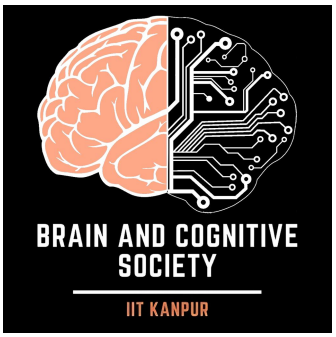


Drowsiness Detection

Endterm Report

Mentors:
Manasvi Nidugala
Zehaan Naik
Viral Chitlangia

Submitted by :
Aarsh Jain, Abhinav Sinha, Anupama Prabhakaran, Ayush Karan, Krish Jain, Pranay Saini, Shrey Solanki, Soham Roychowdhury, Vishwas Pathania, Vivek

Abstract

This report provides a detailed account of the work conducted thus far in our project focused on drowsiness detection using EEG signals and machine learning techniques. The project aims to develop a reliable and efficient system for detecting drowsiness taking inspiration from existing work in this domain.

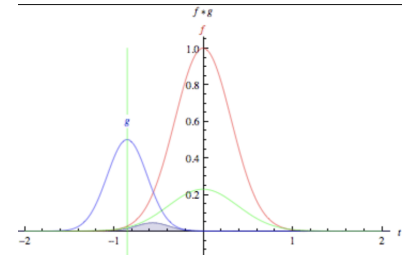
Contents

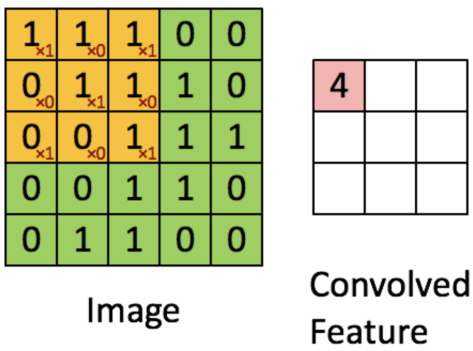
1. Introduction

Drowsiness detection is crucial for ensuring driver safety. Utilizing EEG signals to monitor brain activity offers a promising approach to assess drowsiness levels and alert individuals to prevent accidents. EEG, or electroencephalography, captures brain wave patterns that reflect cognitive states such as drowsiness. By analyzing these signals, we can extract valuable features that indicate changes in alertness. This documentation delves into the application of EEG-based drowsiness detection systems, exploring preprocessing techniques, feature extraction methods, and machine learning algorithms. Understanding these technologies is essential for developing effective systems aimed at reducing the risks associated with driver drowsiness.

2. Convolution

A convolution is an integral that expresses the overlap of one function g as it is shifted over another function f.





The green curve shows the convolution of the blue and red curves as a function of t, the position indicated by the vertical green line. The gray region indicates the product g(τ)f(t-τ) as a function of t, so its area as a function of t is precisely the convolution.

Functionality

Here's a step-by-step explanation of the process:

1. Flip and Shift: One of the functions, say g, is flipped horizontally. Then, it is shifted over to the domain of the other function f.

2. Pointwise Multiplication: At each shift position, the values of f and the shifted g are multiplied point by point.

3. Sum of Products: The results of these multiplications are summed up to produce a single value. This value represents the convolution of f and g at that particular shift.

4. Form the Convolution Function: This process is repeated for every possible shift, and the resulting values form a new function $(f * g)(t)$, which represents the convolution of f and g.

2.A. What exactly are CNNs

CNNs, a powerful type of machine learning, excel at computer vision tasks. These neural networks are designed to analyze grid-like data, particularly images, but at the very core, they are networks used to figure out the function that, when convolving our original data, gives us an output we can use. CNNs mimic the human brain's visual processing with stacked layers. These layers, convolutional, pooling, activation, and fully connected, work together to identify important patterns and relationships within images. Based on the task at hand we select the type, number and order of layers to include in our stack.

1. Convolutional Layers: These layers are like the "feature hunters" of our network. They slide small filters, called kernels, across the image to find specific patterns – edges, textures, or even more intricate shapes. This process also keeps track of where these features appear in the images in the output layer.

2. Pooling Layers: These layers act like "input shrinkers". They take the output from the convolutional layers and condense it, keeping only the most important information. This makes the

network run faster and prevents it from getting overloaded with too much data. There are several ways to shrink our data, some of them are

- Max Pooling: The most common way to pool data. It picks the single most intense pixel from a tiny neighbourhood in the image and passes it on to the next layer.
- Average Pooling: This method, unlike max pooling which finds the most intense pixel, calculates the average value within a local region. It can be useful for tasks where capturing the overall essence of a feature is more important than specific high points.
- Global Pooling: This type of pooling applies pooling (either max or average) to the entire feature map, reducing it to a single value for each channel. This is helpful when you want to find the overall presence of a feature rather than its specific location.

3. Activation Functions: As AndrewNg mentions in his lectures, a Neural Network without an activation function is just a set of matrix multiplications and we can replace all of them with just one giant matrix. This "flat" network can only learn very basic relationships. Activation functions add Non-Linearity to the network, allowing the network to learn more complex patterns. Some widely used activation functions are:
   - Rectified Linear Unit (ReLU): This is the most popular activation function, acting like a switch. It allows positive values to pass through unchanged but sets negative values to zero. This helps the network focus on the important features in the data.
   - Sigmoid Function: This "squishification" function gives out an output value between 0 to 1 for any input value, making it useful for tasks that require probabilities or when our neuron is taken to be a number having these constraints.
   - Softmax Function: This function is used in the output layer of a network for multi-class classification problems. It takes a set of values and transforms them into probabilities that sum to 1, for when we want the likelihood of all features representing our output.
4. Fully Connected Layers: These are the "decision makers" of the CNN. Unlike convolutional and pooling layers that operate on local features, these layers take the big picture into account. They receive the high-level features extracted from previous layers and connect each piece of information to everything else. By analyzing these intricate relationships, fully connected layers can make the final call

2.B. Training of a CNN

After we decide on our stack of layers, the next task is to train our neural network so that it is able to classify the data. CNNs are trained using a supervised learning approach. This means the network is presented with a dataset of labeled images. Each image has a corresponding label that identifies its content. The CNN's goal is to learn the mapping betwen these input images and their correct labels. The steps in the training process include:

- Data Preparation: The training data is preprocessed to ensure that all of them are in the same format and size.
- Forward Pass: Data and its label are fed into the CNN. The data travels through the network layer by layer and a prediction for the label is produced.
- Calculating Loss: The network's predicted label is compared to the actual label. We devise a loss function that acts as a "performance meter" during training. It quantifies how well the CNN's predictions match the actual labels of the training images.

- Optimization: Once we know the loss, we need to update our network in a way that the loss decreases, for this we choose different optimizers. The optimizer is the "trainer" which changes the parameters of the CNN to minimize the loss. Some examples of optimizers are Gradient Descent, Stochastic Gradient Descent, Adam etc.

- Backpropagation: As the optimizer figures out the changes it is very important to change each layer along the step and also pay attention to how they effect the accuracy of the network. Backpropagation essentially allows us to propagate the changes layer by layer.

- Repetition: The process of forward pass, loss calculation, and backpropagation is repeated numerous times over a set of training images, typically divided into batches. Each complete iteration through the training set is called an epoch. With each epoch, the network refines its internal parameters, becoming better at mapping input images to their correct labels.

2.C. Testing our network

After training, the CNN's performance is evaluated on a separate set of unseen data(testing set). This helps assess how well the network has generalized its learning ability and can handle new data it hasn't encountered during training. We can find several metrics of the data relating to the accuracy of it. Changing the properties of our network on these bases.

- Classification Accuracy: This is a common metric for evaluating CNNs in classification tasks. It simply calculates the percentage of images where the network's prediction matches the actual label. However, it also may be misleading in certain situations if our data set is skewed, having mostly one type of label, and the network keeps predicting that label.

- Confusion Matrix: This is a more informative way to evaluate a CNN's performance, especially for multi-class classification problems. It's a visual table that shows how many images were correctly classified into each category, and how many were misclassified into other categories.The confusion matrix helps identify potential biases or weaknesses in the network's predictions. Showing us exactly what data was misclassified and into what label.

- Precision: Measures the proportion of true positives out of all positive predictions. In simpler terms, it tells you how many of the images the network classified as a particular class. This is helpful when you have a skewed data set.

- Recall: Measures the proportion of true positives out of all actual positives in the dataset for a particular class. In other words, it tells you what percentage of actual positives in the test set were correctly identified by the network.

3. Mini Project 1

In this chapter, we will be using the MNIST dataset, which is a set of 70,000 small images of digits handwritten by high school students and employees of the US Census Bureau. We will train a CNN model and finally predict the image label by the image.

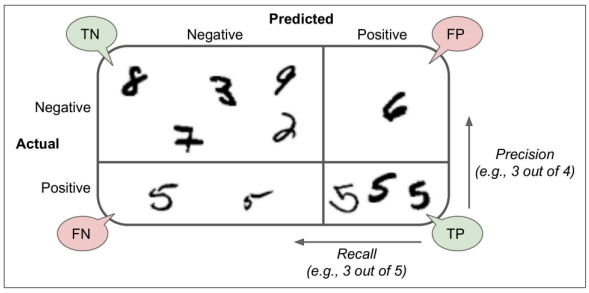
3.A. MNIST Dataset

Each image is labelled with the digit it represents. Scikit-Learn provides many helper functions to download popular datasets. MNIST is one of them. There are 70,000 images, and each image has 784 features. This is because each image is 28×28 pixels, and each feature simply represents one pixel's intensity, from 0 (white) to 255 (black). The MNIST dataset is actually already split into a training set (the first 60,000 images) and a test set (the last 10,000 images).

some images from the MNIST dataset

3.B. Confusion Matrix

A good way to calculate the performance of a classifier is to look at the confusion matrix. There are two terms: *precession* and *recall*. *Precession* is the ratio of True positives to the total number of True Positives and False Positives and *recall* is the ratio of the True Positives to the total number of True Positives and False negatives.



confusion matrix

3.C. Loading the data

After importing the required libraries as shown in the below image, we load the data through keras.datasets.mnist and labelled the training data xtrain and ytrain and testing data as xtest and ytest and scaled the data by dividing it by 255 for better accuracy.

3.D. Description

In this project we compared the ANN model with CNN model for digit recognition by comparing the accuracies of their respective models.

3.D.I. Disadvantages of ANN model

Some of the disadvantages are: a) It involves too much computation. b) It treats local pixels the same as pixels far apart. c) It is sensitive to location of an object in an image.

3.D.II. Advantages of CNN model

Some of the advantages are: a) It extracts the features first and then does the classification(as shown in the image below). b) Conv+pooling gives location invariant feature detection. c) Connections sparsity reduces overfitting.

3.E. ANN Model

```
ann_model=models.Sequential([
    layers.Flatten(input_shape=(28,28)),
    layers.Dense(100,activation='relu'),
    layers.Dense(10,activation='sigmoid')
])
ann_model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
history=ann_model.fit(x_train,y_train,epochs=5)
final_accuracy = history.history['accuracy'][-1]
print("Accuracy:", final_accuracy * 100, "%")
```

In the ANN model we have 3 layers:

- The first layer flattens the input matrix and here we have to specify the input shape as it is the first layer.
- Second layer is a deep layer with 100 neurons with activation 'relu' as it helps in non linearity and better accuracy.
- Last layer contains 10 categories i.e. the 10 digits.

We used 'adam' as optimizer and 'sparse-categorical-crossentropy' as our loss function and run the model through 5 epochs.

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 2ms/step - accuracy: 0.8757 - loss: 0.4463
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9605 - loss: 0.1361
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9740 - loss: 0.0883
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9808 - loss: 0.0635
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9838 - loss: 0.0518
Accuracy: 98.37499856948853 %
```

By importing the confusion matrix and classification report we get the following results

```
from sklearn.metrics import confusion_matrix , classification_report
y_predict=ann_model.predict(x_test)
y_predict_class=[np.argmax(element) for element in y_predict] # gives the element
print("classification report: \n",classification_report(y_test,y_predict_class))
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
classification report:
              precision    recall  f1-score   support

           0       0.98      0.99      0.99       980
           1       0.98      0.99      0.98      1135
           2       0.98      0.97      0.98      1032
           3       0.93      0.99      0.96      1010
           4       0.98      0.97      0.98       982
           5       0.98      0.97      0.97       892
           6       0.98      0.98      0.98       958
           7       0.97      0.98      0.97      1028
           8       0.98      0.95      0.96       974
           9       0.98      0.94      0.96      1009

    accuracy                           0.97     10000
   macro avg       0.97      0.97      0.97     10000
weighted avg       0.97      0.97      0.97     10000
```

3.F. CNN MODEL

Typical CNN architectures stack a few convolutional layers (each one generally followed by a ReLU layer), then a pooling layer, then another few convolutional layers (+ReLU), then another pooling layer, and so on. The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper. It is generally preferable to stack two layers with 3×3 kernels: it will use less parameters and require less computations, and it will usually perform better. The number of filters grows as we climb up the CNN towards The output layer since the number of low level features is often fairly low. It is a common practice to double the number of filters after each pooling layer: since a pooling layer divides each spatial dimension by a factor of 2. Our CNN model consists of the following layers:

- It consists of three conv2D+MaxPooling layer combination which is for feature extraction.
- The filters go on increasing with the depth as we extract more and more complex features.
- The kernel size used is (3,3) and the activation function is 'relu' for non-linearity and better accuracy.
- then after feature extraction layers we have the classification layers.
- We have the flatten layer and then a deep layer and the last layer consists of the 10 categories i.e. the 10 digits.

Then we run our model through 5 epochs and got the following results We got the following accuracy on the test data

```
cnn_model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy'])
```

```
history1=cnn_model.fit(x_train,y_train,epochs=5)
```

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 60s 29ms/step - accuracy: 0.8773 - loss: 0.3986
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 57s 31ms/step - accuracy: 0.9786 - loss: 0.0701
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 61s 32ms/step - accuracy: 0.9856 - loss: 0.0464
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 59s 31ms/step - accuracy: 0.9880 - loss: 0.0342
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 77s 29ms/step - accuracy: 0.9912 - loss: 0.0260
```

By importing the confusion matrix and classification report we get the following results

3.G. RESULTS

We have developed two machine learning models, an Artificial Neural Network (ANN) and a Convolutional Neural Network (CNN), to tackle our classification problem. The performance of these models, as measured by their accuracy, demonstrates their effectiveness:

- ANN Model Accuracy: 98.37
- CNN Model Accuracy: 99.08

The results indicate that both models perform exceptionally well, achieving high accuracy rates. However, the CNN model outperforms the ANN model with an accuracy percentage of 99.08, compared to the ANN's accuracy percentage of 98.37. This marginal increase in accuracy suggests that the CNN model is slightly better suited for our task, likely due to its ability to capture spatial hierarchies in the data more effectively. In conclusion, while both models are highly effective, the CNN model is the preferred choice for this classification problem given its superior performance

4. EEG

4.A. Introduction

EEG, or electroencephalography, is a technique that records electrical activity from the brain. Typically, it is recorded non-invasively, from electrodes placed on the scalp, although it can also be recorded from electrodes placed directly on the surface of the brain. It captures the

```python
cnn_model=models.Sequential([
    #feature extrction(cnn):
    layers.Conv2D(filters=32,kernel_size=(3,3),activation='relu',input_shape=(28,28,1)),
    layers.MaxPooling2D((2,2)),

    layers.Conv2D(filters=64,kernel_size=(3,3),activation='relu'),
    layers.MaxPooling2D((2,2)),

    layers.Conv2D(filters=128,kernel_size=(3,3),activation='relu'),
    layers.MaxPooling2D((2,2)),
    #classification(dense):
    layers.Flatten(),
    layers.Dense(100,activation='relu'),
    layers.Dense(10,activation='softmax')
])
```

Figure 1: CNN Model

```python
from sklearn.metrics import confusion_matrix , classification_report
y_predict=cnn_model.predict(x_test)
y_predict_class=[np.argmax(element) for element in y_predict] # gives the element
print("classification report: \n",classification_report(y_test,y_predict_class))
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step
classification report:
              precision    recall  f1-score   support

           0       0.99      0.98      0.99       980
           1       1.00      0.99      1.00      1135
           2       1.00      0.98      0.99      1032
           3       0.99      0.99      0.99      1010
           4       1.00      0.97      0.99       982
           5       0.96      0.99      0.98       892
           6       0.99      0.99      0.99       958
           7       0.98      0.99      0.99      1028
           8       0.99      0.99      0.99       974
           9       0.98      0.98      0.98      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000
```

Figure 2: Classification Report of CNN Model

brain's spontaneous electrical activity over a period of time. It can be seen from Figure 3(a that The EEG data is in the form of a graph of microvolts vs milliseconds. Each channel represents a single electrode. It is a time series data

4.B. Artifacts

These are unwanted electrical signals that can obscure or mimic true brain activity. They can arise from various sources: Physiological Artifacts: Eye Movements: Blinking and eye movements produce electrical signals. Muscle Contractions: Electrical signals are produced during muscle movements of the subject which are often picked up by the EEG device. Cardiac Activity: The heart's electrical activity interferes with EEG signals. External Artifacts: Environmental Noise: Electrical devices affect EEG recordings. The powerline frequency is 50 Hz in India which causes the EEG data to have a slight spike at 50 Hz.

4.C. Brain Waves and Their Significance

1. Delta Waves (0.5-4 Hz): Predominant during deep sleep (stages 3 and 4 of NREM sleep). High delta activity can indicate deep sleep.

2. Theta Waves (4-8 Hz): Commonly observed during light sleep (stage 1 of NREM sleep) and drowsiness. Increased theta activity is a key indicator of drowsiness.

3. Alpha Waves (8-13 Hz): Typically seen during relaxed wakefulness, especially with closed eyes. Reduced alpha activity

can indicate increased cognitive load or transition to sleep.

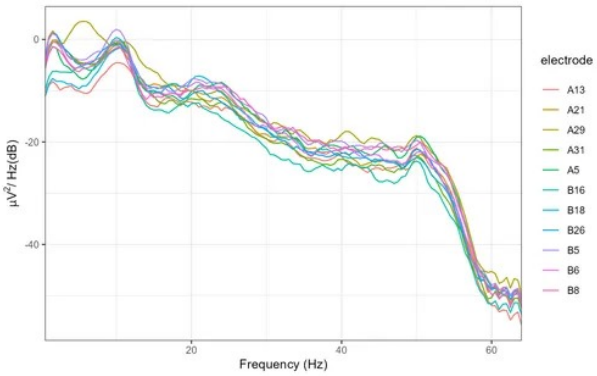
4. Beta Waves (13-30 Hz): Associated with active thinking, concentration, and anxiety. Increased beta activity can be seen in alert states

4.D. PSD Plots

Power Spectral Density (PSD) plots are graphs that show how the power (strength) of a signal is distributed across different frequencies. What Does a PSD Plot Look Like? See figure

- X-Axis : This represents the frequencies, usually measured in Hertz (Hz). It shows a range from low frequencies to high frequencies.
- Y-Axis : This represents the logarithm of power of the frequencies, showing how strong the signal is at each frequency.
- The different color curves represent the different channels.

A PSD plot helps you see which frequencies are more dominant in a signal. For example, in an EEG signal, certain frequencies might be stronger when a person is awake, and others might be stronger when they are drowsy or asleep.



(1)

4.E. Filtering in EEG

Filtering is essential in EEG signal processing to isolate relevant frequency bands and remove noise. For EEG common bands include:

1. Band-pass Filters: Designed to pass frequencies within a certain range and attenuate frequencies outside that range
2. Low Pass Filters : It allows frequencies less than a certain frequency to pass.
3. High pass filters: It allows frequencies greater than a certain frequency to pass.
4. Notch Filters: Remove specific frequencies, such as the 50/60 Hz power line interference, which can contaminate EEG signals

5. Mini-Project MNE

5.A. Introduction

Our experimental design was based on the oddball paradigm, which involves presenting participants with a sequence of repetitive standard stimuli interspersed with infrequent target stimuli. The brain's response to these unexpected target stimuli is of particular interest, as it can provide insights into cognitive processes such as attention and memory.

In this project, we focused on preprocessing, analyzing, and visualizing EEG data collected during an oddball paradigm experiment. The preprocessing steps included filtering the data, removing artifacts using Independent Component Analysis (ICA), and segmenting the data into epochs around each stimulus. Additionally, we conducted time-frequency analysis using Morlet wavelets to explore the dynamic changes in frequency content over time. These analyses allowed us to investigate how the brain's electrical activity differs in response to standard and target stimuli, providing insights into the neural mechanisms underlying cognitive processing of unexpected events.

5.B. Experiment

5.B.I. ICA Toolkit for Artifact Removal

After initial data acquisition using `mne.read_raw_fif()`, the EEG data was loaded into the raw object for preprocessing. Our approach focused on enhancing data quality by filtering and removing artifacts, particularly eyeblink artifacts, using Independent Component Analysis (ICA) within the MNE-Python environment.

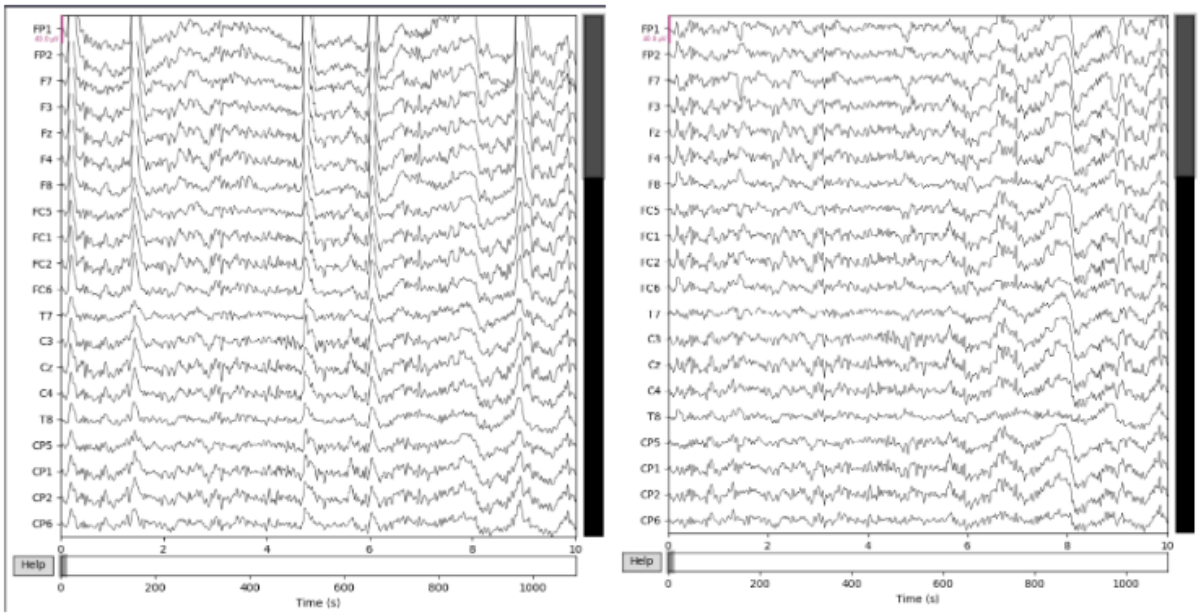


Fig. (a) Fig. (b)

Figure 3: Fig. (a) and (b) shows the EEG data before and after ICA processing

5.B.I.1. Data Preparation: Raw EEG data was loaded and inspected using `mne.read_raw_fif()` to ensure data integrity.

5.B.I.2. Data Filtering: Prior to ICA decomposition, the data underwent bandpass filtering (e.g., 1-40 Hz) to remove high-frequency noise and low-frequency drifts, ensuring that subsequent analyses focused on relevant neural signals.

5.B.I.3. ICA Decomposition: ICA was applied using the `mne.preprocessing.ICA()` function to decompose the preprocessed EEG data into independent components. Parameters such as the number of components to decompose and other ICA-specific settings were optimized to effectively separate neural components from artifacts.

Listing 1: ICA Decomposition

```
ica = mne.preprocessing.ICA(
    n_components=20,
    random_state=0)
ica.fit(raw.copy().filter(8,
    35))
```

5.B.I.4. Artifact Identification and Removal: Independent components were visually inspected to identify those representing artifacts, such as eye blinks and muscle activity. Artifactual components were removed from the data using the `ica.exclude` attribute in MNE-Python.

5.B.I.5. Applying Changes: The identified changes, including artifact removal, were applied to the original EEG data using the `ica.apply()` method. This step involved transforming the cleaned components back into the sensor space, resulting in an EEG dataset free from the identified artifacts.

Listing 2: Applying ICA Changes

```
ica.apply(raw.copy(), exclude
    =ica.exclude).plot()
```

5.B.II. Events and Epoching

After preprocessing the EEG data to remove artifacts using ICA, the next step involved segmenting the continuous data into epochs based on the experimental events of interest. This process is crucial for isolating and analyzing the brain's responses to specific stimuli or conditions.

5.B.II.1. Event Handling: Events of interest, such as the onset of target and standard stimuli in the oddball paradigm, were identified and extracted from the continuous EEG data. Event markers, including their timing and type (e.g., target or standard), were extracted using functions like `mne.find_events()` in MNE-Python.

5.B.II.2. Epoching: EEG data was segmented into epochs centered around each event of interest. For instance, epochs were defined from a specified time window before to after each event onset (e.g., -200 ms to 800 ms relative to stimulus onset). Epochs were extracted using functions like `mne.Epochs()` in MNE-Python, which allows for flexible handling of epoch creation based on event markers.

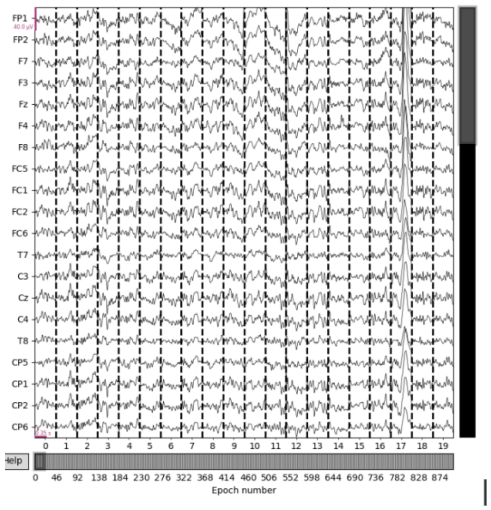
Listing 3: Epoching

```
event_ids = {"standard/
    stimulus": 200, "target/
    stimulus": 100}
epochs = mne.Epochs(raw,
    events, event_id=event_ids
    , preload=True)
```

5.B.II.3. Baseline Correction: To establish a baseline for comparison, the EEG signal within each epoch was baseline corrected by subtracting the mean amplitude of a pre-stimulus baseline period.

Listing 4: Baseline Correction

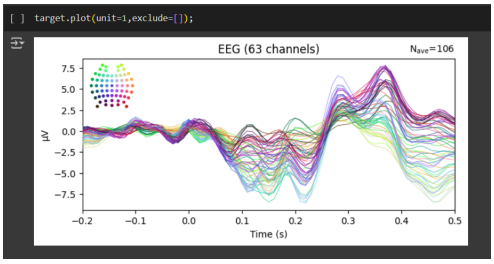
```
epochs.apply_baseline((None, 0))
```



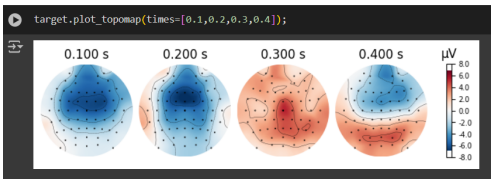
5.B.III. Evoked Potentials Analysis

5.B.III.1. Average Calculation: We computed the average of epochs for target/standard stimuli to analyze the brain's response to infrequent target/standard events and improve the signal-to-noise ratio.

5.B.III.2. Average Visualization: We visualized the average response for target/standard stimuli to observe the neural activity patterns.



5.B.III.3. Topographic Maps: We generated topographic maps at various time points for the target stimuli to illustrate the spatial distribution of neural activity.

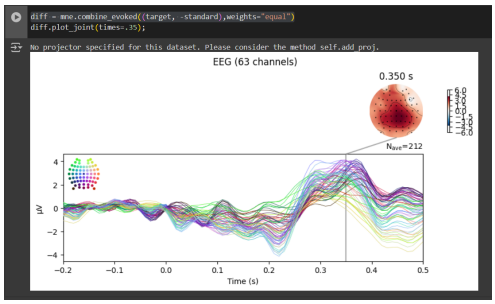


5.B.III.4. Difference Calculation: We calculated the difference between the averages of target and standard stimuli to understand the distinct neural responses elicited by the target stimuli.

Listing 5: Difference Calculation

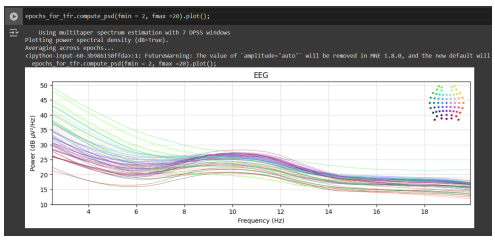
```
diff = mne.combine_evoked((
    target, -standard), weights="
    equal")
```

5.B.III.5. Difference Visualization: We used joint plots to visualize the differences between the neural responses to target and standard stimuli. We created image plots of the differences to examine the temporal and spatial dynamics of the brain's response to the sudden change in stimuli.



5.B.IV. Time-Frequency Analysis using Morlet Wavelets

5.B.IV.1. PSD Computation: We computed the Power Spectral Density (PSD) for the epochs to investigate the frequency-specific neural activity. We generated PSD plots to visualize the distribution of power across different frequency bands for both target and standard stimuli.



5.B.IV.2. Time-Frequency Decomposition: We applied Morlet wavelets to the epochs for a time-frequency analysis to study the oscillatory dynamics of neural activity in response to the stimuli.

Listing 6: Time-Frequency Decomposition

```
freqs = list(range(3, 30))
tfr_target = tfr_morlet(
    epochs_for_tfr["target"],
    freqs, 3, return_itc=False)
tfr_standard = tfr_morlet(
    epochs_for_tfr["standard"],
    freqs, 3, return_itc=False)
```

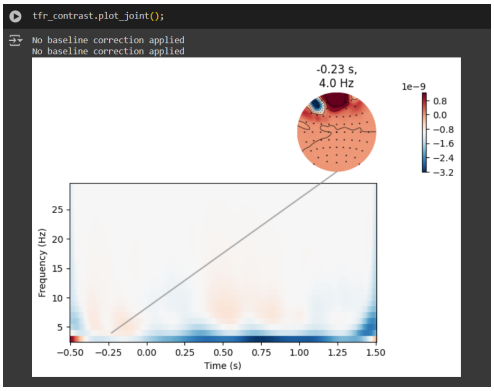
5.B.IV.3. Time-Frequency Visualization: We plotted the time-frequency representation for target stimuli to visualize how neural oscillations change over time and across frequencies.

5.B.IV.4. Time-Frequency Contrast Calc: We combined the time-frequency representations of target and standard stimuli to contrast their neural responses. We applied baseline correction to the combined time-frequency data to normalize the results.

Listing 7: Time-Frequency Contrast Calculation

```
tfr_contrast = mne.
    combine_evoked((tfr_target,
    tfr_standard), (-0.5, -0.5))
```

5.B.IV.5. Time-Frequency Contrast Visualization: We used joint plots to visualize the combined time-frequency representation, highlighting the differences in neural oscillations between the target and standard stimuli.



6. MAT to MNE

6.A. .mat Files

.mat files are data files used by MATLAB to store variables, arrays, and other types of data. They are often used for:

- Data Storage: Saving datasets, matrices, or other data structures in a compact format.
- Data Sharing: Easily sharing data between MATLAB users or importing or exporting data to/from other software.
- Interoperability: They can be read by other programming languages such as Python (using libraries like scipy.io).

6.B. SciPy Library

The scipy library is an open-source Python library used for scientific and technical computing. It builds on the numpy library, providing a large number of higher-level

functions that are useful for a variety of scientific and engineering applications.

Listing 8: SciPy Library

```
import scipy.io
# Reading a .mat file
data = scipy.io.loadmat('
   filename.mat')
# Writing to a .mat file
scipy.io.savemat('filename.mat',
    data)
```

- scipy.io.loadmat : This function loads the data from a .mat file into a Python dictionary, where the keys are the variable names and the values are the data arrays.
- scipy.io.savemat : used to save data to a .mat file, which can be read by MATLAB.

6.C. BCI Competition IV Dataset 1

Dataset 1 of the BCI Competition IV focuses on motor imagery tasks, where subjects imagine movements without any physical execution. There are continuous signals of 59 EEG channels and, for the calibration data, markers that indicate the time points of cue presentation and the corresponding target classes. Data are provided in Matlab format (*.mat) containing variables:

- cnt: The continuous EEG signals, size [time x channels]. The array is stored in datatype INT16. To convert it to uV values, use cnt= 0.1*double(cnt); in Matlab.
- mrk: Structure of target cue information with fields (the file of evaluation data does not contain this variable)
  - pos: Vector of positions of the cue in the EEG signals given in unit sample, length cues
  - y: Vector of target classes (-1 for class one or 1 for class two), length cues
- nfo: Structure providing additional information with fields
  - fs: Sampling rate,
  - clab: Cell array of channel labels,
  - classes: Cell array of the names of the motor imagery classes,
  - xpos: x-position of electrodes in a 2d-projection,
  - ypos: y-position of electrodes in a 2d-projection.

6.C.I. Loading the dataset :

Loads the dataset from the .mat file into data

Listing 9: Loading the dataset

```
data = sio.loadmat('mat2mne.
   mat') ## loading our data
    into data variable
```

6.C.II. Contents of the dataset :

The keys of the database represent the contents

Listing 10: Dataset contents

```
data.keys()
## Output : dict_keys(['
   __header__', '__version__
   ', '__globals__', 'mrk',
   'cnt', 'nfo'])
```

6.C.III. Retrieving EEG data :

Get the EEG data mapped with cnt and convert it into microvolts

Listing 11: EEG Data

```
cnt = data['cnt']
cnt_1 = 0.1 * cnt.astype(np.
   float64)  # converting
   the data into proper
   untis(to microvolts)
```

6.C.IV. Retrieving other details :

We find out the sampling rate *sampling_rate*, channels *c_lab*, the motor imagery classes *classes*

Listing 12: Other details

```
nfo = data['nfo']
sampling_rate = nfo['fs'][0,
    0][0, 0]
c_lab = [c[0] for c in nfo['
   clab'][0, 0][0]]
```

```
classes = [c[0] for c in nfo
    ['classes'][0, 0][0]]    #
    # defining and
    intializing few variables
     we'll need in future
```

6.C.V. Formation of Time Array :

We form the time array for plotting graphs

Listing 13: Time Array

```
time = np.arange(cnt_1.shape
    [0]) / sampling_rate
```

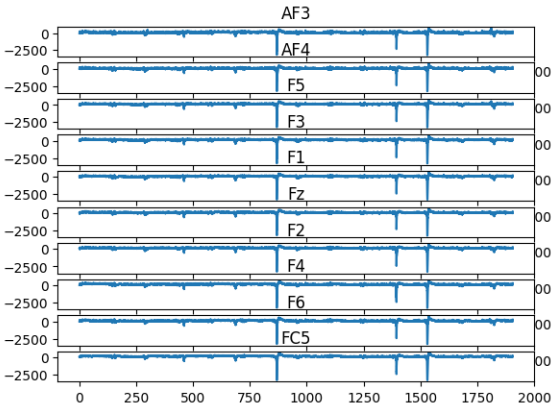
6.C.VI. Plotting EEG graph plots :

Plot the EEG data wrt the time array

Listing 14: EEG Graphs

```
for i in range(
    num_channels_to_plot):
plt.subplot(
    num_channels_to_plot, 1,
    i+1)
plt.plot(time, cnt_1[:, i])
plt.title(c_lab[i])
plt.tight_layout()
plt.show()  ## visualizing
    our dataa
```

Here, we choose the number of channels to be 10 and get the output as



This is the EEG data of 10 channels plotted with time

6.D. Application of MNE :

We need to create a mne info object for further processing of the imported EEG data. The Info object in MNE-Python is a container for metadata about an EEG/MEG dataset. It includes information such as:

- Channel names and types: Names of sensors (e.g., EEG, MEG).
- Sampling frequency: The rate at which data was collected.
- Measurement date: Date and time when the data was recorded.
- Montage: Electrode placement on the scalp.
- Bad channels: List of channels marked as bad.
- Projectors: SSP projectors used for noise reduction.

The Info object is essential for preprocessing and analyzing neuroimaging data, as it provides context for interpreting the signals.

Listing 15: Info Object Creation

```
info = mne.create_info(
    ch_names=c_lab, sfreq=
    sampling_rate, ch_types='
    eeg')  ##Creating our
    info object (to later get
     the raw data)
```

The mne.create_info creates the info object using the channel names list c_lab, the sampling frequency sampling_rate, and the type of data i.e. EEG

Currently, the info object doesn't have any digitized points or montage, i.e. it doesn't describe the position of the sensors and hence would give issues in ICA analysis plots.

6.D.I. Creation of Montage :

The DigMontage object represents a montage of digitized electrode positions used to define the spatial locations of electrodes on the scalp. This object is crucial for accurate source localization and for aligning the electrode positions with anatomical images.

- Electrode Positions: Stores the 3D coordinates of each electrode.
- Fiducial Points: Contains coordinates for anatomical landmarks.

- Transformations: Provides transformations to align electrode positions with anatomical images.
- Standard Montages: Can load standard electrode layouts.

The DigMontage object is integral for ensuring that the spatial configuration of electrodes is accurately represented, which is vital for subsequent analyses.
Retrieval of the coordinate point position of the sensors :

Listing 16: Finding Coordinates

```
xpos = nfo['xpos'][0, 0].
   flatten()
ypos = nfo['ypos'][0, 0].
   flatten()
```

Storing the positions in a numpy array to be used for montage creation :

Listing 17: Storing Positions

```
positions = np.vstack([xpos,
    ypos, np.zeros(len(c_lab
   ))]).T
```

Montage creation :
The mne.channels.make_dig_montage maps channel names with their positions and returns a montage of digitized points, which is then input into the info object created before.

Listing 18: Creating Montage

```
montage = mne.channels.
   make_dig_montage(ch_pos=
   dict(zip(c_lab, positions
   )))
info.set_montage(montage)
   ## including the position
    our electrodes in our
   info object
```

6.D.II. Creation of Raw object :

Listing 19: Creating Raw object

```
raw = mne.io.RawArray(cnt_1.
   T, info) ## creating our
   raw object after
   including the location of
    electrodes
```

We can create the Raw object using the mne.io.RawArray function which takes in the EEG data and the info object The Raw object in MNE-Python represents continuous EEG/MEG data. It includes:

- Data: Continuous time series data from sensors.
- Info: Metadata about the recording (channels, sampling frequency, etc.).
- Methods: Functions for data manipulation (filtering, resampling, etc.).

Once the raw object is made, we can construct plots, get PSD plots, filter the data, use ica to plot further graphs, and perform more processing on the data.

6.E. ICA

ICA(Independent Component Analysis) is a technique used for the separation of mixed signals into individual original signals that we get in the EEG data. We can think of it as an example in which we can separate the different voices in a piece of music.

In our project of Drowsiness Detection,ICA is used as a pre-processing step for EEG data analysis. EEG measures brain activity, and drowsiness can be reflected in changes in specific brainwave patterns. However, EEG signals are easily contaminated by other sources like eye movements or muscle activity. By using ICA to pre-process EEG data and extract relevant features, drowsiness detection systems can become more accurate and reliable.

Here is how ICA can help us :-

- Separating Sources: ICA can separate the EEG signal from these unwanted artifacts.
- Feature Extraction: Once the clean EEG signal is isolated, ICA can help identify specific features within the signal that correlate with drowsiness.

Listing 20: Creating ICA object

```
ica = mne.preprocessing.ICA(
   n_components=20,
   random_state= 0)
ica.fit(raw)  ##
   preprocessing using ica
```

6.E.I. Importing ICA:

This line imports the ICA functionality from the mne.preprocessing sub-library in Python.

6.E.II. Initializing ICA:

This line defines the parameters for the ICA object (ica):
n_components=20 : This specifies the number of independent sources(components) we want to extract from the EEG data.

The code snippet applies ICA to separate sources in your EEG data (stored in raw) which likely has 59 channels. ICA aims to decompose the mixed signals in each channel into 20 independent sources, potentially revealing brain activity related to drowsiness. The output informs you that 20 components were extracted and the process took 36.5 seconds. However, a warning suggests pre-processing the data with a high-pass filter before applying ICA might improve the results by removing slow-frequency noise.

Listing 21: ICA plots

```
ica.plots_components();
```

6.E.III. Plotting

This refers to the ICA object created. It holds the information about the ICA decomposition process applied to your EEG data.
plot_components(): This is a method built into the ICA object. It's specifically designed to generate a visual representation of the independent components extracted from the data. This function typically creates a set of subplots, one for each extracted component (up to 20 in our case).
Each subplot might display: A topography which is a head map-like image representing the scalp distribution of the component. This indicates where on the scalp the brain activity associated with that component is most prominent. Each subplot might contain a head map showing the distribution of activity for a specific component across the scalp. Darker/brighter colors indicate stronger activity in those regions.

Additionally, some subplots might include time series graphs representing how the component's activity changes over time during the EEG recording. By analyzing these scalp distributions and potentially time series, we can gain insights into which brain regions are most involved in the activities captured by each component, potentially revealing drowsiness-related brain activity patterns.

Listing 22: idx

```
ica.plot_properties(raw,idx)]
```

Component Indices: This line defines a list named idx containing the values [0, 1, 2, 3, 4, 5]. These indices likely represent the specific independent components you want to investigate further.
Plotting Properties: This line calls the plot_properties function from the ICA object. This function is used to visualize various properties of the
ICA components specified in the idx list.

6.E.IV. Subplots

Subplots: The image likely consists of multiple subplots arranged in a grid, with one subplot for each component index in your idx list .
Power Spectrum: Each subplot might display a power spectrum. This is a graph that shows how the power (amplitude) of the component's activity is distributed across different frequencies. By looking at the power spectrum, you can see which frequency bands have the most activity for that particular component.

Listing 23: Removing Bad Components

```
ica.exclude = bad_idx   ###
   removing some of the bad
   components
ica.apply(raw)
```

6.E.V. Removing Bad Components and Applying ICA

Removing Bad Components: This line assumes we have a list named bad idx containing indices of the ICA components we identified as bad or irrelevant to your analysis (e.g., components resulting from eye movements or muscle activity).
Here, we're assigning the bad index list to the

exclude attribute of the ICA object.
This tells the ICA object to exclude those specific components during further processing.
Applying ICA: This line calls the apply method of the ica object.
This method applies the ICA decomposition (excluding the components marked for exclusion) to the original EEG data stored in the raw variable.
The ICA transform essentially separates the mixed signals in each channel into the independent components you previously extracted. By applying ICA, we project the data into the ICA space defined by these components.

6.E.VI. Visualising after using ICA

Listing 24: Visualising after using ICA

```
raw.plot(duration=100.0,
   scalings={'eeg': 100});
```

Time Series: The x-axis represents time (in seconds), showing a 100-second snippet of your EEG recording after applying ICA.
EEG Channels: The y-axis likely represents voltage (scaled by a factor of 100 according to your code), with each line in the plot corresponding to a single EEG channel. The channel numbers or labels might be displayed on the right side of the plot for reference.
Brain Activity: The lines in the plot represent the electrical activity recorded from different scalp locations (electrodes) over time. After applying ICA, the activity in each channel reflects a specific independent component extracted from the data.

6.E.VII. Finding Events

Listing 25: Finding Events

```
 event_id = {'first_type': -1, '
    second_type': 1}
y = data['mrk']['y'][0,0].
   flatten()
pos = data['mrk']['pos'][0,0].
   flatten()
events = np.column_stack((pos,
   np.zeros_like(pos), y))
events
```

Event Labeling: event id is a dictionary that assigns labels to different event types, with -1 for 'first type' and 1 for 'second type'.
Extracting Event Labels: y extracts event labels from the data by accessing the 'mrk' dictionary within data, and flattens the 2D array at key 'y' into a 1D array.
Extracting Event Positions: pos extracts event positions from the data in a similar manner, flattening the 2D array at key' pos' into a 1D array.
Creating Events Array: events is a 2D array created by stacking three columns: event positions (pos), a column of zeros, and event labels (y). Each row in the events array represents a single event.

6.E.VIII. Creating Epochs

Listing 26: Creating Epochs

```
epochs = mne.Epochs(raw, events,
    event_id=event_id, tmin
   =-0.5, tmax=2, baseline=None,
    preload=True)
```

This line of code creates an object of the Epochs class from the MNE library, which is used to segment your continuous EEG data (raw) into time epochs around specific events (events). This code takes your raw EEG data (raw), event information (events), and time window specifications (tmin and tmax) to create epochs. Each epoch is a segment of your continuous data centered around an event of a specific type (defined by event_id).
The epochs might be further processed and analyzed to study event-related brain activity.

Listing 27: removing the bad index and visualizing the epoched data

```
epochs = ica.apply(epochs,
   exclude=ica.exclude)
epochs.plot(n_channels=10,
   scalings={'eeg': 200});
```

This code first applies the previously computed ICA decomposition to remove bad components from your epoched data. Then, it visualizes the resulting epochs, allowing you to inspect the activity in selected EEG channels around the events of interest.

Applying ICA to epochs (removing bad components):
epochs = ica.apply(epochs, exclude=ica.exclude)
Visualizing the epoched data:
epochs.plot(n_channels=10, scalings='eeg': 200);
Epochs: The x-axis likely represents time (in seconds), with each horizontal section corresponding to an epoch (time window around an event).
EEG Channels: The y-axis likely represents voltage (scaled by a factor of 200 according to your code), with each line in the plot corresponding to activity in a specific EEG channel
Event-Related Activity: The lines in each epoch represent the electrical activity recorded from the scalp electrodes for that channel after ICA was applied to remove bad components.

Listing 28: plotting the evoked data

```
epochs.average().plot(
   spatial_colors=True);
```

Epochs.average(): This calculates the average brain activity across all epochs (trials) for each time point and channel.The result is an Evoked object representing the typical response to the events.
Plot(spatial_colors=True): This plots the average response, with different colors for different EEG channels.

,

```
diff = mne.combine_evoked((first
   ,-second),weights='equal')
diff.plot_joint(times
   =[0.22,.35,0.38,0.46]);
```

epochs.average(): This calculates the average brain activity across all epochs (trials) for each time point and channel.The result is an Evoked object representing the typical response to the events.
plot(spatial_colors=True): This plots the average response,
with different colors for different EEG channels.

7. Drowsiness detection using Interpretable CNN

We reviewed the research paper published by Cui Jian et al, which proposed a novel CNN named "Interpretable CNN". Here's what we learned.

7.A. Why drowsiness detection?

Driver drowsiness, which significantly impairs reaction times and decision-making abilities, is the leading cause of vehicular accidents.

As a solution, a drowsiness detection system is designed to alert the driver when drowsiness is detected. Non-EEG solutions generally involve monitoring the subject's face, lane-tracking performance, steering pattern, face temperature, heart rate, and skin conductivity. Using EEG is advantageous over these as it directly monitors brain activities with a high temporal resolution, detecting drowsiness in early stages before it is reflected in the subject's behavior.

7.B. An innovative enhancement

Conventional methods to detect drowsiness using EEG signals rely on studying hand-crafted features that might exclude other relevant information. Deep learning eliminates this need for prior feature crafting.

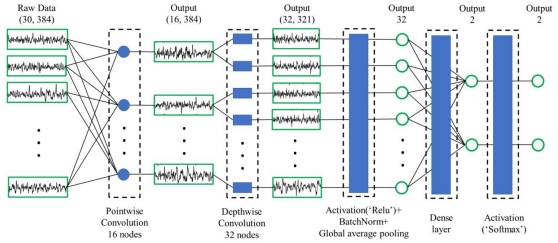
Moreover, deep learning models are typically seen as black-box classifiers. To improve this, an "InterpretableCNN" has been developed by Cui Jian et al, which is a compact CNN model that can provide explanations for its decisions. Interpretation of decisions allows us to find out whether the classification is driven by relevant features as well as discover new neurophysiological patterns.

7.C. Description of the dataset

EEG data from 27 subjects was recorded during a sustained-driving task in a virtual reality simulator. During the process, random lane-departure events were introduced, causing the car to drift from the central lane and participants had to respond quickly by steering the car back to the central lane. The speed of their responses served as an indicator of their drowsiness level.

After processing and balancing the data, the labeled dataset comprised 2022 samples, each 3 seconds long, from 30 channels at a sampling rate of 128Hz.

7.D. Network structure



The network comprises 7 layers as shown above. The first two layers perform pointwise and depthwise convolutions, respectively. These are followed by a ReLU activation layer, a batch normalization layer, a global average pooling layer, a dense layer, and a softmax activation layer. The output layer has two nodes to classify the data as either 0 ("alert") or 1 ("drowsy").

7.D.I. Pointwise convolutional layer

The first layer contains 16 nodes, each of which is connected to every node (every channel) in the input layer. This layer aims to spatially filter the data by creating linear combinations of the input EEG signals. The weights serve as spatial filters. By having 16 nodes, which is roughly half the number of input channels, redundancy is reduced, and network convergence is encouraged.

$$h_{i,j}^{(1)} = \sum_{p=1}^{m} w_{i,p}^{(1)} x_{p,j}^{(1)} + b_i^{(1)}$$

7.D.II. Depthwise convolutional layer

This layer uses depthwise convolutions to extract features. Every signal of the previous layer is convoluted with different kernels of same length at two nodes of this layer. the length of a kernel is set as l = 64, which is half of the sampling rate (128 Hz). The size of the output $h_{i,j}^{(2)}$ is (2N1, n-l+1), which is (32, 321).

$$h_{i,j}^{(2)} = \begin{cases} \sum_{r=1}^{l} h_{\frac{i+1}{2},j+r-1}^{(1)} w_{i,r}^{(2)} + b_i^{(2)}, \textit{when i is odd.} \\ \sum_{r=1}^{l} h_{\frac{i}{2},j+r-1}^{(1)} w_{i,r}^{(2)} + b_i^{(2)}, \textit{when i is even.} \end{cases}$$

7.D.III. Global average pooling (GAP) layer

The GAP layer significantly reduces the number of parameters in the neural network, which helps in effectively preventing overfitting. It averages the output of the previous layer depthwise.

$$h_i^{(5)} = \frac{\left(\sum_{j=1}^{n-l+1} h_{i,j}^{(4)}\right)}{n-l+1}$$

7.E. Interpretation technique

The model is called "InterpretableCNN" because its classification results can be explained using the technique explained as follows, giving it an edge over other cutting-edge deep learning models.

The Class Activation Map (CAM) method is an effective interpretive technique that can identify the important regions of each input sample. For each input sample, a heatmap is generated from the activations of the last convolutional layer. This heatmap is then resized to match the input sample's dimensions, highlighting the extent to which different local regions contribute to the classification. Furthermore, due to the complexity of the model, the CAM method is slightly modified to better suit its requirements.

Suppose a given input EEG sample X(m,n) is classified with label c, where c is either 0 (alert) or 1 (drowsy). The goal is to generate a heatmap C(m,n) for X(m,n) that highlights important regions for the network's prediction. The input signal X(m,n) generates activation $h_c^{(6)}$ in the 6th layer:

$$h_c^{(6)} = \sum_{i=1}^{2N_1} w_{i,c}^{(6)} h_i^{(5)} = \sum_{i=1}^{2N_1} \sum_{j=1}^{n-l+1} w_{i,c}^{(6)} h_{i,j}^{(4)} = \sum_{i=1}^{2N_1} \sum_{j=1}^{n-l+1} M_{i,j}^{c} ,$$

where

$$M_{i,j}^{c} = w_{i,c}^{(6)} h_{i,j}^{(4)}$$

This map $M^c{}_{i,j}$ is viewed as the distribution of $h_c^{(6)}$ for class c in a map of size 2N1*(n-l+1).

The original CAM method upsamples $M^c{}_{i,j}$ to match the input sample size, but it cannot be directly used here due to channel mixing by pointwise convolutions in the first layer. Instead, inspired by the CNN-Fixation method, the most contributive positions in $M^c{}_{i,j}$ is traced back to their corresponding areas in the input signal.

The $M^c{}_{i,j}$ values are ranked, and the top N positions (i_k, j_k) are traced to corresponding locations (p_k, q_k) in the input. The final

heatmap is obtained by:

$$S^c_{p,q} = \frac{1}{\sigma\sqrt{2\pi}} \sum_{k\ for\ p_k=p} e^{\left(-\frac{1}{2}\frac{(q-q_k)^2}{\sigma^2}\right)}$$

Finally, these discriminative locations are traced through the network layers. Discriminative locations remain unchanged after the 3rd and 4th layers. Through the depthwise and pointwise convolutional layers, $h_{ik,jk}^{(2)}$ is generated from the local area in the input as:

$$\begin{aligned} h^{(2)}_{i_k,j_k} &= \sum_{r=1}^{l} h^{(1)}_{\frac{i_k+1}{2},j_k+r-1} w^{(2)}_{i_k,r} \\ &= \sum_{r=1}^{l} w^{(2)}_{i_k,r} \sum_{p=1}^{m} w^{(1)}_{\frac{i_k+1}{2},p} x_{p,j_k+r-1} \\ &= \sum_{p=1}^{m} w^{(1)}_{\frac{i_k+1}{2},p} \sum_{r=1}^{l} w^{(2)}_{i_k,r} x_{p,j_k+r-1}, \end{aligned}$$

when i_k is odd, and

$$h^{(2)}_{i_k,j_k} = \sum_{p=1}^{m} w^{(1)}_{\frac{i_k}{2},p} \sum_{r=1}^{l} w^{(2)}_{i_k,r} x_{p,j_k+r-1}$$

when i_k is even. This shows how $h_{ik,jk}^{(2)}$ is generated from the weighted sum of convoluted signals in the input's local area. It is observed that $h_{ik,jk}^{(2)}$ is generated from a segment of the input signals in the local area from time point j_k to $j_k + l - 1$. It is the weighted sum of the convoluted signals across all m channels, with the weight assigned to channel p being $w_{(ik+1)/2,p}^{(1)}$ or $w_{ik/2,p}^{(1)}$.

Therefore, the discriminative location i_k, j_k in $M^c_{i,j}$ can be traced back to the center (p_k, q_k) of the strongest contributing segment in the input signal, where:

$$h^{(2)}_{i_k,j_k} = \sum_{p=1}^{m} w^{(1)}_{\frac{i_k}{2},p} \sum_{r=1}^{l} w^{(2)}_{i_k,r} x_{p,j_k+r-1}$$

σ is set as $l/2 = 32$ so that the discriminative location in the input signal will highlight the entire segment of the strongest contributing signal. The top 100 (N=100) discriminative locations in the class activation map $M^c_{i,j}$ are traced, which accounts for around 1% of all entries.

7.F. Interpretation on the learned characteristics of EEG

Understanding what the model has learned from the data is crucial for validation. Here, we list the patterns the model uses to distinguish between alert and drowsy EEG signals.

The model has learned specific characteristics to distinguish between alertness and drowsiness in EEG signals. Drowsiness is identified by a high portion of Theta or Alpha waves, rhythmic bursts in the Theta band (referred to as "drowsy bursts"), and spindle-like structures in the Alpha frequency or a narrow frequency peak within the Alpha band, which are indicators of early drowsiness. The central and centro-parietal EEG channels also play a more significant role in identifying drowsiness.

On the other hand, alertness is characterized by more artifacts due to eye movement, which is a strong indicator of a wakeful state, high power of Beta and Delta waves (also caused by eye movements), and large voltage changes in signals from the frontal EEG channels.

7.G. Conclusion

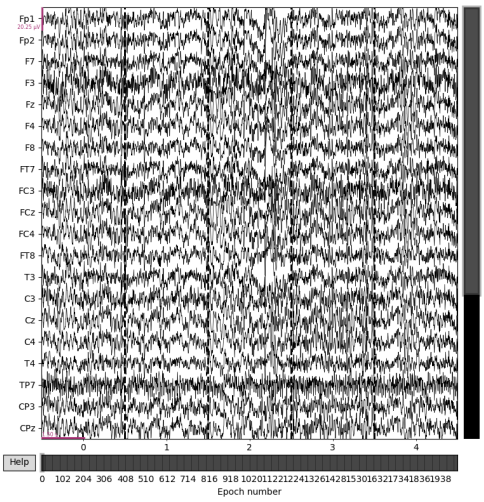
Reviewing the research paper by Cui Jian et al. gave us valuable insights into how "InterpretableCNN" outperforms other deep learning models. Its ability to provide explanations for its decisions allows us to determine if the classification is based on relevant features and uncover new neurophysiological patterns.

8. Implemenation

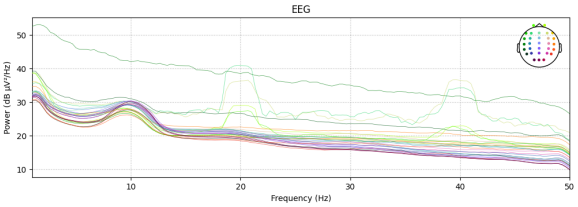
We executed the paper in four sections, in order to simplify the process.

8.A. Dataset Analysis

Using the processed dataset[25], we first attempted to visualize the EEG data using MNE. The dataset comprises of 2022 epochs of EEG data from 30 channels, recorded at a sampling rate of 128Hz, equally distributed and labelled as "drowsy" and "alert". The EEG plot of the first five epochs are shown below.



The power spectral density plot allowed us to visualize the power present in the signal across different frequencies.



Furthermore, we created an evoked object of the "alert" and "drowsy" epochs separately, aiming to obtain a smoother signal by averaging the epochs. The plots of the evoked objects are shown below.

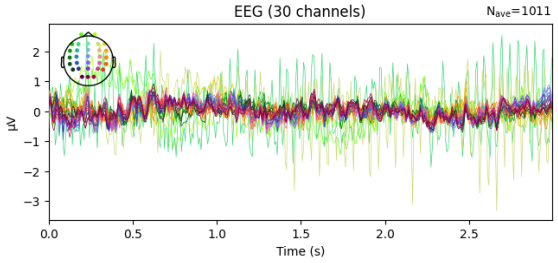


Fig: Evoked EEG plot of "alert" signals

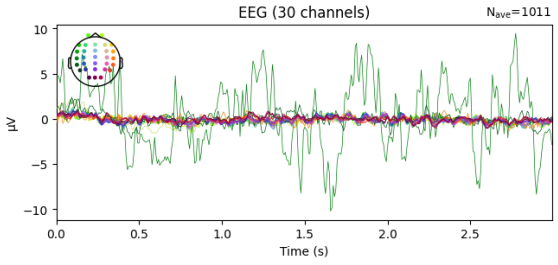


Fig: Evoked EEG plot of "drowsy" signals

Moreover, we computed the time frequency representation of both classes using morlet wavelets. The TFRs of the signal received by the Fz electrode are as shown.

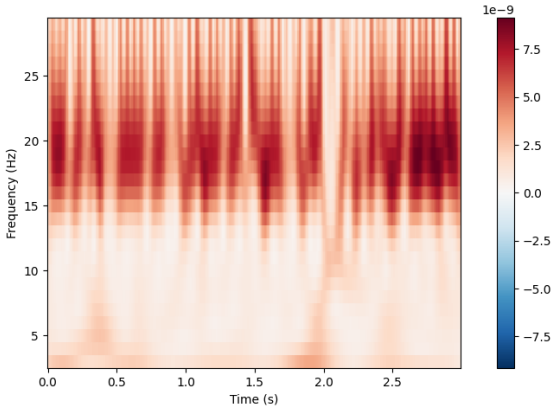


Fig: TFR plot of "alert" EEG

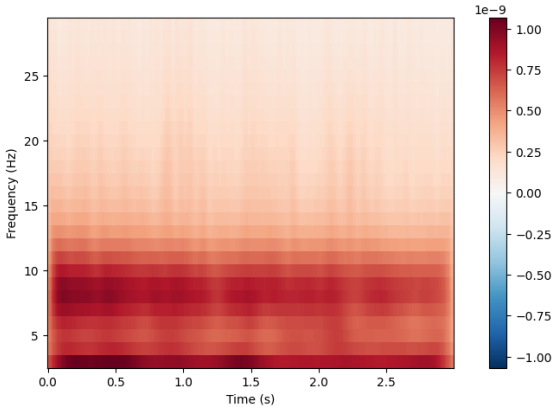


Fig: TFR plot of "drowsy" EEG

8.B. The Interpretable CNN Model

As mentioned in the paper, we tried to implement the "InterpretableCNN".
The model uses a combination of pointwise and depthwise convolutions, which is a technique borrowed from MobileNets. This structure reduces the number of parameters and computational cost compared to traditional convolution layers. Pointwise convolution learns cross-channel correlations, while depthwise convolution captures spatial patterns independently for each channel.

Listing 29: Creation of Interpretable CNN class

```
class InterpretableCNN(torch.nn.
    Module):
```

We create our own PyTorch class named **InterpretableCNN** that inherits from the torch.cnn.Module class. The torch.nn.Module class is the base class for all neural network modules in PyTorch. It provides essential functionalities like defining layers, forward pass,

backward pass, and parameter management.

Listing 30: Initialisation function of InterpretableCNN

```
def __init__(self, classes=2,
    sampleChannel=30,
    sampleLength=384 ,N1=16, d=2,
    kernelLength=64):
     super(InterpretableCNN, self
        ).__init__()
     self.pointwise = torch.nn.
        Conv2d(1,N1,(
        sampleChannel,1))
     self.depthwise = torch.nn.
        Conv2d(N1,d*N1,(1,
        kernelLength),groups=N1)
     self.activ=torch.nn.ReLU()
     self.batchnorm = torch.nn.
        BatchNorm2d(d*N1,
        track_running_stats=False
        )
     self.GAP=torch.nn.AvgPool2d
        ((1, sampleLength-
        kernelLength+1))
     self.fc = torch.nn.Linear(d*
        N1, classes)
     self.softmax=torch.nn.
        LogSoftmax(dim=1)
```

For the initialisation function we have parameters necessary to create the model such as classes, sample channels, sample length, N1 i.e. the number of nodes in the first pointwise layer, d i.e. the number of kernels for each new signal in the second depthwise layer, and the length of convolutional kernel in the depthwise layer, all their default values set according to our model.
super() is a built-in function in Python used to access methods of the parent class. In this context, *super(InterpretableCNN, self).__init__()* calls the initialization function of the parent class and passes self (the current object instance) as an argument. This ensures that the initialization process of the parent class is executed before the InterpretableCNN class performs its own initialization logic.
The Network Layers class defines several layers using PyTorch's built in libraries.
self.pointwise creates a 2D convolutional layer with 1 input channel (assuming your data has 1 channel), N1 output channels, a kernel size of (sampleChannel, 1) which performs a pointwise convolution along the channel dimension.
self.depthwise creates a depthwise separable convolutional layer. It has N1 input channels, d*N1 output channels, a kernel size of (1, kernelLength), and uses group convolution with the number of groups set to N1. This essentially applies N1 different filters of size (1, kernelLength) to the input independently.
self.activ defines a ReLU activation function for introducing non-linearity.
self.batchnorm creates a batch normalization layer with d*N1 channels. For our model the *track_running_stats* argument is set to False, meaning the layer won't update its statistics during training, setting running mean and variance to none.
self.GAP creates a global average pooling layer that takes the average over the entire width dimension (temporal dimension) of the feature maps. The specific size ensures the entire remaining width is considered after the depthwise convolution.
self.fc defines a fully-connected layer with d*N1 input units (number of channels after depthwise separable convolution and pooling) and classes output units (number of classes to be predicted).
self.softmax defines a softmax layer applied across the dimension 1 (classes) to produce class probabilities. Here we use the *LogSoftmax* which is more stable than standard Softmax.

Listing 31: Forward Pass function

```
def forward(self,inputdata):
    x=self.pointwise(inputdata)
    x=self.depthwise(x)
    x=self.activ(x)
    x=self.batchnorm(x)
    x=self.GAP(x)
    x=x.view(x.size()[0],-1)
    x=self.fc(x)
    x=self.softmax(x)
    return x
```

The *forward* function defines the computation process for a single data point passing through the network. It applies the defined layers sequentially, performs necessary reshaping, and finally returns the class probabilities.

8.C. Visualisation Techniques

We defined the **VisTech** class to create visualizations that effectively highlight the most discriminative features in the EEG data used by the CNN model to classify the state of alertness (alert or drowsy). The heatmaps generated provide insights into which parts of the EEG signals were most influential in the model's decision, helping in understanding and interpreting the model's behavior. Let's see how this works:

The *heatmap_calculation* function first passes the input batch of EEG data (*batchInput*) through the initial layers of the CNN model up to the fourth layer. This is done to obtain the intermediate activations. ,

```
batchActiv1 = self.model.
   pointwise(batchInput)
batchActiv2 = self.model.
   depthwise(batchActiv1)
batchActiv3 = self.model.activ(
   batchActiv2)
batchActiv4 = self.model.
   batchnorm(batchActiv3)
```

We then extract the weights of the layers involved in the model. These weights are necessary to understand how the model combines different features to make its predictions.

We then introduce a *Class Activation Map (CAM)* to highlight the regions of the input data that are important for the classification decision. This is done by combining the activations from the final convolutional layer (*batchActiv4*) with the weights of the final fully connected layer (*layer6weights*). ,

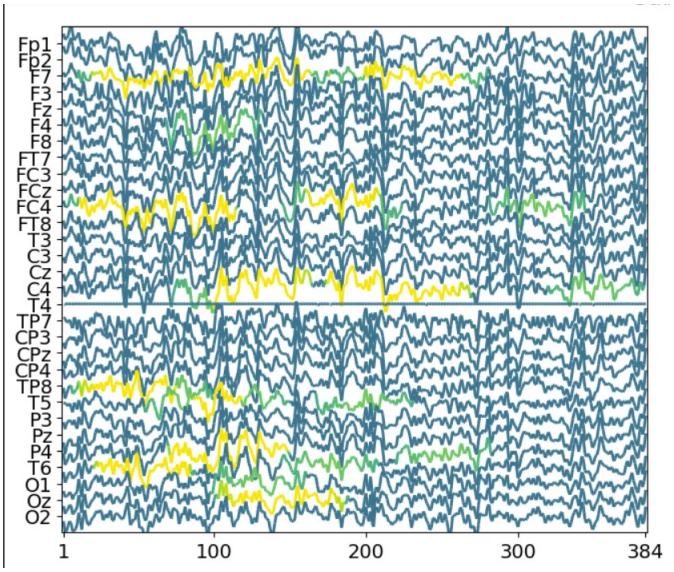
```
CAM=sampleActiv4*np.tile(np.
   array([layer6weights[:,state
   ]]).transpose(),(1,
   sampleActiv4.shape[1]))
CAMsorted = np.sort(CAM, axis=
   None)
CAMthres = CAMsorted[-100]
```

A fixation map is created to identify the locations of the most discriminative points based on the CAM threshold. We then map these discriminative locations back to the original input space. This involves reversing the operations of the convolutional layers to find out which parts of the input data contributed to the discriminative features.

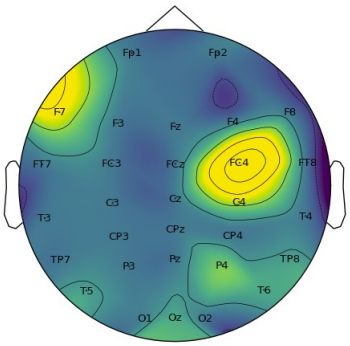
A heatmap is generated using a Gaussian function to smooth the discriminative points. This heatmap visually highlights the regions of the input data that were most influential in the classification decision. ,

```
heatmap[p,qk]=heatmap[p,qk]+ 1/
   radius/np.sqrt(2 * np.pi) *
   np.exp(-(qk-q)** 2/(2*radius
   *radius))
```

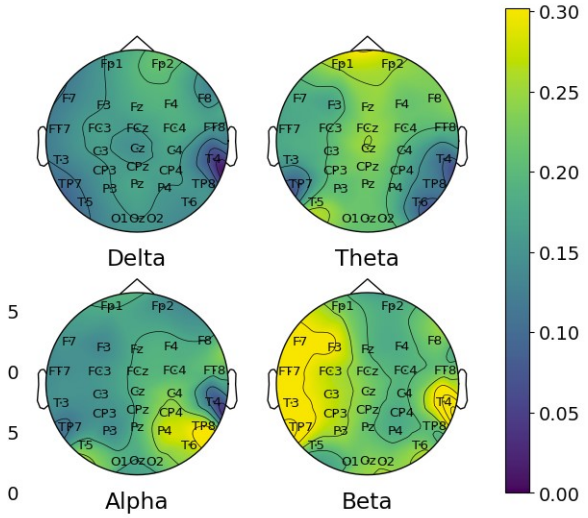
The *generate_heatmap* function calls *heatmap_calculation* to generate the heatmap and then creates various plots to visualize the EEG data along with the heatmap. This includes plotting the raw EEG signals, the heatmap overlaid on the signals, and topographic maps showing the distribution of power across different frequency bands (delta, theta, alpha, beta).



EEG data Visualisation



Heatmap Topograph



Power distribution across frequency bands

The images attached are from the EEG with index no. 30, with P_alert= 0.75.

8.D. Leave-One-Out Cross Validation

Leave-One-Out Cross-Validation (LOOCV) is a technique used to evaluate the performance of a machine learning model. It is a special case of k-fold cross-validation where k is equal to the number of samples in the dataset. In LOOCV, each sample in the dataset is used once as a test set while the remaining samples are used as the training set. This process is repeated for each sample, resulting in a total of n iterations (where n is the number of samples).

LOOCV provides an almost unbiased estimate of the model's performance because it uses nearly all data points for training in each iteration. This reduces the bias that can arise from insufficient training data.

Since each sample is used as a test set exactly once, LOOCV makes maximal use of the available data for training the model. This is particularly beneficial for small datasets where preserving as much training data as possible is crucial.

Our code effectively demonstrates the application of leave-one-subject-out cross-validation to evaluate the performance of a CNN model on EEG data. **We achieved an accuracy of 77% indicating the model's performance across different subjects, showcasing its generalizability and robustness.**

8.E. Conclusion

In leveraging the InterpretableCNN model designed for EEG data processing and employing leave-one-subject-out cross-validation (LOOCV), this study advances drowsiness detection systems by directly monitoring brain activity with high temporal resolution, surpassing non-EEG methods. LOOCV ensures an unbiased assessment of the model's performance across subjects, highlighting its robustness in distinguishing between alert and drowsy states with a notable 77% accuracy. Visualizations like heatmaps further elucidate the model's interpretability by revealing key EEG features driving classification decisions. This integrated approach not only enhances understanding of neural correlates of drowsiness but also underscores its potential in real-world applications for proactive driver safety interventions.

9. References

References

[1] 3Blue1Brown, *Neural Networks Series*, YouTube, https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi.

[2] 3Blue1Brown, *But what is a convolution?*, YouTube, https://youtu.be/KuXjwB4LzSA?feature=shared.

[3] Dr. Harish Garg, *Regression Analysis, Models, Lines Coefficients*, YouTube, https://youtu.be/PyzZBQKaj10?si=HNuModV7LJBD67nV.

[4] StatQuest with Josh Starmer, *Gradient Descent, Step-by-Step*, YouTube, https://youtu.be/sDv4f4s2SB8?si=c-QB2FwD6aIcwJiC.

[5] StatQuest with Josh Starmer, *Stochastic Gradient Descent, Clearly Explained!!!*, YouTube, https://youtu.be/vMh0zPT0tLI?si=zWfQYwcEgwjaLKej.

[6] Jason Yosinski, *Deep Visualization Toolbox*, YouTube, https://www.youtube.com/watch?v=AgkfIQ4IGaM.

[7] Brajesh Kumar, *Convolutional Neural Networks: A Brief History of Their Evolution*, Medium, https://medium.com/appyhigh-technology-blog/convolutional-neural-networks-a-brief-history-of-their-evolution-ee3405568597.

[8] Kunihiko Fukushima, *Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position*, Biological Cybernetics, vol. 36, no. 4, pp. 193–202, 1980, Springer.

[9] GeeksforGeeks, *CNN | Introduction to Pooling Layer*, https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/.

[10] Davis Neurotech, *Neurotech Crash Course*, YouTube, https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi.

[11] Jag Singh, *Module 4: Introduction to EEG and Neural Signal Processing*, YouTube, https://www.youtube.com/watch?v=cMwOvDhwrgA.

[12] Learning EEG, *Terminology and Waveforms*, https://www.learningeeg.com/terminology-and-waveforms.

[13] Brain-Computer Interfaces (EEG, MEG ...), *MNE Python*, YouTube, https://www.youtube.com/playlist?list=PLXtvZiGkmNVvPS0N9UNBVkIFe0_0t_Nqt.

[14] MNE-Python, *MNE-Python Tutorials*, https://mne.tools/stable/auto_tutorials/index.html.

[15] EEGLab, *Time-Frequency Analysis of EEG Time Series Part 3: Wavelet Transforms*, YouTube, https://www.youtube.com/watch?v=eUFf5eFpdLg.

[16] Younes Subhi, Analytics Vidhya, *Classifying Brain Signals for Brain-Computer Interfaces: A Short Introduction*, Medium, https://medium.com/analytics-vidhya/classifying-brain-signals-for-brain-computer-interfaces-a-short-introduction-53fe70486653.

[17] Neurotist, *Importing MATLAB Files into Python: A Step-by-Step Guide for EEG Data Analysis with MNE*, Medium, https://medium.com/@neurotist/importing-matlab-files-into-python-a-step-by-step-guide-for-eeg-data-analysis-with-mne-d6454a07c066.

[18] Brain-Computer Interfaces (EEG, MEG ...), *How to import EEG Matlab data into MNE-Python*, YouTube, https://www.youtube.com/watch?v=BbCUMgFJdsw.

[19] Jian Cui, Zirui Lan, Olga Sourina, Wolfgang Muller-Wittig, *EEG-based cross-subject driver drowsiness recognition with an interpretable convolutional neural network*, 2022, https://dr.ntu.edu.sg/bitstream/10356/156069/2/FINAL%20VERSION.pdf.

[20] Ross Brancati, Plain English AI, *Leave-One-Subject-Out Cross-Validation for Machine Learning Model*, https://ai.plainenglish.io/leave-one-subject-out-cross-validation-for-machine-learning-model-557a09c7891d.

[21] Kunjan, S., et al. (2021). *The Necessity of Leave One Subject Out (LOSO) Cross Validation for EEG Disease Diagnosis*. In: Mahmud, M., Kaiser, M.S., Vassanelli, S., Dai, Q., Zhong, N. (eds), *Brain Informatics*, BI 2021. Lecture Notes in Computer Science, vol. 12960, Springer, Cham, https://doi.org/10.1007/978-3-030-86993-9_50.

[22] Mat2MNE Dataset Description, *Dataset Description*, https://bbci.de/competition/iv/desc_1.html.

[23] Mat2MNE Dataset, https://drive.google.com/file/d/1gPyVLDmxBwmf4u086eNf5sKAij_J6C-y/view?usp=sharing.

[24] Oddball Experiment Dataset, https://drive.google.com/file/d/18uXP1tjoG1_i_LXaNwfeYq0HiEXE8MHk/view.

[25] EEG Driver Drowsiness Dataset, https://figshare.com/articles/dataset/EEG_driver_drowsiness_dataset/14273687.