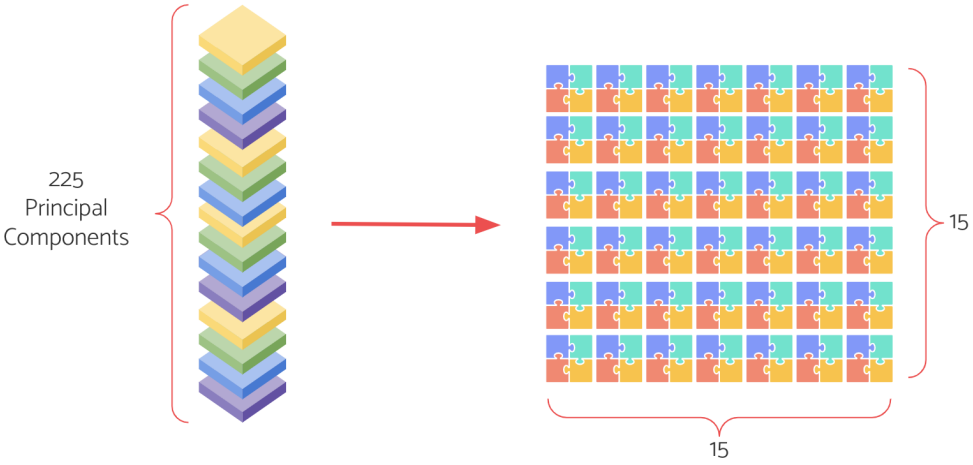
Figure 6: Comparing PCs #4 and #100



7.5. Image Generation

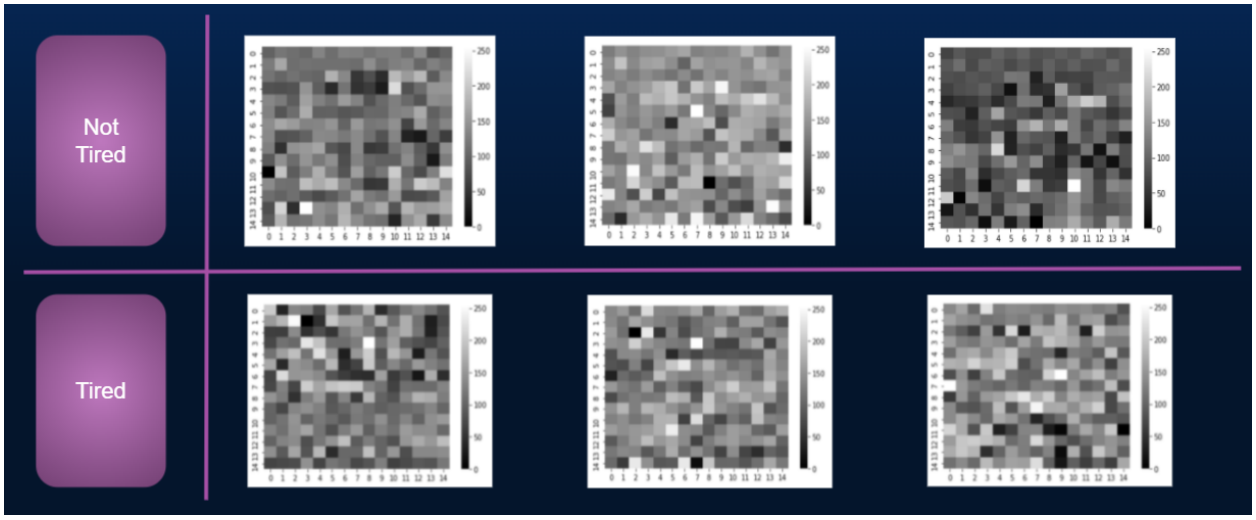
Grayscale data images were generated for each row of data in order to leverage the image processing capabilities of Convolutional Neural Networks (see Section 8.2). This approach of converting neural oscillation data to images for use in CNN models was drawn from the work of Jordan Bird, as mentioned in Section 6. The following figure shows the general procedure that was followed in generating the images:

Figure 7: General image generation process



Grayscale imaging was chosen as in this form of imaging the value of each pixel of an image represents the value of a single principal component. Below are some examples of these images for both the 'Tired' and 'Not Tired' classes.

Figure 8: Grayscale images representing each class



In order to generate these images, the data was scaled from 0-256 as 8-bit integer values (with 0 representing a black pixel and 255 representing a white pixel) and a series of images was generated using the NumPy Python method *reshape()*. The library *Imageio* was then used to convert and save these images as .PNG files as this format is compatible for use in keras's CNNs.

8. Machine Learning Models

This section will explain some of the fundamental concepts regarding the two principal machine learning (ML) models used in this study. While not a thorough explanation for each of the models, this section will help establish the foundation for developed models. Additionally, the developed models' architecture will be explored, as well as the hyperparameters that were used in this study.

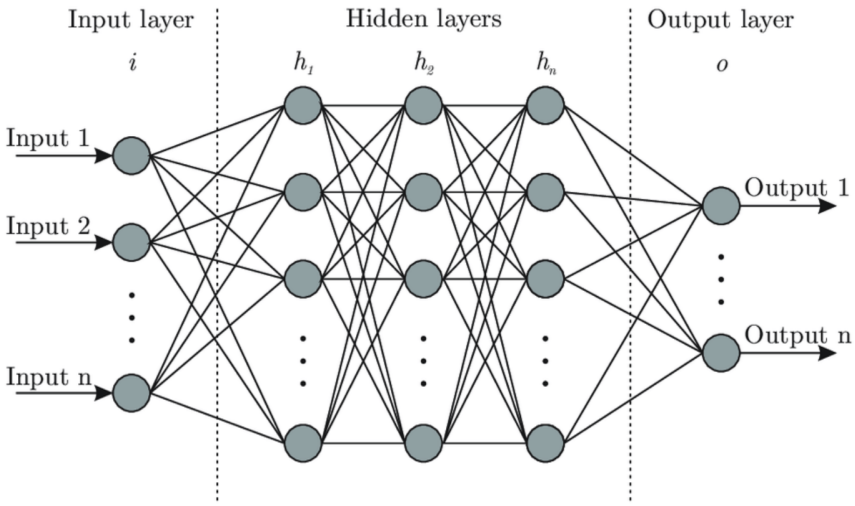
8.1. Artificial Neural Networks

8.1.1. Theory

Artificial Neural Networks are a class of machine learning models which are composed of virtual neurons –called perceptrons– and are modeled to resemble the animal brain. Much like in the biological brain, the perceptrons receive some input, process it, and if the result is above a certain threshold the signal is sent to another neuron. This process, formally called *forward propagation,* is what allows networks of perceptrons to convert inputs into outputs.

Typically, ANNs are composed of an input layer, one or more hidden layers, and a single output layer. All layers in the *middle* are called hidden due to the fact that their inputs and outputs are hidden from the end-user, and only visible to the layers immediately preceding and following them. The following figure illustrates the general architecture, or structure, of an ANN model.

Figure 9: Typical ANN model architecture (via: towardsdatascience.com)

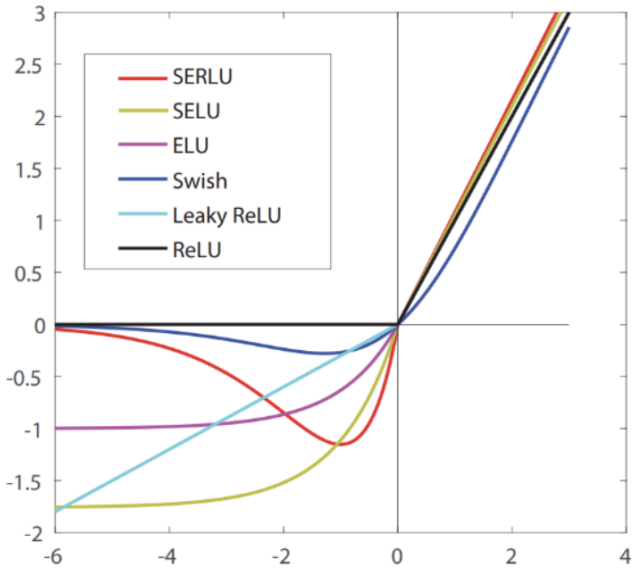


It is common to use densely connected layers throughout the model, where each neuron in a layer has an individual connection to every neuron in both the preceding and following layers. Alternative due exist, which may be more or less pertinent depending on the model architecture, data, and application.

Finally, it is important to mention that each neuron has its own activation function. These functions are the determining factor on whether a neuron will pass a significant output to the others.

These functions are nonlinear, and can be one of several forms. The following figure shows a few of the most popular activation functions currently in use:

Figure 10: Popular activation functions (via: developpaper.com)



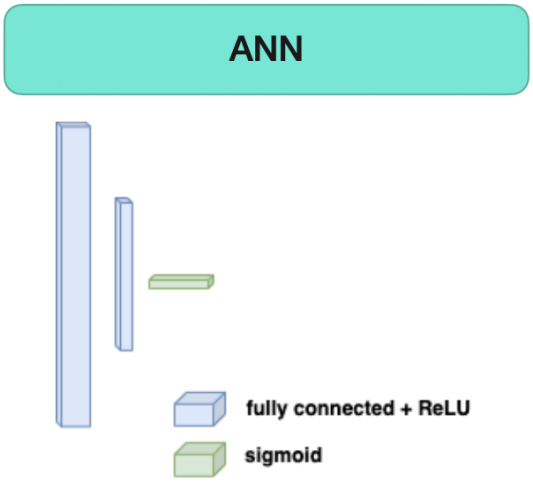
Both the ANN and CNN models developed in this study exclusively use the Sigmoid and Rectified Linear Unit (ReLu) activation functions, which can be defined mathematically as follows:

Sigmoid	ReLu
$S(x) = \frac{1}{1+e^{-x}}$	$f(x) = x^{+} = MAX(0, x)$

8.1.2. Architecture

The developed ANN model in the present study was composed of a single input layer with 225 neurons (one for each Principal Component), a hidden layer with 112 neurons, and an output layer with only one neuron. All the layers were densely connected. The input and output layers make use of the ReLu activation function, while the output layer utilizes the Sigmoid function. The following image describes the ANN model's architecture.

Figure 11: ANN model architecture



The following table offers a summary of the ANN model's architecture.

Table 10: Summary of ANN model architecture

ANN
Dense (225 neurons, ReLu)
Dense (122 neurons, ReLu)
Convolution2D (112 neurons, ReLu)

Finally, it is important to mention the different hyperparameters that were chosen for this model. It must be noted that due to issues with overfitting, these parameters were chosen but could not be further optimized (see Section 9 for further details). For the ANN model, the selected optimizer was 'adam' paired with a 'binary_crossentropy' loss function.

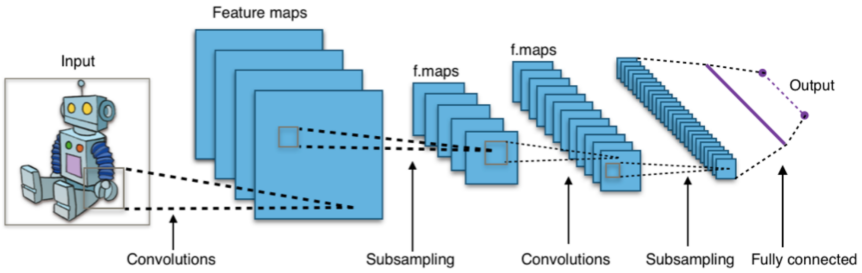
8.2. Convolutional Neural Networks

8.2.1. Theory

Convolutional Neural Networks (CNNs) are a subclass of ANNs. CNNs are similar to ANNs in that they implement several multilayered perceptrons and consist of input, hidden, and output layers. CNNs, however, have additional *steps* between the input and hidden layers which make them

ideal for specific tasks like image and video recognition. The main distinguishing factor between ANN and CNN is the fact that some of the CNN's hidden layers use convolution.

Figure 12: Typical CNN model architecture (via: Aphex34 at Wikimedia Commons)



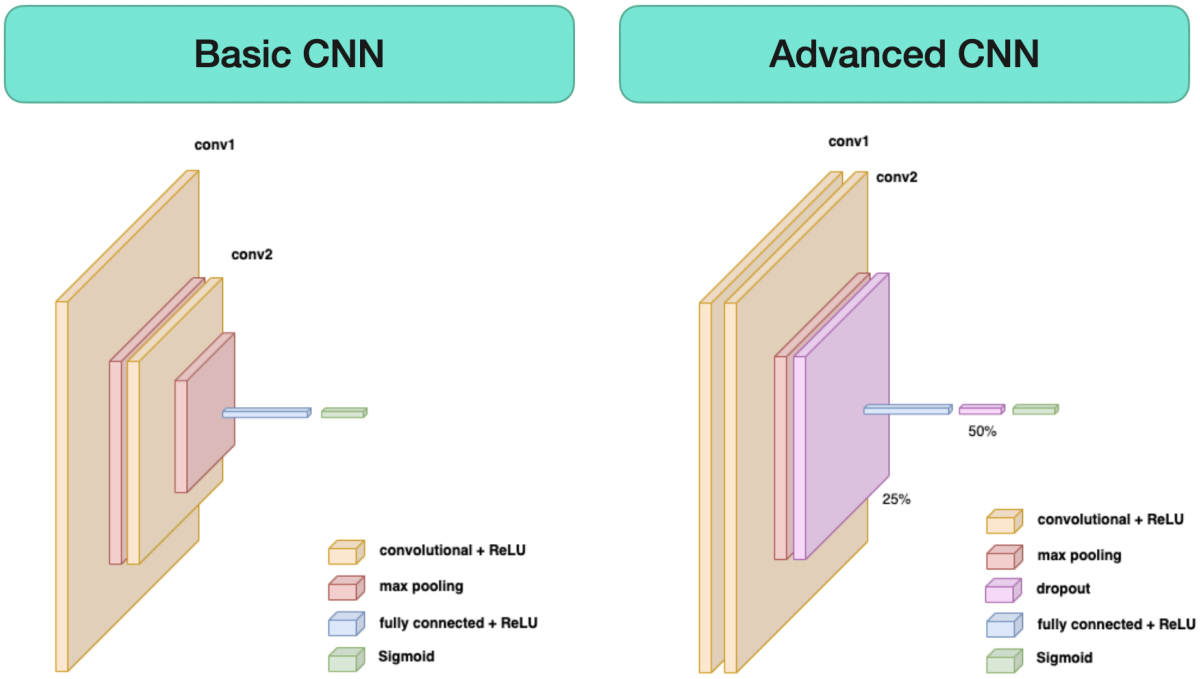
The convolutional layers offer two main advantages over ANN for image processing. Firstly, the amount of densely connected layers is reduced since the convolution has a single output per cluster of pixels, as opposed to an output for each pixel. Secondly, these layers allow for relational information to be taken into account. For example, patterns in neighboring pixels would play an effect in the convolutional layers even when *looking* at a particular pixel.

Pooling layers are typically also used in CNN models. These layers reduce the number of outputs after a given convolution by taking either the maximum or average (typically) value. These effects, arising from the convolution and pooling layers, are advantageous because they reduce the number of learnable parameters, or weights, for the model, resulting in lower training times and resource usage.

8.2.2. Architecture

Two CNN models were developed, trained, and tested for this study. The first is a simple model which uses only basic layers for a CNN model. The second is slightly more advanced, in that it includes successive convolutional layers and dropout layers. An important distinguishing factor between these models is the fact that the advanced CNN architecture is designed to prevent the common problem of overfitting. By including two dropout layers (see below), the model is less likely – but still susceptible – to overfit the data. The following figure details each model's architecture:

Figure 13: Basic and Advanced CNN model architectures



The different model architectures can be summarized as per the following table:

Table 11: CNN model architecture summary

Basic CNN	Advanced CNN
Convolution2D (225 neurons, ReLu)	Convolution2D (225 neurons, ReLu)
Max Pooling	Convolution2D (112 neurons, ReLu)
Convolution2D (112 neurons, ReLu)	Max Pooling
Max Pooling	Dropout (25%)
Flatten	Flatten
Dense (225 neurons, ReLu)	Dense (225 neurons, ReLu)
Dense (1 neuron, Sigmoid)	Dropout (50%)
	Dense (1 neuron, Sigmoid)

In a similar fashion to ANN, the chosen hyperparameters were not optimized due to problems with overfitting (see Section 9). These hyperparameters were: 'adam' as the optimizer and 'binary_crossentropy' as the loss function.

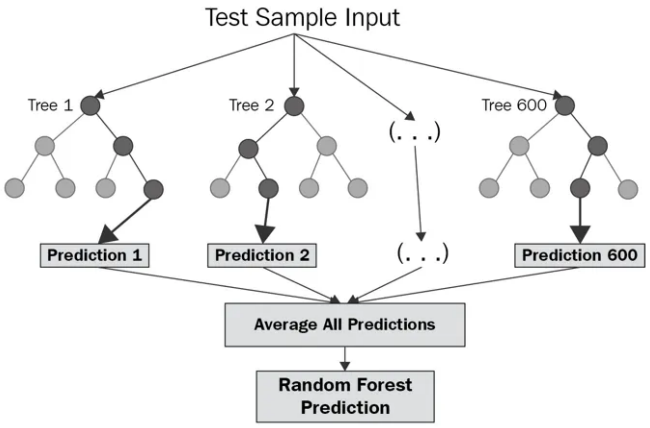
8.3. Random Forest

Random Forest is a classification algorithm which is made up of multiple decision trees acting as an ensemble. Data is fed into the algorithm and each tree spits out a prediction on whether the data is representative of the 'Tired' or 'Not Tired' class. The class with more outputs is chosen as the model prediction.

For this particular dataset, the number of trees was set to 100 when running the model. Data containing two distinct classes, such as 'Tired' and 'Not Tired' in this case, is where the RF model is very well suited as a prediction algorithm. Bagging and feature randomness are used so that the decision trees in the model are each independent of one another, and by acting and forming independent decisions the model is able to create a more accurate prediction than any one decision tree would on its own.

RF models are known to be extremely susceptible to data overfitting. While the model developed in this study was no exception, it certainly was not the only model to exhibit this shortcoming. The following figure details a general random forest structure:

Figure 13: General random forest ML model architecture (via: corporatefinanceinstitute.com)



8.4. Logistic Regression

Logistic Regression (LR) is a statistical model that is used to model the behaviour of binary variables. In LR, there is a *logistic* function which converts the logarithm of the odds of the variable (log-odds) being "1" using a linear combination of *predictors* as inputs. This model can be used with a combination of categorical and numerical data in order to regress the probability of class association; hence, LR can act as a classifier if the predicted (regressed) odds are above a certain threshold – typically above 50%. The present study utilized Newton's method as the numerical solver's algorithm as its only parameter.

8.5. Model Training

All the aforementioned models were trained according to the specific dataset they were designed for. Specifically, the ANN models were trained on the Principal Component dataset (225 features), and the CNN models were trained on the generated images (15x15 px). Random Forest and Logistic Regression models were also trained on the PC dataset.

In order to optimize the training process, the callback functionality of keras's and scikit-learn's model.fit() functions was used. This allowed the training process to stop automatically as soon as the model had stopped learning. This time and resource saving functionality was implemented using keras's (keras.callbacks) EarlyStopping() function. This function was set up to stop the training once the validation accuracy – the accuracy of the model when tested on the test set – had stopped increasing for 10 epochs. Additionally, the function restores the model weights for the last step of the epoch with the best validation accuracy.

9. Results

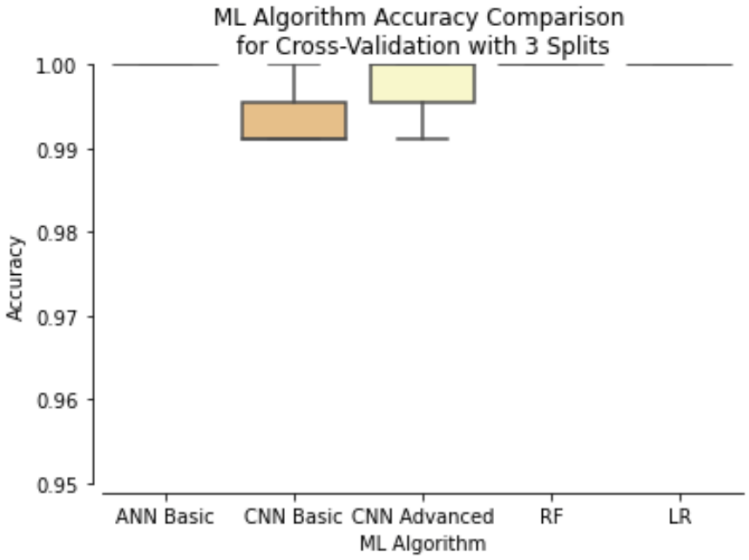
Once the models are trained, they can be used to predict whether a specific sample of neural oscillations belongs to the fatigued or not fatigued class. Evaluation of the models was performed via K-Fold Cross Validation, which allows the models to be evaluated on different train/test splits for the most accurate description of model performance.

K-Fold Cross Validation is a statistical procedure which allows for unbiased evaluation of a ML model. It consists of a single parameter K, which determines the number of *splits* to be made on

the dataset. Once the splits are made, a random split will be used as a test set, while the others are used to train the models. Once trained, the model is evaluated on the test set, and its score (typically accuracy) is stored. The final evaluation metric is the mean accuracy of the model on all the K splits.

In this study, K was fixed to 3 due to a low dataset size. Choosing a larger value of K would have resulted in the testing set no longer being representative of the entire dataset. After performing 3-Fold Cross Validation using the Scikit-Learn library, the following results were achieved:

Figure 14: 3-Fold cross validation results summary



It is important to notice the y-axis limits of Figure 14, which range from 95% to 100% accuracy. While these results seem to indicate near 100% accuracy across all models, it would be fallacious to interpret these as a true measure of the model's skill, due to overfitting.

Given the small data collection time window, and the resulting small-sized dataset, overfitting is observed across all models. The lack of general variability, randomness, and asymmetry in data collection is reflected in the fact that it is very *easy* for the models to identify the boundaries that separate fatigued and not fatigued individuals. Had the dataset been larger, and the data collection taken over a longer time period and with more participating individuals in both states, the problem of overfitting would become less dominant, and a truer evaluation of model skill could be performed.

10. Conclusions

With the availability of sophisticated technology such as the Muse 2, it is valuable to understand how brainwave data is impacted by the level of fatigue experienced by a person. Our study aimed to determine whether a machine learning model can accurately predict if a person is fatigued or not simply by analyzing their brainwaves. After collecting data for both fatigued and not fatigued individuals, randomizing and scaling the data and training five different machine learning models on the dataset, our study concluded that all of our models (Artificial Neural Network, two different Convolutional Neural Networks, Random Forest and Logistic Regression) were able to predict fatigued or not with an accuracy of 100% or very close to it. Although our results show our study to be surprisingly promising, it is important to note machine learning models tend to overfit smaller amounts of data inputs resulting in accuracies of ~100%, as is the case in our study. Due to this reason, it is difficult to compare the accuracy of this study to previous work in the field.

Future research into the correlation between brain activity and fatigue state should aim to apply more data, collected over the span of many months on several individuals. The Muse 2 headband offers a variety of other raw data, such as blinking and jaw clench detections, along with motion readings collected from a built-in gyroscope and accelerometer. In future iterations of this study, these additional data points can be included in developing a stronger distinction between fatigued and not fatigued states. Furthermore, while our research focused on fatigue resulting from sleep deprivation, this definition can be expanded to include a wide range of both physical and mental fatigue sources in order to expand the applications of the study, and the scalability of the model.

11. References

Ashford J., Bird J.J., Campelo F., Faria D.R. (2020) Classification of EEG Signals Based on Image Representation of Statistical Features. In: Ju Z., Yang L., Yang C., Gegov A., Zhou D. (eds) Advances in Computational Intelligence Systems. UKCI 2019. Advances in Intelligent Systems and Computing, vol 1043. Springer, Cham. https://doi.org/10.1007/978-3-030-29933-0_37

Available data. Muse Developers. (2018, November). Retrieved December 1, 2021, from https://web.archive.org/web/20181105231756/http://developer.choosemuse.com/tools/available-data#Absolute_Band_Powers.

Bird, J. J., Manso, L. J., Ribeiro, E. P., Ekárt, A., & Faria, D. R. (2018, September). A study on mental state classification using eeg-based brain-machine interface. In 2018 International Conference on Intelligent Systems (IS) (pp. 795-800). IEEE.

Bird, J. J., Faria, D. R., Manso, L. J., Ekárt, A., & Buckingham, C. D. (2019). A deep evolutionary approach to bioinspired classifier optimisation for brain-machine interaction. Complexity, 2019.

Bird, J. J. (2019) [Source Code] EEG Feature Generation. Retrieved November 1, 2021. Retrieved from https://github.com/jordan-bird/eeg-feature-generation

Clutterbuck, J. (n.d.). *Technical Manual*. Mind monitor. Retrieved December 1, 2021, from https://mind-monitor.com/Technical_Manual.php.

Clutterbuck, J. (2016) *MindMonitor* (Version 2.2) [iPhone mobile application software]. Retrieved from https://apps.apple.com/us/app/mind-monitor/id988527143

National Safety Council. (2021). *NSC Fatigue Reports*. Retrieved December 1, 2021, from https://www.nsc.org/workplace/safety-topics/fatigue/fatigue-reports.

Tefft, B. C. (2014). Prevalence of motor vehicle crashes involving drowsy drivers, United States, 2009-2013. *Accident Analysis & Prevention*, *45*. https://doi.org/10.1016/j.aap.2011.05.028

Wikipedia contributors. (2021, November 26). Electroencephalography. In Wikipedia, The Free Encyclopedia. Retrieved 08:00, December 1, 2021, from https://en.wikipedia.org/w/index.php?title=Electroencephalography&oldid=1057304778

12. Appendix A – Code

The following is the code developed and used throughout the project.

(*Page intentionally left blank, code commences on the following page.*)

Project Description

The following is the code used in the making of a MANU465 Capstone Project for Group 1. The entirety of the code and the data used throughout this project is available at this github repository.

Authors

Name	Student ID
Anant Goyal	46894325
Alberto Mussali	50684182
Musa Habib	25899808
Sadul Bombuwala	76343292

Overview

Motivation

Possibly the most incredible aspect of Machine Learning and Artificial Intelligence is the ability to mimic human-like decision making. Upon learning about how Artificial Neural Networks are constructed and how they work, we were incredibly intrigued by the workings of the neural networks within our brains. We see this project as a means to further study and understand the complexities of our minds. Collecting and analyzing brainwave data is something we have never had the opportunity to do, and we hope that during the course of this project we will gain a valuable understanding of how brainwave data within humans can be used for research in the world of Artificial Intelligence.

Goals and Objective

Objective: To be able to use machine learning models to predict whether or not a person is fatigued, based on brainwave data.

The Muse 2 is a multi-sensor meditation device that provides real-time feedback on brain activity, heart rate, breathing, and body movements to help users build a consistent meditation practice. When paired with the Mind Monitor phone application, one can view and analyze the neural oscillation readings picked up by the Muse 2, and use this headband for purposes beyond mediation.

> https://choosemuse.com/muse-2/
> https://mind-monitor.com/

Our project aims to use these tools to collect data on the variations in brain activity when an individual is in a Sleep-deprived state of mind, and compare this to when they are well rested. Using this data we plan to build a Machine Learning algorithm which accurately predicts the level of sleep deprivation of an individual, based on the brainwave data passed into the algorithm.

Challenges

It should be noted that due to the small amount of data compiled, all the trained models are observed to overfit the dataset. This can be remedied in the future by taking more data over the span of a longer timeframe (approximately 1 year) for multiple subjects on both states.

General Setup

Fix Random State

In [1]:
```
SEED = 55;
```

Importing Libraries

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import tensorflow as tf
import tensorflow.keras as kr
import seaborn as sns
import os
```

Importing the Raw Data

In [3]:
```
%%time


# Get Current Working directory and append the data relative dir
cwd = os.getcwd()
notTiredDir = cwd + r"\Data\Raw\NotTired"
tiredDir = cwd + r"\Data\Raw\Tired"

# Hold file locations
filesTired=[];
filesNotTired=[];

#Populate file location arrays
for file in os.listdir(notTiredDir):
    if file.endswith('.csv'):
        filesNotTired.append(os.path.join(notTiredDir, file))
for file in os.listdir(tiredDir):
        if file.endswith('.csv'):
            filesTired.append(os.path.join(tiredDir, file))

#Test reading files by changing num
num=6;
sample = pd.read_csv(filesNotTired[num])
sample
```

```
Wall time: 16 ms
```

Out[3]:

	TimeStamp	Delta_TP9	Delta_AF7	Delta_AF8	Delta_TP10	Theta_TP9	Theta_AF7	Theta_AF8	Theta_

	TimeStamp	Delta_TP9	Delta_AF7	Delta_AF8	Delta_TP10	Theta_TP9	Theta_AF7	Theta_AF8	Theta_
0	2021-11-01 17:54:38.045	1.110457	-0.382196	0.082630	0.743808	0.455723	-0.523256	0.086015	0.4
1	2021-11-01 17:54:39.045	0.904642	-0.382196	0.236881	0.613098	0.313527	-0.523256	0.171247	0.5
2	2021-11-01 17:54:39.187	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
3	2021-11-01 17:54:40.045	0.652124	-0.382196	0.462323	0.410327	0.293693	-0.523256	0.267178	0.46
4	2021-11-01 17:54:41.043	0.558608	-0.382196	0.502156	0.877835	0.281408	-0.523256	0.337400	0.46
...	...	...	...	...	...	...	...	...	
156	2021-11-01 17:56:57.042	1.011459	-0.382196	0.502156	0.955036	0.456557	-0.523256	0.337400	0.43
157	2021-11-01 17:56:57.111	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
158	2021-11-01 17:56:57.852	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
159	2021-11-01 17:56:58.042	1.011459	-0.382196	0.502156	0.955036	0.456557	-0.523256	0.337400	0.43
160	2021-11-01 17:56:58.222	NaN	NaN	NaN	NaN	NaN	NaN	NaN	

161 rows × 39 columns

In [4]:
```
#Mini-Summary of Block
print(f"> {len(filesNotTired)} files were added from the NOT TIRED category")
print(f"> {len(filesTired)} files were added from the TIRED category\n")
```

```
> 31 files were added from the NOT TIRED category
> 20 files were added from the TIRED category
```

Available Features

In [5]:
```
print("Features generated by the Muse 2 headband:")

pd.DataFrame(sample.columns).T
```

```
Features generated by the Muse 2 headband:
```

Out[5]:

	0	1	2	3	4	5	6	7	8
0	TimeStamp	Delta_TP9	Delta_AF7	Delta_AF8	Delta_TP10	Theta_TP9	Theta_AF7	Theta_AF8	Theta_TP10	Al

1 rows × 39 columns

Raw Data Structure

In [6]:
```
#quick view of data
```

```
sample.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 161 entries, 0 to 160
Data columns (total 39 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   TimeStamp         161 non-null    object 
 1   Delta_TP9         141 non-null    float64
 2   Delta_AF7         141 non-null    float64
 3   Delta_AF8         141 non-null    float64
 4   Delta_TP10        141 non-null    float64
 5   Theta_TP9         141 non-null    float64
 6   Theta_AF7         141 non-null    float64
 7   Theta_AF8         141 non-null    float64
 8   Theta_TP10        141 non-null    float64
 9   Alpha_TP9         141 non-null    float64
 10  Alpha_AF7         141 non-null    float64
 11  Alpha_AF8         141 non-null    float64
 12  Alpha_TP10        141 non-null    float64
 13  Beta_TP9          141 non-null    float64
 14  Beta_AF7          141 non-null    float64
 15  Beta_AF8          141 non-null    float64
 16  Beta_TP10         141 non-null    float64
 17  Gamma_TP9         141 non-null    float64
 18  Gamma_AF7         141 non-null    float64
 19  Gamma_AF8         141 non-null    float64
 20  Gamma_TP10        141 non-null    float64
 21  RAW_TP9           141 non-null    float64
 22  RAW_AF7           141 non-null    float64
 23  RAW_AF8           141 non-null    float64
 24  RAW_TP10          141 non-null    float64
 25  AUX_RIGHT         141 non-null    float64
 26  Accelerometer_X   141 non-null    float64
 27  Accelerometer_Y   141 non-null    float64
 28  Accelerometer_Z   141 non-null    float64
 29  Gyro_X            141 non-null    float64
 30  Gyro_Y            141 non-null    float64
 31  Gyro_Z            141 non-null    float64
 32  HeadBandOn        141 non-null    float64
 33  HSI_TP9           141 non-null    float64
 34  HSI_AF7           141 non-null    float64
 35  HSI_AF8           141 non-null    float64
 36  HSI_TP10          141 non-null    float64
 37  Battery           141 non-null    float64
 38  Elements          20 non-null     object 
dtypes: float64(37), object(2)
memory usage: 49.2+ KB
```

Data Preprocessing

Note: Initial preprocessing here was done to tailor the data so it could be passed to Jordan Bird's function 'EEG_feature_extraction' (https://github.com/AlbertoMussali-UBC/MANU465_Team1_Project)

Creating the RAW Dataset

In [7]:

```
%%time
## Extract rows 21-25 from all files,
## these are the only 5 features relevant for use in the EEG_feature_extraction function.
```

```
rowsTired=[];
for f in filesTired:
    for r in range(pd.read_csv(f).shape[0]):
        rowsTired.append(pd.read_csv(f).iloc[r,[0, 21,22,23,24,25]])

rowsNotTired=[];
for f in filesNotTired:
    for r in range(pd.read_csv(f).shape[0]):
        rowsNotTired.append(pd.read_csv(f).iloc[r,[0, 21,22,23,24,25]])
```

```
Wall time: 15min 5s
```

In [8]:

```
#Convert to DataFrames:

data_NT = pd.DataFrame(rowsNotTired);
original_NT = data_NT.copy();
data_NT
```

Out[8]:

	TimeStamp	RAW_TP9	RAW_AF7	RAW_AF8	RAW_TP10	AUX_RIGHT
0	2021-11-01 17:27:27.362	792.967033	811.904762	769.194139	788.937729	680.549451
1	2021-11-01 17:27:28.362	751.465201	770.805861	780.879121	803.846154	799.816850
2	2021-11-01 17:27:29.362	747.435897	827.619048	793.369963	802.234432	906.190476
3	2021-11-01 17:27:30.371	838.901099	803.040293	803.443223	795.787546	817.142857
4	2021-11-01 17:27:31.366	809.890110	780.476190	798.205128	743.406593	785.311355
...	...	...	...	...	...	...
2888	2021-11-27 19:42:02.022	800.219780	804.652015	0.000000	778.864469	1116.923077
2889	2021-11-27 19:42:03.017	799.816850	803.040293	1246.666667	751.868132	1625.018315
2890	2021-11-27 19:42:04.017	759.120879	799.010989	116.446886	828.424908	796.593407
2891	2021-11-27 19:42:05.016	780.879121	815.934066	0.000000	788.937729	628.974359
2892	2021-11-27 19:42:06.016	843.333333	795.384615	0.000000	818.754579	666.849817

31213 rows × 6 columns

In [9]:

```
data_T = pd.DataFrame(rowsTired);
original_T = data_T.copy();
data_T
```

Out[9]:

	TimeStamp	RAW_TP9	RAW_AF7	RAW_AF8	RAW_TP10	AUX_RIGHT
0	2021-11-11 00:54:16.918	782.893773	767.985348	1100.805861	799.413919	732.930403
1	2021-11-11 00:54:17.318	NaN	NaN	NaN	NaN	NaN
2	2021-11-11 00:54:17.920	790.549451	887.655678	1648.791209	809.487179	734.139194
3	2021-11-11 00:54:18.686	NaN	NaN	NaN	NaN	NaN
4	2021-11-11 00:54:18.929	755.494505	0.000000	0.000000	732.124542	565.714286
...	...	...	...	...	...	...
1295	2021-11-27 02:35:30.353	NaN	NaN	NaN	NaN	NaN

	TimeStamp	RAW_TP9	RAW_AF7	RAW_AF8	RAW_TP10	AUX_RIGHT
1296	2021-11-27 02:35:30.710	801.025641	812.307692	0.000000	802.637363	608.827839
1297	2021-11-27 02:35:30.977	NaN	NaN	NaN	NaN	NaN
1298	2021-11-27 02:35:31.711	808.681319	367.069597	46.336996	785.311355	652.344322
1299	2021-11-27 02:35:32.711	789.743590	280.439560	207.106227	822.783883	838.095238

26394 rows × 6 columns

In [10]:
```
#quick check of DataFrames

print(f"Not Tired Data size is: \t{data_NT.shape}", f"\nTired Data size is: \t\t{data_T.sh
```

```
Not Tired Data size is:         (31213, 6)
Tired Data size is:             (26394, 6)
```

Remove Empty Rows

Remove NaN values associated with blinking and jaw clenching

In [11]:
```
data_T = data_T.dropna()
```

In [12]:
```
data_NT = data_NT.dropna()
```

Convert Datetime Column to Timestamps

Required for compatibility with EEG_feature_extraction function

In [13]:
```
from datetime import datetime

ind = 0;
for time in data_T.iloc[:, 0]:
    tmstmp = datetime.strptime(str(time), '%Y-%m-%d %H:%M:%S.%f').timestamp()
    data_T.iat[ind, 0] = (tmstmp);
    ind=ind+1;

ind = 0;
for time in data_NT.iloc[:, 0]:
    tmstmp = datetime.strptime(str(time), '%Y-%m-%d %H:%M:%S.%f').timestamp()
    data_NT.iat[ind, 0] = (tmstmp);
    ind=ind+1;
```

In [14]:
```
#quick check
data_NT.head()
```

Out[14]:

	TimeStamp	RAW_TP9	RAW_AF7	RAW_AF8	RAW_TP10	AUX_RIGHT
0	1635812847.362	792.967033	811.904762	769.194139	788.937729	680.549451
1	1635812848.362	751.465201	770.805861	780.879121	803.846154	799.816850
2	1635812849.362	747.435897	827.619048	793.369963	802.234432	906.190476

	TimeStamp	RAW_TP9	RAW_AF7	RAW_AF8	RAW_TP10	AUX_RIGHT
3	1635812850.371	838.901099	803.040293	803.443223	795.787546	817.142857
4	1635812851.366	809.890110	780.476190	798.205128	743.406593	785.311355

Save Data to File

Alternative STARTING POINT once data collection is finalized

Note: this step was done to skip Section 3.1 which would take very long to run each time

In [15]:
```
savelocT = cwd + r"\Data\Preprocessed\Tired.csv"
savelocNT = cwd + r"\Data\Preprocessed\NotTired.csv"

if os.path.exists(savelocT):
    os.remove(savelocT)

if os.path.exists(savelocNT):
    os.remove(savelocNT)

data_T.to_csv(savelocT,  mode='w', index = False)
data_NT.to_csv(savelocNT,mode='w', index = False)
```

EEG Feature Generation

Execution of the function

In [16]:
```
from eegFG import EEG_feature_extraction as FG

#tried various combinations of Nsamp and Perio
#This combination was optimal
Nsamp = 50;
Perio = 6;

xT, yT = FG.generate_feature_vectors_from_samples(file_path=savelocT,
                                        nsamples=Nsamp,
                                        period=Perio,
                                        #state=data_NT.iloc[:,-1],
                                        slide_percent=0.05,
                                        remove_redundant=False,
                                        cols_to_ignore=None)
xT.shape
```

Out[16]:
```
(119, 810)
```

In [17]:
```
Nsamp = 50;
Perio = 5;

xNT, yNT = FG.generate_feature_vectors_from_samples(file_path=savelocNT,
                                        nsamples=Nsamp,
                                        period=Perio,
                                        #state=data_NT.iloc[:,-1],
                                        slide_percent=0.06,
                                        remove_redundant=False,
```

```
                                        cols_to_ignore=None)
xNT.shape
```

Out[17]: (219, 810)

The following code was used to optimize feature generation in Jordan Bird's method and can be ignored for now

```
%%time

from importlib import reload

flaggity=False

tmp_results=[]
thresh = 95;
for ns in range(50,256,1):
    if (flaggity==True):
        break;
    for p in range(3,8):

        try:
            reload(FG);
            xT, yT =
FG.generate_feature_vectors_from_samples(file_path=savelocT,
                                    nsamples=ns,
                                    period=p,
                                    #state=data_NT.iloc[:,-1],
                                    slide_percent=0.01,
                                    remove_redundant=False,
                                    cols_to_ignore=None)

            xNT, yNT =
FG.generate_feature_vectors_from_samples(file_path=savelocNT,
                                    nsamples=ns,
                                    period=p,
                                    #state=data_NT.iloc[:,-1],
                                    slide_percent=0.01,
                                    remove_redundant=False,
                                    cols_to_ignore=None)

        except (UnboundLocalError):
            continue;


        if (xNT.shape[1] == xT.shape[1]):
            print('Cols match!', xT.shape, xNT.shape)
            if (xNT.shape[0] >= thresh and xT.shape[0] >= thresh):
                print('Thresh met.')

tmp_results.append((ns,p,xNT.shape[0],xT.shape[0],xNT.shape[1]))
                flaggity=True;
                break;
```

```
tmp_results
```

In [18]:
```
#some quick checks

X_NT = pd.DataFrame(np.real(xNT))
X_NT.columns = np.hstack((['TimeStamp'], yNT))
X_NT.describe()
```

Out[18]:

	TimeStamp	lag1_mean_0	lag1_mean_1	lag1_mean_2	lag1_mean_3	lag1_mean_4	lag1_mean_d_h2h
count	219.000000	219.000000	219.000000	219.000000	219.000000	219.000000	219.000(
mean	779.105307	790.250350	793.258774	774.799042	826.072764	0.795194	-0.121
std	13.221727	13.962959	10.783482	15.090785	48.215973	29.464956	27.057
min	742.197802	753.641026	778.260073	731.721612	730.996337	-88.384188	-79.917(
25%	773.142857	786.197802	785.391941	770.543956	795.304029	-20.452674	-10.171
50%	780.717949	793.531136	791.194139	776.608059	825.362637	0.764957	3.152
75%	790.065934	797.560440	797.399267	784.102564	849.296703	18.143992	11.409
max	797.479853	813.838828	826.329670	816.578755	970.981685	81.924841	93.667

8 rows × 810 columns

In [19]:
```
with pd.option_context('display.max_rows', None, 'display.max_columns', None):  # more opt
    display(pd.DataFrame(pd.DataFrame(X_NT).head()))
```

	TimeStamp	lag1_mean_0	lag1_mean_1	lag1_mean_2	lag1_mean_3	lag1_mean_4	lag1_mean_d_h2h1_0
0	788.131868	798.769231	789.018315	786.842491	797.802198	55.860118	-1.775760
1	788.131868	798.769231	789.018315	786.842491	797.802198	55.860118	-1.775760
2	788.131868	798.769231	789.018315	786.842491	797.802198	55.860118	-1.775760
3	788.131868	798.769231	789.018315	786.842491	797.802198	55.860118	-1.775760
4	782.571429	798.688645	794.336996	789.582418	827.377289	3.583346	-14.421597

In [20]:
```
X_T = pd.DataFrame(np.real(xT))
X_T.columns = np.hstack((['TimeStamp'], yT))
X_T.describe()
```

Out[20]:

	TimeStamp	lag1_mean_0	lag1_mean_1	lag1_mean_2	lag1_mean_3	lag1_mean_4	lag1_mean_d_h2h
count	119.000000	119.000000	119.000000	119.000000	119.000000	119.000000	119.000
mean	853.022850	742.893619	911.472025	844.334452	784.668021	1.439227	-0.123
std	118.126749	251.525994	217.512393	104.498981	33.160809	254.559496	381.727
min	732.796093	350.818071	532.069597	664.499389	705.665446	-628.239954	-655.880

	TimeStamp	lag1_mean_0	lag1_mean_1	lag1_mean_2	lag1_mean_3	lag1_mean_4	lag1_mean_d_h2h
25%	765.869963	558.797314	777.084860	784.841270	770.570818	-34.380751	-320.33
50%	805.189255	626.153846	882.954823	803.308913	794.645910	-0.355619	88.870
75%	934.731380	967.301587	950.042735	876.507937	810.024420	30.105383	252.124
max	1152.985348	1349.884005	1541.947497	1150.097680	823.656899	717.580021	901.701

8 rows × 810 columns

In [21]:
```
# Drop TimeStamps as they are not needed anymore

X_T=X_T.iloc[:,1:];
X_NT=X_NT.iloc[:,1:];
```

Attach Labels for Each Class

In [22]:
```
#Stack ones or zeros for each class [0 = NotTired, 1 = Tired]
X_T = pd.DataFrame(np.hstack((X_T.to_numpy(),   np.ones((X_T.shape[0], 1)))))
X_NT= pd.DataFrame(np.hstack((X_NT.to_numpy(), np.zeros((X_NT.shape[0], 1)))))
```

In [23]:
```
#Add label heading
X_T.columns  = np.hstack((yT, ['Target']))
X_NT.columns = np.hstack((yNT, ['Target']))
```

Check Column Coherency

In [24]:
```
#Ensure Data is Coherent (same number of columns for X_T and X_NT)

print(X_T.shape[1], X_NT.shape[1])

if (X_T.shape[1] == X_NT.shape[1]):
    dataset = np.vstack((X_T, X_NT))
    dataset = pd.DataFrame(dataset)
    print('Columns are coherent')
else:
    print('NOT COHERENT')
```

```
810 810
Columns are coherent
```

Randomize the Dataset

In [25]:
```
dataset.columns = np.hstack((yT, ['Target']))
dataset = dataset.sample(frac = 1).reset_index(drop=True)
dataset.head()
```

Out[25]:

	lag1_mean_0	lag1_mean_1	lag1_mean_2	lag1_mean_3	lag1_mean_4	lag1_mean_d_h2h1_0	lag1_mean_d
0	796.109890	798.285714	782.410256	838.014652	46.848403	12.418692	-2

	lag1_mean_0	lag1_mean_1	lag1_mean_2	lag1_mean_3	lag1_mean_4	lag1_mean_d_h2h1_0	lag1_mean_d
1	794.739927	781.523810	770.886447	798.285714	-11.325690	-20.580290	
2	786.761905	801.025641	737.604396	887.736264	43.895367	56.602530	-1
3	793.369963	787.245421	789.340659	836.967033	6.148547	5.800402	4
4	770.644689	822.622711	761.296703	756.622711	-55.367517	-47.030048	39

5 rows × 810 columns

Separating Input and Output

In [26]:
```
x = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values

#Y labeled for plotting or result check
def label(n):
    if (n==0):
        return 'Not Tired'
    return 'Tired'

y_labeled = list(map(label, y));
```

Splitting Dataset into the Training and Test Sets

In [27]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, random_state =
```

Feature Scaling

Note for future: see if this step is not needed as we rescale the PCs anyway

In [28]:
```
from sklearn.preprocessing import StandardScaler
scX = StandardScaler();
scX.fit(X_train); #Fit to training data only
x = scX.transform(x)
```

In [29]:
```
#quick view of scaled data
pd.DataFrame(x)
```

Out[29]:

	0	1	2	3	4	5	6	7	8
0	0.152249	-0.261045	-0.258016	0.535385	0.263150	0.059903	-0.140409	-0.049241	0.915919
1	0.143130	-0.378017	-0.416003	-0.281708	-0.104084	-0.084495	0.016555	0.020376	0.450715
2	0.090031	-0.241925	-0.872286	1.557996	0.244509	0.253245	-0.088993	-0.203154	-1.164588
3	0.134012	-0.338089	-0.163003	0.513839	0.006225	0.030942	0.035758	0.118816	-0.140858

	0	1	2	3	4	5	6	7	8
4	-0.017241	-0.091212	-0.547474	-1.138579	-0.382105	-0.200236	0.255581	-0.844705	-1.992430
...	...	...	...	...	...	...	...	...	...
333	2.036304	3.039935	-0.201671	-0.544405	0.664601	-2.356708	-0.712618	0.207210	0.555963
334	0.121140	-0.397137	-0.377335	-0.972840	0.187052	0.055486	-0.000495	-0.029690	-1.693337
335	-0.003296	-0.065343	-0.517645	-1.665630	-0.590529	-0.234137	0.418808	-0.346840	-2.187593
336	0.270248	-0.289164	-0.403850	0.366331	-0.086712	0.160995	0.040968	0.363213	-0.076785
337	-0.030025	-0.277541	1.159076	-0.538880	-0.013956	1.099290	0.974565	-0.239682	0.092976

338 rows × 809 columns

Principal Component Analysis

Calculate Principal Components

In [30]:

```
from sklearn.decomposition import PCA

information = 225; #15^2=225;
PrinCom=PCA(n_components=information, random_state = SEED)
PrinCom.fit(X_train)

#save as new variable for PCs so as to not tamper with old variable
Z_train = PrinCom.transform(X_train);
Z_test  = PrinCom.transform(X_test);
Z       = PrinCom.transform(x)

print('Train set shape = ',Z_train.shape, '\nTest set shape  = ',Z_test.shape)


pd.DataFrame(Z).describe() #Data No longer Standard
print(f"Using the first {Z.shape[1]} Principal Components describes {np.round(PrinCom.expl
pd.DataFrame(Z)
```

```
Train set shape =  (270, 225)
Test set shape  =  (68, 225)
Using the first 225 Principal Components describes 100.0% of the data.
```

Out[30]:

	0	1	2	3	4	5	
0	-188799.714527	-23073.518325	-9801.856529	9566.113542	-14332.068735	-23187.043665	2995.2264
1	-188799.762822	-23073.511101	-9801.836145	9566.094555	-14331.974606	-23187.555427	2995.2112
2	-188799.538669	-23073.571970	-9801.963634	9566.308546	-14332.134694	-23186.135988	2995.0849
3	-188799.763902	-23073.506342	-9801.835448	9566.090586	-14331.999563	-23187.476976	2995.2126
4	-188799.012054	-23073.602369	-9802.169941	9566.541268	-14332.831484	-23180.039102	2995.5483
...	...	...	...	...	...	...	
333	-188789.845100	-23064.588811	-9804.542697	9569.568994	-14333.395024	-23188.534687	2995.7787
334	-188799.688663	-23073.579169	-9801.823081	9566.098257	-14332.109641	-23186.989600	2995.1499
335	-188799.024606	-23073.593786	-9802.170699	9566.550611	-14332.782422	-23180.225029	2995.5454

	0	1	2	3	4	5	
336	-188799.701441	-23073.536908	-9801.860717	9566.113083	-14331.995001	-23187.420330	2995.2175
337	-188798.872897	-23072.944347	-9801.793007	9565.975333	-14331.777909	-23187.911325	2995.1179

338 rows × 225 columns

Scaling the Principal Components

In [31]:

```
## Ignore for now - Ask Ahmad later [Al&Mus]
# scZ = StandardScaler();
# scZ.fit(Z_train);
# Z = scZ.transform(Z)
# Z_train = scZ.transform(Z_train)
# Z_test = scZ.transform(Z_test)
# pd.DataFrame(Z).head()

scZ = StandardScaler();
scZ.fit(Z);
Z = scZ.transform(Z)
pd.DataFrame(Z).head()
```

Out[31]:

	0	1	2	3	4	5	6	7	8	
0	-0.417982	0.031656	0.230817	-0.228082	-0.306366	-0.006241	0.049221	0.134513	-0.322128	-0.24
1	-0.431762	0.034535	0.248130	-0.244527	-0.192686	-0.261985	0.024957	0.307369	-0.408121	-0.32
2	-0.367806	0.010274	0.139843	-0.059181	-0.386026	0.447356	-0.177100	0.221564	0.185753	0.35
3	-0.432070	0.036432	0.248722	-0.247965	-0.222827	-0.222781	0.027233	0.294052	-0.365110	-0.31
4	-0.217551	-0.001842	-0.035391	0.142390	-1.227541	3.494176	0.564089	-2.462506	-0.355460	0.79

5 rows × 225 columns

Image Creation

Rescaling and Reshaping

In [32]:

```
## Scale all the PCA components on 0-256 (image greyscale range)


def gen_images(data):
    images=[];
    for r in range(0,data.shape[0]): #Cycle over rows
        pixels=[];
        mini=min(data[r,:])
        maxi=max(data[r,:])
        m = (maxi-mini)/(256);

        for c in range(0,225): #Cycle over cols
            curPixel = data[r,c]
            pixels.append((((curPixel - mini) / (maxi - mini)) * 255.9).astype(np.uint8))
```

```
        #once cols are done running add the image to the images[] array
        img = np.reshape(pixels, (15,15)); #reshape into a square image
        images.append(img)

    return images;

#Generate images from each data split
all_images  = gen_images(Z)
x_train_img = gen_images(Z_train)
x_test_img  = gen_images(Z_test)

#Get number of rows in each data split
height_total = Z.shape[0];
height_train = Z_train.shape[0];
height_test =  Z_test.shape[0];

#Reshape into input shape for CNN models
#15,15,1 indicates a 15x15 pixel greyscale image
x_train_img = np.array(x_train_img).reshape(height_train,15,15,1)
x_test_img  = np.array(x_test_img).reshape(height_test,15,15,1)
all_images  = np.array(all_images).reshape(height_total,15,15,1)
```

Saving Images

In [33]:

```
import imageio

# Relative paths to saved folders
TiredImgFolder = cwd + r"\Data\GeneratedImages\Tired"
NotTiredImgFolder = cwd + r"\Data\GeneratedImages\Not Tired"

# Clear the folders
import glob

files = glob.glob(TiredImgFolder + r"\*")
for f in files:
    os.remove(f)

files = glob.glob(NotTiredImgFolder + r"\*")
for f in files:
    os.remove(f)


ctr1=0;
ctr2=0;
for img in all_images.reshape(height_total,15,15):

    if (y[ctr1+ctr2] == 0): #Not Tired
        fstr = NotTiredImgFolder + r"\img_" + str(ctr1) + r".png"
        imageio.imwrite(fstr, img[:, :], dpi=(300,300))
        ctr1+=1; #Counter
    else:
        fstr = TiredImgFolder + r"\img_" + str(ctr2) + r".png"
        imageio.imwrite(fstr, img[:, :], dpi=(300,300))
        ctr2+=1;
```

Image Feature Generation

Datagen Definitions

Generate image data for each of the images - allows us to generate more images to increase input to CNN

In [34]:
```
from keras.preprocessing.image import ImageDataGenerator

#rpath to IMageFolder
genimgsPath = cwd + r"\Data\GeneratedImages"

r  = 1       #rescale
sr = 0.2     #shear range
zr = 0.2     #zoom range
hf = False   #horizontal flip

ValidationSplit = 0.2

imageGenerator = ImageDataGenerator(rescale = r,
                                    shear_range = sr,
                                    zoom_range = zr,
                                    horizontal_flip = hf,
                                    validation_split=ValidationSplit)
```

Generating Features

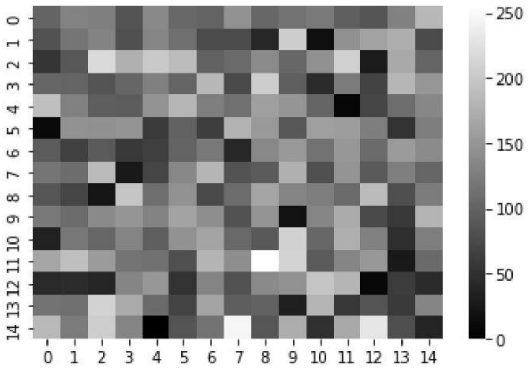
In [35]:
```
imgs_train = imageGenerator.flow_from_directory(genimgsPath,
                                                target_size = (15, 15),
                                                batch_size = 32,
                                                subset="training",        #creates traini
                                                class_mode='categorical',
                                                shuffle=True,
                                                color_mode="grayscale")

imgs_test = imageGenerator.flow_from_directory(genimgsPath,
                                               target_size = (15, 15),
                                               batch_size = 32,
                                               subset="validation",       #creates test su
                                               class_mode='categorical',
                                               shuffle=True,
                                               color_mode="grayscale")
print(imgs_test.class_indices)
```

```
Found 272 images belonging to 2 classes.
Found 66 images belonging to 2 classes.
{'Not Tired': 0, 'Tired': 1}
```

Image Examples

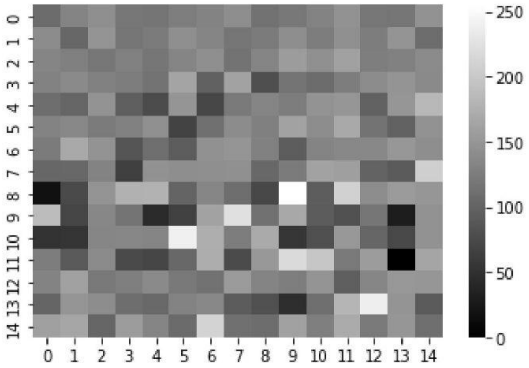
In [62]:
```
#change n to see a different data image
n = 104
sns.heatmap(all_images.reshape(height_total,15,15)[n], cmap='gray');
print(f'This image is for the: \"{y_labeled[n]}\" class.')
```

```
This image is for the: "Tired" class.
```



In [37]:
```
#nth row of data
n=252
sns.heatmap(all_images.reshape(height_total,15,15)[n], cmap='gray');
print(f'This image is for the: \"{y_labeled[n]}\" class.')
```

```
This image is for the: "Not Tired" class.
```

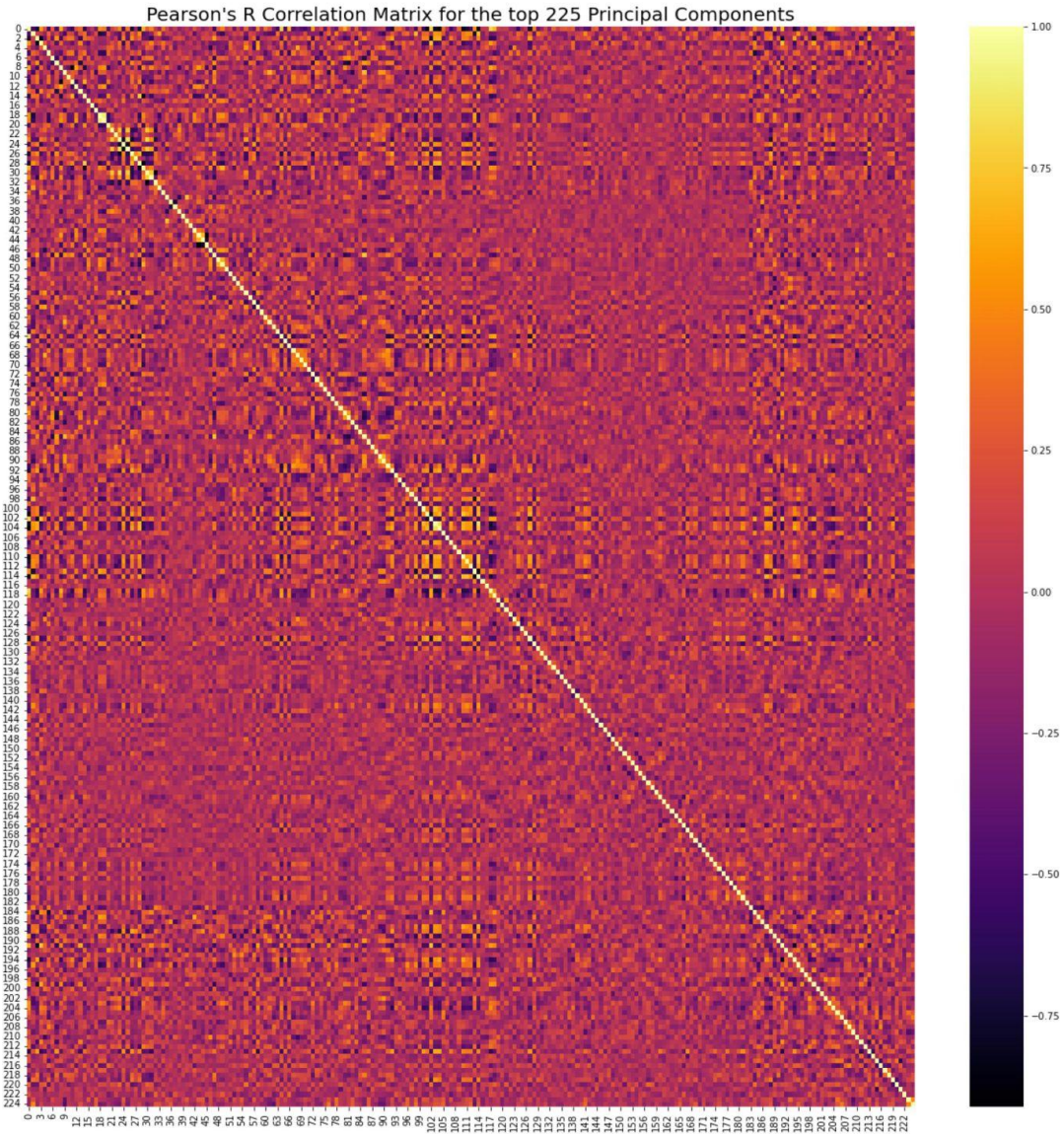


Data Exploration

General Correlation Matrix for Principal Components

In [38]:
```
corr_mat = pd.DataFrame(Z).corr(method='pearson');
#mask = np.triu(np.ones_like(corr_mat, dtype=bool));
plt.figure(dpi=300);
plt.subplots(figsize=(21,21));
plt.title("Pearson's R Correlation Matrix for the top 225 Principal Components", fontsize=
sns.heatmap(corr_mat, annot=False, lw=0, linecolor='white', cmap='inferno');
#print('Too many features to visualize at once!')
```

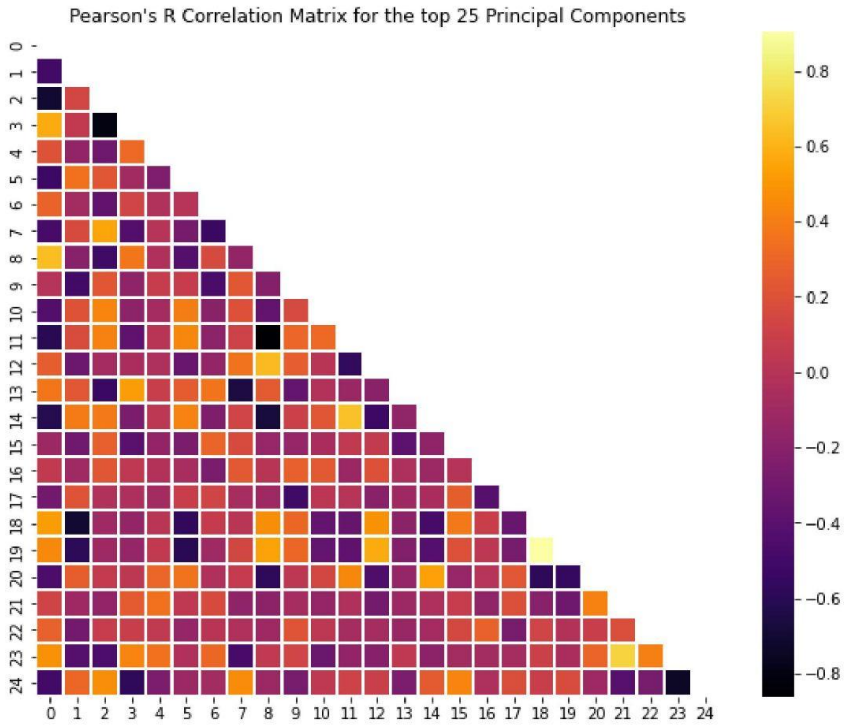
```
<Figure size 1800x1200 with 0 Axes>
```



In [39]:

```
corr_mat = pd.DataFrame(Z[:,0:25]).corr(method='pearson');
mask = np.triu(np.ones_like(corr_mat, dtype=bool));
plt.figure(dpi=300);
plt.subplots(figsize=(10,8));
plt.title("Pearson's R Correlation Matrix for the top 25 Principal Components", fontsize=1
sns.heatmap(corr_mat, annot=False, lw=0.2, linecolor='white', cmap='inferno', mask=mask);
#print('Too many features to visualize at once!')
```

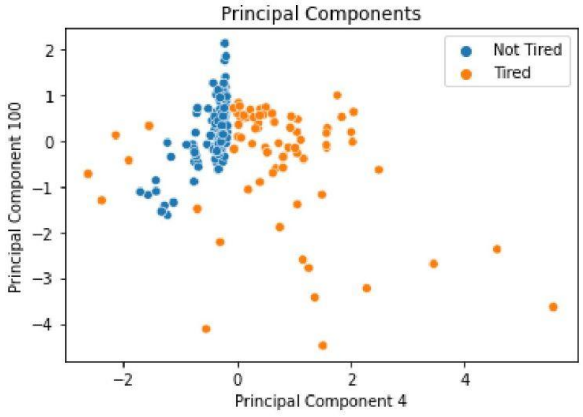
```
<Figure size 1800x1200 with 0 Axes>
```



Plotting the Principal Components

In [40]:

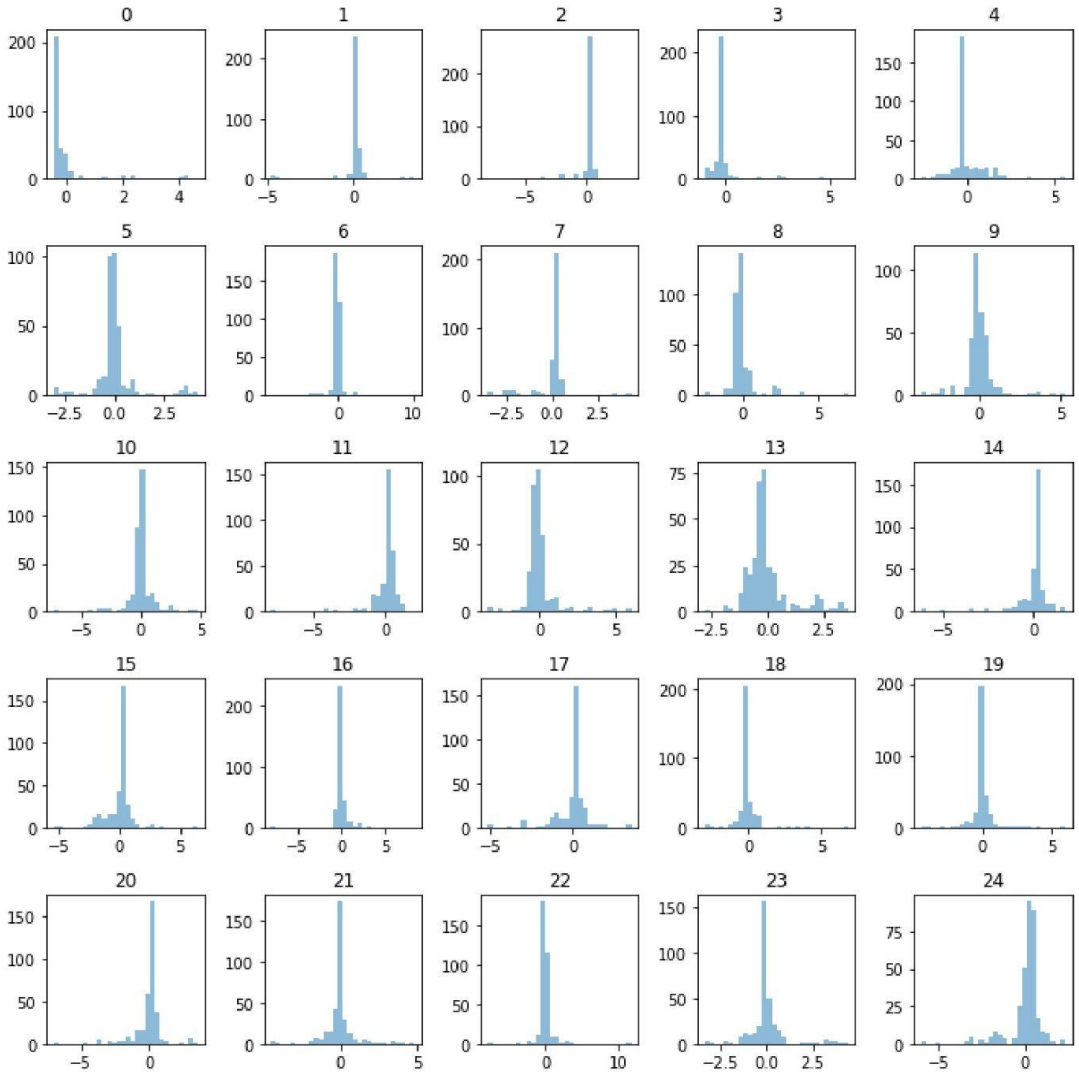
```
p1=4;
p2=100;
ax1 = sns.scatterplot(x=Z[:,p1], y=Z[:,p2], hue=y_labeled);
ax1.set(title='Principal Components',
        ylabel=f'Principal Component {p2}',
        xlabel=f'Principal Component {p1}');
```



PC Distributions

In [41]:

```
pc_title=[];
for i in range(1,25):
    pc_title.append(f'Principal Component {i}');

Z25 = Z[:,0:25]

import warnings
warnings.filterwarnings('ignore')
with warnings.catch_warnings():       #Catch warnings in code section
    warnings.simplefilter("ignore")

    plt.subplots(figsize=(10,10));
    ax = plt.gca();
    pd.DataFrame(Z25).hist(bins=30, figsize=(1,1), grid=False, layout=(5,5), sharex=False,
    plt.tight_layout();
```



ML Models

Definitions

In [42]:
```
from sklearn.model_selection import GridSearchCV
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
from sklearn.metrics import accuracy_score
from sklearn import model_selection
from sklearn.model_selection import StratifiedKFold

#Callbacks
from keras.callbacks import EarlyStopping
es = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=10, restore_best_we
```

Basic ANN Model

In [43]:
```
#array to hold model info (str: name, model: model, data_to_take: z/img)
models = [];
```

In [44]:
```
def build_basicANN(optimizer='adam', epochs=100, batch_size=50, neurons=225):

    #Initializing ANN
    m= tf.keras.models.Sequential()

    #Add input layer
    m.add(tf.keras.layers.Dense(units=neurons, activation='relu'))

    #Add hidden layer
    m.add(tf.keras.layers.Dense(units=(neurons/2), activation='relu'))

    #Add output layer
    m.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))

    #Compiling ANN
    m.compile(optimizer = optimizer, loss = 'binary_crossentropy', metrics = ['accuracy'])

    #Return compiled, unfitted model
    return m;
```

In [45]:
```
%%time

#Build Model, Using defaults
## mANNBasic = build_basicANN()
mANNBasic = (KerasClassifier(build_fn=build_basicANN, epochs=100, batch_size=50, optimizer

#Training ANN
hist_ANNBasic = mANNBasic.fit(Z_train, y_train, batch_size = 100, epochs = 100, verbose=0)

models.append(('ANN Basic', mANNBasic, 'z'))
```

```
Wall time: 1.01 s
```

In [46]:
```
print(f'Accuracy of the unoptimized Basic ANN model = {round(accuracy_score(y_true=y_test,
```

```
Accuracy of the unoptimized Basic ANN model = 100.0%
```

Basic CNN Model

See report for info on how we defined "Basic" vs "Advanced" CNN

In [47]:
```
# Random-ish architecture

def build_basicCNN(optimizer='adam', epochs=100, batch_size=50, neurons=225):

    m = tf.keras.models.Sequential()
    m.add(tf.keras.layers.Conv2D(filters=neurons, kernel_size=3, activation='relu', input_
    m.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
    m.add(tf.keras.layers.Conv2D(filters=neurons/2, kernel_size=3, activation='relu'))
    m.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
    m.add(tf.keras.layers.Flatten())
    m.add(tf.keras.layers.Dense(units=neurons, activation='relu'))
    m.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
    m.compile(optimizer = optimizer, loss = 'binary_crossentropy', metrics = ['accuracy'])

    return m;
```

In [48]:
```
%%time

#Build model using defaults
#mCNNBasic = build_basicCNN()
mCNNBasic = (KerasClassifier(build_fn=build_basicCNN, epochs=100, batch_size=50, optimizer

### ORIGINAL
hist_CNNBasic = mCNNBasic.fit(x=x_train_img,
                              y=y_train,
                              batch_size = 50,
                              epochs = 100,
                              verbose=0,
                              callbacks=es,
                              validation_data=(x_test_img, y_test))


models.append(('CNN Basic', mCNNBasic, 'img'))
```

```
Restoring model weights from the end of the best epoch.
Epoch 00083: early stopping
Wall time: 12.8 s
```

In [49]:
```
print(f'Accuracy of the unoptimized Basic CNN model = {round(accuracy_score(y_true=y_test,
```

```
Accuracy of the unoptimized Basic CNN model = 100.0%
```

In [50]:
```
%%time
#### DATAGEN -- DOES NOT USE KERASCLASSIFIER DUE TO ERROR
mCNNBasic2 = build_basicCNN()
hist_CNNBasic2 = mCNNBasic2.fit(
                              x=imgs_train,
                              #y=y_train,
                              batch_size = 50,
                              epochs = 100,
                              verbose=0,
                              callbacks=es,
```

```
                                validation_data=imgs_test
                                )
```

```
Restoring model weights from the end of the best epoch.
Epoch 00051: early stopping
Wall time: 14.1 s
```

Advanced CNN Model

In [51]:

```
def build_advancedCNN(optimizer='adam', epochs=100, batch_size=50, neurons=225):
    #params
    initFilt = neurons;
    initUnits= neurons;

    #model
    m = tf.keras.models.Sequential([
        tf.keras.layers.Conv2D(filters=initFilt, kernel_size=3, activation='relu', input_s
        tf.keras.layers.Conv2D(filters=initFilt/2, kernel_size=3, activation='relu'),
        tf.keras.layers.MaxPool2D(pool_size=2, strides=2),
        tf.keras.layers.Dropout(0.25),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(units=initUnits),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(units=1, activation='sigmoid')
    ])

    m.compile(optimizer = optimizer, loss = 'binary_crossentropy', metrics = ['accuracy'])

    return m;
```

In [52]:

```
%%time

# build using defaults
#mCNNAdvanced = build_advancedCNN()
mCNNAdvanced = (KerasClassifier(build_fn=build_advancedCNN, epochs=100, batch_size=50, opt

#fit
hist_CNNAdvanced = mCNNAdvanced.fit(x_train_img,
                        y=y_train,
                        batch_size = 50,
                        epochs = 100,
                        verbose=0,
                        callbacks=es,
                        validation_data=(x_test_img, y_test))

models.append(('CNN Advanced', mCNNAdvanced, 'img'))
```

```
Restoring model weights from the end of the best epoch.
Epoch 00074: early stopping
Wall time: 44.2 s
```

In [53]:

```
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR) #ignore warnings

print(f'Accuracy of the unoptimized Advanced CNN model = {round(accuracy_score(y_true=y_te
```

```
Accuracy of the unoptimized Advanced CNN model = 100.0%
```

In [54]:

```
%%time
```

```
#### DATAGEN -- DOES NOT USE KERASCLASSIFIER DUE TO ERROR
mCNNAdvanced2 = build_advancedCNN()
hist_CNNAdvanced2 = mCNNBasic2.fit(
                                x=imgs_train,
                                #y=y_train,
                                batch_size = 50,
                                epochs = 100,
                                verbose=0,
                                callbacks=es,
                                validation_data=imgs_test
                                )
```

```
Restoring model weights from the end of the best epoch.
Epoch 00011: early stopping
Wall time: 3.81 s
```

Random Forest Model

In [55]:

```
%%time
from sklearn.ensemble import RandomForestClassifier
RFCmodel = RandomForestClassifier(n_estimators=100); #N_estimators and criterion can be o
RFCmodel.fit(Z_train, y_train);
models.append(('RF', RFCmodel, 'z'));
```

```
Wall time: 352 ms
```

Logistic Regression Model

In [56]:

```
from sklearn.linear_model import LogisticRegression
LRmodel = LogisticRegression(solver='newton-cg');
LRmodel.fit(Z_train, y_train);
models.append(('LR',LRmodel, 'z'));
```

Performance Comparison

Via K-Fold Cross-Validation

For SKLearn Models

In [57]:

```
%%time
#Suppress warnings for non-convergent ANN models
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)

# Number of splits to make.
N = 3;


CV_results = [];
scoring = 'accuracy';

trun=0;
for tp in models:

    #Check whether model uses Z dataset or images for training
```

```
    mode = tp[2];

    if (mode == 'z'):
        kfold = StratifiedKFold(n_splits=N, shuffle=True)
        #kfold = model_selection.KFold(n_splits=N);
        CVinternal_results = model_selection.cross_val_score(tp[1], Z, y, cv=kfold, scorin
        CV_results.append((CVinternal_results));

    if (mode == 'img'):
        kfold = StratifiedKFold(n_splits=N, shuffle=True)
        #kfold = model_selection.KFold(n_splits=N);
        CVinternal_results = model_selection.cross_val_score(tp[1], all_images, y, cv=kfol
        CV_results.append((CVinternal_results));

    print(f'run#{trun} for model \"{tp[0]}\" returned {CVinternal_results}')
    trun+=1;
```

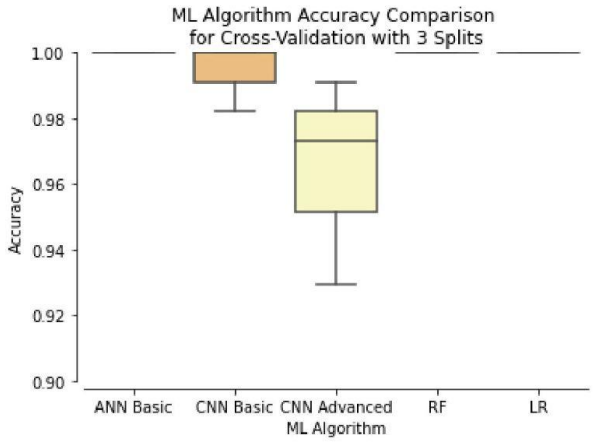
```
run#0 for model "ANN Basic" returned [1. 1. 1.]
run#1 for model "CNN Basic" returned [1.         1.         0.98214286]
run#2 for model "CNN Advanced" returned [0.99115044 0.92920354 0.97321429]
run#3 for model "RF" returned [1. 1. 1.]
run#4 for model "LR" returned [1. 1. 1.]
Wall time: 2min 55s
```

For Keras Models

NOTE : The Following was not implemented due to model overfitting by all developed ML models. The results of K-Fold CV are essentially useless until more data is compiled.

In [63]:

```
names = [];
for tp in models:
    names.append(tp[0]);

CVdf = pd.DataFrame(CV_results).T;
CVdf.columns = names;
CVdf.T

ax2 = sns.boxplot(data=CVdf, palette='Spectral')
ax2.set(xlabel = "ML Algorithm",
       ylabel = 'Accuracy',
       title = f"ML Algorithm Accuracy Comparison \nfor Cross-Validation with {N} Splits")
sns.despine(ax=ax2,offset=5, trim=False)
ax2.plot();
plt.ylim(0.90,1);
```



Conclusion

With the availability of sophisticated technology such as the Muse 2, it is valuable to understand how brainwave data is impacted by the level of fatigue experienced by a person. Our study aimed to determine whether a machine learning model can accurately predict if a person is fatigued or not simply by analyzing their brainwaves.

After collecting data for both fatigued and not fatigued individuals, randomizing and scaling the data and training five different machine learning models on the dataset, our study concluded that all of our models (Artificial Neural Network, two different Convolutional Neural Network, Random Forest and Logistic Regression) were able to predict fatigued or not with an accuracy of 100% or very close to it. Although our results show our study to be surprisingly promising, it is important to note machine learning models tend to overfit smaller amounts of data input resulting in an accuracy of 100%, as is the case in our study. Due to this reason, it is difficult to compare the accuracy of one model to another.

Future research into the correlation between brain activity and state of fatigue should aim to apply data collected over the span of many months. Furthermore, while our research focused on fatigue resulting from sleep deprivation, this can be expanded to include a wide range of both physical and mental fatigue.

13. Appendix B – Sample RAW Data

The following is an unabridged sample dataset as exported from the MindMonitor App. The table is split for visibility.

	TimeStamp	Delta_TP9	Delta_AF7	Delta_AF8	Delta_TP10	Theta_TP9	Theta_AF7	Theta_AF8	Theta_TP10	Alpha_TP9	...
0	2021-11-01 17:54:38.045	1.110457	-0.382196	0.082630	0.743808	0.455723	-0.523256	0.086015	0.487615	0.493558	...
1	2021-11-01 17:54:39.045	0.904642	-0.382196	0.236881	0.613098	0.313527	-0.523256	0.171247	0.546970	0.538756	...
2	2021-11-01 17:54:39.187	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
3	2021-11-01 17:54:40.045	0.652124	-0.382196	0.462323	0.410327	0.293693	-0.523256	0.267178	0.466408	0.343593	...
4	2021-11-01 17:54:41.043	0.558608	-0.382196	0.502156	0.877835	0.281408	-0.523256	0.337400	0.469669	0.381862	...
...	...	...	...	...	...	...	...	...	...	...	...
156	2021-11-01 17:56:57.042	1.011459	-0.382196	0.502156	0.955036	0.456557	-0.523256	0.337400	0.439388	0.575279	...
157	2021-11-01 17:56:57.111	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
158	2021-11-01 17:56:57.852	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...
159	2021-11-01 17:56:58.042	1.011459	-0.382196	0.502156	0.955036	0.456557	-0.523256	0.337400	0.439388	0.575279	...
160	2021-11-01 17:56:58.222	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...

...	Gyro_X	Gyro_Y	Gyro_Z	HeadBandOn	HSI_TP9	HSI_AF7	HSI_AF8	HSI_TP10	Battery	Elements
...	4.134674	-5.824432	-1.510315	1.0	1.0	2.0	1.0	1.0	70.0	NaN
...	4.329071	-2.990723	-1.644897	1.0	1.0	2.0	1.0	1.0	70.0	NaN
...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	/muse/elements/jaw_clench
...	5.622559	-5.099182	-0.732727	1.0	1.0	2.0	1.0	1.0	70.0	NaN
...	4.882355	-3.536530	-1.652374	1.0	1.0	2.0	1.0	1.0	70.0	NaN
...	...	...	...	...	...	...	...	...	...	...
...	5.510406	-7.880554	-2.257996	1.0	1.0	4.0	2.0	1.0	70.0	NaN
...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	/muse/elements/blink
...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	/muse/elements/blink
...	4.844971	-6.190796	-2.781372	1.0	1.0	4.0	4.0	1.0	70.0	NaN
...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	/muse/elements/blink