Comparing Mental Arithmetic Task Count Quality Subjects Using Visual and Neuromorphic Computing Analysis

Catherine Donner
CS 5723 Neural Data Science Project

***Abstract*—When doing mental arithmetic tasks, there are individuals who tend to efficiently execute several arithmetic operations in a certain amount of time, while others can only execute a significantly lower amount of operations in the same amount of time. This can lead to an indication that such individuals in the latter may have a learning disability or disorder that may need to be diagnosed, or that they tend to not function as well with an non-optimized learning style. Data science as a field can help effectively classify individuals who have a good or bad count quality performance, and this study attempted to contribute a perspective on differentiating these count quality classes using neuromorphic computing through spiking neural networks (SNNs) and visual analysis with before and during task EEG signal data. The results from this study showed that some visual analysis comparisons revealed substantial differences between the count quality subjects, and that a trained SNN was able to achieve an accuracy of up to 71-72%. This study still contributes certain technical impacts including the novel use of SNNs on the dataset in this study, and since very few previous studies utilized analysis of the count quality groups, this study contributes an additional data-centered perspective focusing on these groups.**

I. Introduction

The research problem to be addressed in this project is whether there are significant differences in the EEG signals of good count quality individuals and bad count quality individuals in the EEG During Mental Arithmetic Tasks dataset [3] through visual and neuromorphic computing analysis. This dataset has 24 subjects that are classified as performing good quality count (mean number of operations per minute = 21, standard deviation = 7.4) and 12 subjects that are classified as performing bad quality count (mean number of operations = 7, standard deviation = 3.6) when doing the arithmetic tasks [8]. Although there will be a class imbalance in the dataset [3] for this project, it is important to address this research objective because neuromorphic computing through spiking neural networks (SNNs) is very instrumental in capturing the spatiotemporal qualities of EEG signals, and data visualization can also help illustrate patterns among the 2 classes. The SNN model was also chosen for this project because its spiking activation function is similar to that of biological neurons in the brain whereas regular artificial neural networks (ANNs) use non-linear continuous activation functions [6].

For real-world implications of this study, a study by Hellstrand et al. (2024) asserted that mathematical learning disabilities (MLD) rely on arithmetic fluency, which is defined as the ability to perform arithmetic operations accurately, effortlessly, and quickly, and this ability can be assessed through short, timed arithmetic tasks [4]. Reliance on counting-based strategies, which the study collecting the mental arithmetic task EEG data discouraged against [8], is a deficit that is common in individuals with MLD [4]. It is likely the subjects who are classified as having a bad count quality performance may have a learning disability that may need to be diagnosed, thus emphasizing the practical importance and societal implication of this study.

Previously, a study by Salankar, Koundal, & Qaisar (2021) utilized the 2 count quality groups in the dataset [3] for stress classification using a variational mode decomposition (VMD) method and machine learning (support vector machine and multilayer perceptron) [7]. The VMD approach deconstructed the EEG signals allowing for the relevant features of each signal to be extracted; the results showed that the first several variable modes led to better stress classification accuracy of up to 100% [7]. Although no limitations were noted in this study, future work proposed further classification of brain diseases including Alzheimer's and depression, and a need for real-time monitoring [7].

Using the same dataset [3], a study by Catrambone & Valenza (2023) again used the 2 count quality groups. The fuzzy entropy values of the good count and bad count quality groups were compared through visual analysis using the before and after task data [2]. This study showed that the good count quality group tended to have a higher brain-heart interplay (BHI) complexity in the direction from the brain to the heart as well as lower levels of stress during tasks [2]. The main limitations of this study included that the parameter space exploring BHI was limited and not optimized [2].

Although SNNs have not been utilized in studies using the EEG During Mental Arithmetic Tasks dataset, SNNs have been used before in studies analyzing EEG data. In a study by Antelis & Falcón (2020), an SNN was used for classifying motor imagery tasks, and this model achieved an accuracy of 74.54% [1]. Another study by Luo et al. (2020) used SNNs for emotion classification to train on 2 benchmark EEG datasets (DEAP and SEED) [5]. With the SNN, the emotion states from the DEAP dataset achieved a classification accuracy ranging from 74% to 86.27%, and an accuracy of 96.67% was achieved for the SEED dataset; this study outperformed baseline processing methods like fast Fourier transform (FFT)

and discrete wavelength (DWT) [5].

Research gaps in the literature review include that there have been some studies analyzing the differences between the count quality groups in the dataset [3], however more may be needed to contribute more diverse data-centered perspectives. SNNs have especially not been utilized for machine learning analysis involving the dataset. Likewise, a wider diversity of visual analysis techniques can be utilized to differentiate the 2 count quality classes.

II. Methods

The EEG During Mental Arithmetic Tasks dataset can be found for open-source use on PhysioNet. This dataset has been preprocessed, including high-pass filtering at 30 Hz, notch filtering at 50 Hz, and independent component analysis to eliminate biological noise artifacts (i.e., eyes, muscle, and cardiac) [3]. For context, the assigned arithmetic task to subjects was serial subtraction of two numbers without speaking or using hand movements, only with mental effort [8]. The EEG data is dispersed around 21 channels, and further preprocessing was done by renaming the channels and setting a 10-20 standard montage after importing the before/during task data .edf files and converting the class data into raw objects.

The two methods that will be used to address the research question for this project will be EEG data visualization/-analysis and neuromorphic computing through training the SNN model. Visual methods including topomaps, time-domain signals, and power spectral density (PSD) plots will be used to determine significant differences between the count quality groups. The training of the SNN will involve additional preprocessing steps including splitting the data into training and testing sets, converting these sets into tensor datasets, and transforming them into SNN-compatible DataLoaders before training on the SNN model itself. Evaluation metrics will include testing accuracy, balanced accuracy, and F1-score. Analysis using both methods for this project will be conducted once using the before task data and once using the during task data for further exploration of results.

III. Results

A. Visual Analysis

For the visual analysis component of this project, first power spectral density (PSD) plots were made; no hyperparameters were adjusted but this plot option was optimal to display all 21 EEG channel PSD signals. Next, topomaps were made to show the average amplitude of the signals across the different regions of the brain. The total duration of time for the good and bad quality count data was calculated, and given that the time of duration for the bad count quality data was around 600 seconds (for during task data), the topomaps were divided into 12 subplots with each subplot displaying the contours of the amplitude for a 50-second time window. Both good and bad count quality data topomaps were shown for the same time windows. Finally, beta-band filtered time-domain signals were plotted (for during task data), as the beta band is normally utilized for brain activities associated with thinking. For during task data, the plots (see Supplementary Materials) are shown for 10 seconds at the 150-second mark since the topomaps showed a large difference in amplitude between the two groups during that time window. Using the before task data, the time windows for the topomaps were 150 seconds each, and the band filter for the time-domain signal was alpha-band since this is most associated with consciousness. This was then plotted for 10 seconds at the 450-second mark. The time-domain signal plots for the before task data are shown below, however see Supplementary Materials for the rest of the plots.

According to the results, Figure 2 shows for the bad count quality group there tended to be higher oscillations in power and less negative power for the EEG channels located near the front of the brain during the mental arithmetic tasks, and Figure 1 shows less oscillations in frequency for the good count quality group and less negative power for the EEG channels located near the back of the brain. Figures 9 and 10 show about the same strength in oscillations in both groups using the before task data, however for the bad count quality group the channels near the back of the brain had slightly less negative power. During the mental arithmetic tasks, Figure 4 shows significantly more negative amplitude across all time windows for the bad count quality group compared to the good count quality group in Figure 3. For Figures 11 and 12 showing the topomaps using the before task data, with an exception for the 450-600 second time window there did not appear to be any significant differences in amplitude between the count quality groups. The time-domain signal plot in Figure 7 shows during the tasks, the beta-band filtered signal for the good count quality group tended to be slightly more amplified compared to the signal in Figure 8 using the bad count quality data. Finally, the alpha-band filtered signal in Figure 5 using the before task data shows the good count quality group having a significantly more amplified signal compared to the signal in Figure 6 for the bad count quality group.

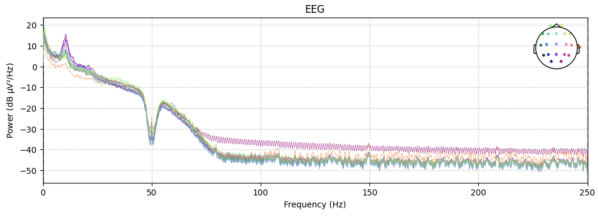


Fig. 1. Power Spectral Density for All Channels (Good Count Quality - During)

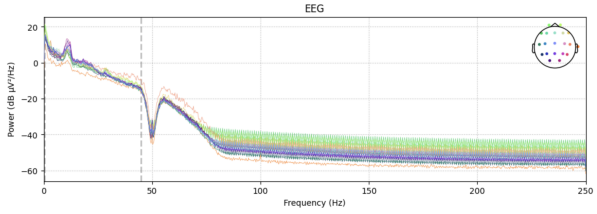


Fig. 2. Power Spectral Density for All Channels (Bad Count Quality - During)

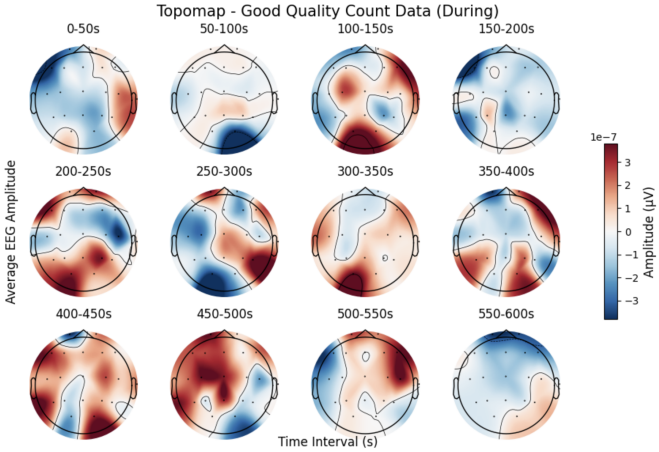


Fig. 3. Amplitude Topomap (Good Count Quality - During)

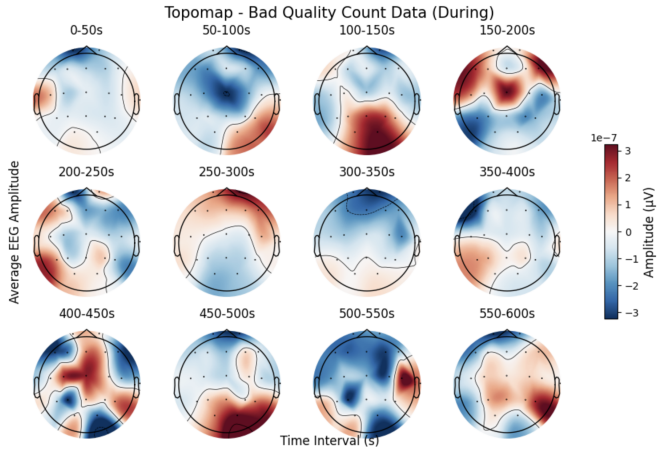


Fig. 4. Amplitude Topomap (Bad Count Quality - During)

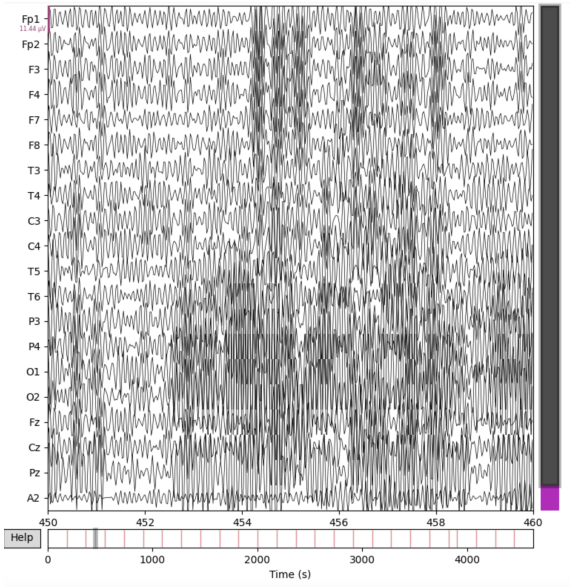


Fig. 5. Alpha-Band Filtered Time-Domain Signal (Good Count Quality - Before)

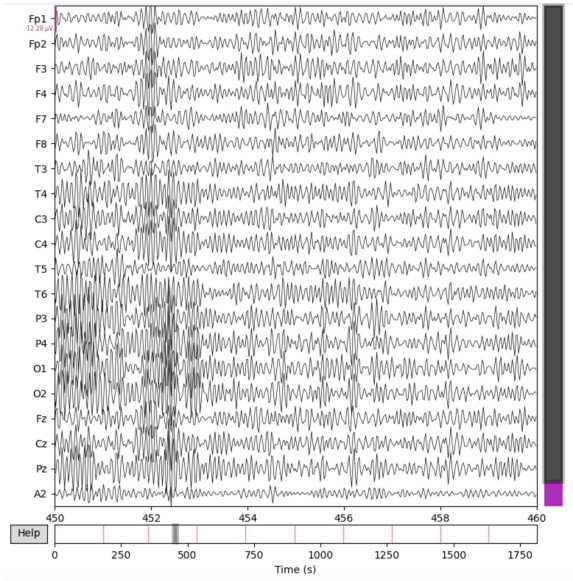


Fig. 6. Alpha-Band Filtered Time-Domain Signal (Bad Count Quality - Before)

B. Neuromorphic Computing Analysis

Using the during task data, the raw good and bad count quality signals were divided in 500 data point segments as this resulted in 2,232 segments total, a sufficient dataset size to train the SNN on in an efficient manner. The dataset was then split into 70% training and 30% testing; increasing or decreasing the training set size did not improve the results further. A batch size of 32 was chosen for the DataLoader and training the SNN to process the data relatively quickly. The SNN model architecture contained 2 layers each containing 100 neurons, as well as a Leaky-Integrate-and-Fire neural spiking layer with a beta of 0.9. The training used an Adam optimizer with a learning rate of 0.001. The data was trained on 50 epochs as convergence was usually reached after that many epochs. All of the parameters discussed provided the best possible results for this component of the project; many diverse efforts were made to improve the model accuracy by at least 5-10% past its peak, including extracting PSD from the segments, changing the number of neurons and/or beta in the layers, and changing the length of segments, however none of these succeeded. Using the before task data, the only parameter that was changed was the length of each segment, as the data was broken up into 1000 data point segments, resulting in a dataset size of 3,222 segments since the before task data contained much larger data point samples than the during task data.

In Table I, the results show that SNN predictions for both before and during task data had a modest accuracy of over 70% after training for 50 epochs. However, the balanced accuracies were very poor as the ratio of good count quality data to bad count quality data was 2:1. The F1-score for both before and during task data was adequate, however it is a metric that could likewise be improved.

Data	Accuracy	Balanced Accuracy	F1-Score
Before Task	0.7135	0.50	0.60
During Task	0.7209	0.50	0.61

TABLE I
EVALUATION METRICS AFTER 50 EPOCHS

IV. DISCUSSION

Based on the visual analysis results, these showed that PSD plots and topomaps using the during task data, and time-domain signal plots using the before task data, led to the most significant distinction results between the count quality groups. The bad count quality group in general had more oscillations in power and more negative amplitude while doing mental arithmetic tasks, and tended to have a less amplified alpha-band signal before doing the tasks. To interpret, this means that the bad count quality group tended to have less engagement and concentration and more stress before and during tasks. For real-world application, these symptoms can be major indicators of a (mathematical) learning disability [4] that may need to be diagnosed in subjects in this group.

Interpreting the neuromorphic computing analysis results, despite many attempts to improve the accuracy of the model, the SNN was able to achieve a modest accuracy of 71-72%; this model can sufficiently classify the two count quality groups for both task states. However, there was not a significant difference in whether the SNN tended to classify the count quality groups better or worse depending on the task state. Likewise, because the ratio of good to bad count quality data was 2:1 and the balanced classification accuracy was 50% for both task states, the SNN did not handle this dataset imbalance well.

The results from this study were probably the closest to the study conducted by Antelis & Falcón (2020) where they achieved a 74% accuracy using SNNs [1], however studies like the one from Luo et al. showed much better accuracy using SNNs [5]. Still, this was for general SNN tasks using EEG data and not SNN tasks using the dataset used in the study [3], which this study contributed this novel approach. Using the dataset [3] in the study done by Salankar, Koundal, & Qaisar (2021), they were able to achieve an accuracy of up to 100% with classifying the 2 count quality groups [7], however it should be noted this was not done using an SNN.

Despite the shortcomings with the neuromorphic computing analysis, the impacts of this study are still significant. This work contributed an additional data-centered perspective on what differentiates the good and bad count quality groups in the EEG During Mental Arithmetic Tasks dataset [3]. The visual analysis component showed some distinctive differences in the EEG signal factors between the 2 groups, and this was the first time an SNN had been utilized on this dataset for training to predict the count quality groups, and this SNN achieved a sufficient accuracy of over 70%. The real-world impact of this study is that visual and neuromorphic computing analysis can help identify and diagnose individuals who may have mathematical learning disabilities.

V. CONCLUSION

As stated before, the limitations of this project include that the SNN could still be improved across all evaluation metrics and the SNN should especially handle imbalanced datasets better. In addition, although some of the visual analysis comparisons were able to effectively distinguish the two groups, not all tested comparisons were able to.

Future work could include an improved approach for training the SNN to make stronger predictions. This can be done through more data collection of bad count quality EEG data from subjects, or generating simulated EEG data of bad count quality subjects using a Generative Adversarial Network (GAN), to make the dataset more balanced. Another technique that could be used is more oversampling of the bad count quality data, however this could only improve balanced accuracy and not the overall accuracy itself, as it is preferable to see both accuracies improved in the future. Or as an alternative approach, the SNN could instead be trained to detect before vs. during task EEG data to see if it would perform better doing this classification task compared to classifying the count quality groups (this would still be an original approach using the dataset), and compare the results with other studies. Future work could likewise include more visual analysis techniques to more effectively distinguish the count quality groups. An example could be reducing the features of the data further and evaluating them visually using methods like anomaly detection and t-Distributed Stochastic Neighbor Embedding (t-SNE) (i.e., analyzing the value ranges of the reduced components to see if there are any differences between the 2 classes).

REFERENCES

[1] Antelis, J. M., & Falcón, L. E. (2020). Spiking neural networks applied to the classification of motor tasks in EEG signals. *Neural networks, 122*, 130-143.

[2] Catrambone, V., & Valenza, G. (2023). Complex brain–heart mapping in mental and physical stress. *IEEE Journal of Translational Engineering in Health and Medicine, 11*, 495-504.

[3] Goldberger, A., Amaral, L., Glass, L., Hausdorff, J., Ivanov, P. C., Mark, R., ... & Stanley, H. E. (2000). PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals. Circulation [Online]. 101 (23), pp. e215–e220.

[4] Hellstrand, H., Holopainen, S., Korhonen, J., Räsänen, P., Hakkarainen, A., Laakso, M. J., ... & Aunio, P. (2024). Arithmetic fluency and number processing skills in identifying students with mathematical learning disabilities. *Research in Developmental Disabilities, 151*, 104795.

[5] Luo, Y., Fu, Q., Xie, J., Qin, Y., Wu, G., Liu, J., ... & Ding, X. (2020). EEG-based emotion classification using spiking neural networks. *IEEE Access, 8*, 46007-46016.

[6] Nunes, J. D., Carvalho, M., Carneiro, D., & Cardoso, J. S. (2022). Spiking neural networks: A survey. *IEEE Access, 10*, 60738-60764.

[7] Salankar, N., Koundal, D., & Mian Qaisar, S. (2021). Stress classification by multimodal physiological signals using variational mode decomposition and machine learning. *Journal of healthcare engineering, 2021*(1), 2146369.

[8] Zyma I, Tukaev S, Seleznov I, Kiyono K, Popov A, Chernykh M, Shpenkov O. Electroencephalograms during Mental Arithmetic Task Performance. Data. 2019; 4(1):14. https://doi.org/10.3390/data4010014

VI. Supplementary Materials

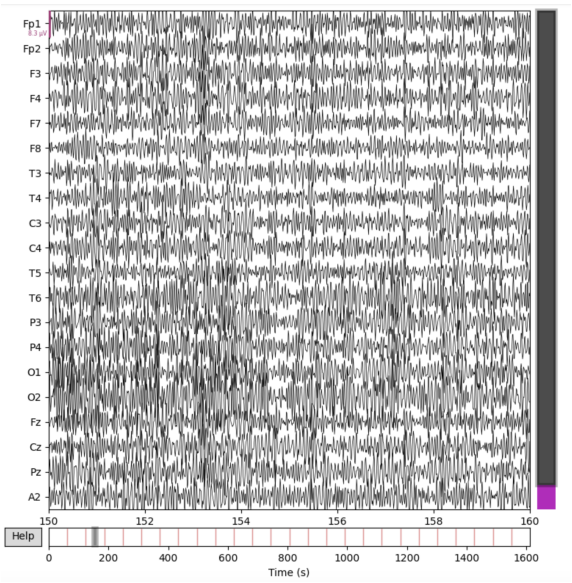


Fig. 7. Beta-Band Filtered Time-Domain Signal (Good Count Quality - During)

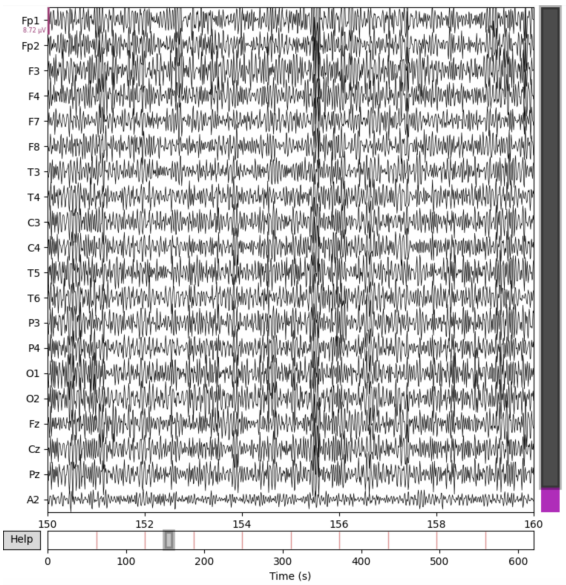


Fig. 8. Beta-Band Filtered Time-Domain Signal (Bad Count Quality - During)

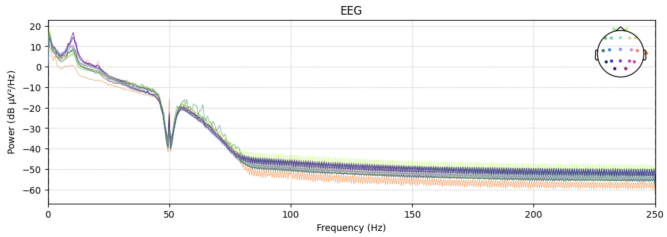


Fig. 9. Power Spectral Density for All Channels (Good Count Quality - Before)

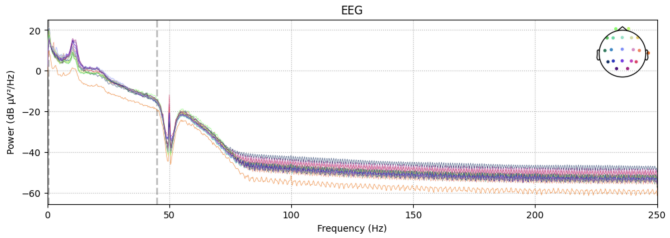


Fig. 10. Power Spectral Density for All Channels (Bad Count Quality - Before)

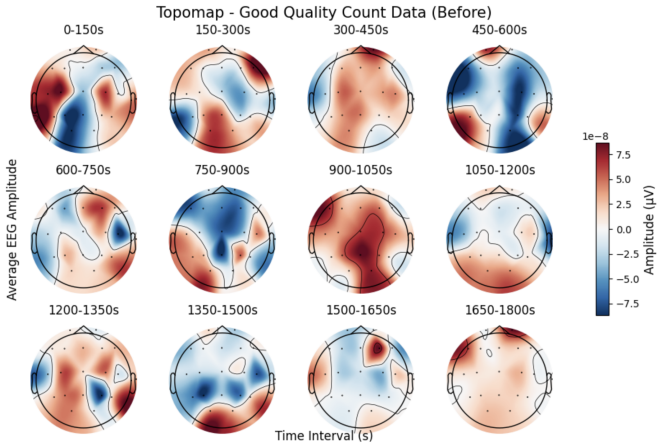


Fig. 11. Amplitude Topomap (Good Count Quality - Before)

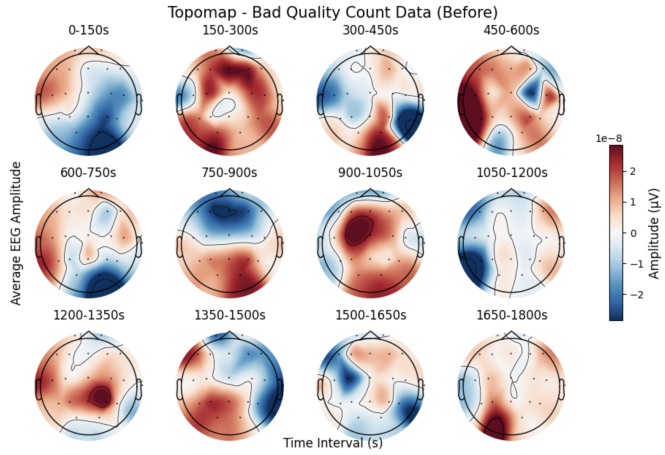


Fig. 12. Amplitude Topomap (Bad Count Quality - Before)

VII. Code

A. Before Task Data

```
# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')
%cd /content/drive/MyDrive/
```

```
Mounted at /content/drive
/content/drive/MyDrive
```

```
# For output file path
import os

# Path to the folder containing EEG for Mental
    Arithmetic Tasks dataset files in Google Drive
edf_folder_path = '/content/drive/MyDrive/eeg-mental
    -arithmetic/'

# Import packages
import pandas as pd
import pyedflib # For reading EEG data
import numpy as np
from torch.utils.data import DataLoader,
    TensorDataset # For dataloader when about to
    train SNN
import snntorch as snn # SNN package
import torch
# Import MNE packages
import mne # Reading EEG data
import mne.viz # EEG data visualizations
from mne.io import concatenate_raws, read_raw_edf #
    Reading in raw EDFs
from mne.channels import make_standard_montage # EDF
     montage

# Load the labels from the CSV file into a
    dictionary for matching with edf files
csv_path = '/content/drive/MyDrive/eeg-mental-
    arithmetic/subject-info.csv'
labels_df = pd.read_csv(csv_path)
# Look at labels for each subject
display(labels_df)
```

	Subject	Age	Gender	Recording year	Number of subtractions	Count quality
0	Subject00	21	F	2011	9.70	0
1	Subject01	18	F	2011	29.35	1
2	Subject02	19	F	2012	12.88	1
3	Subject03	17	F	2010	31.00	1
4	Subject04	17	F	2010	8.60	0
5	Subject05	16	F	2010	20.71	1
6	Subject06	18	M	2011	4.35	0
7	Subject07	18	F	2012	13.38	1
8	Subject08	26	M	2011	18.24	1
9	Subject09	16	F	2010	7.00	0
10	Subject10	17	F	2010	1.00	0
11	Subject11	18	F	2010	26.00	1
12	Subject12	17	F	2010	26.36	1
13	Subject13	24	M	2012	34.00	1
14	Subject14	17	F	2010	9.00	0
15	Subject15	17	F	2012	22.18	1
16	Subject16	17	F	2010	11.59	1
17	Subject17	17	F	2010	28.70	1
18	Subject18	17	F	2010	20.00	1

```
# Import all _1 edf files (before task), manually
    import files corresponding to each group

# Separate into bad quality count (label 0) and good
     quality count (label 1)
bad_count_files = ['Subject00_1.edf', 'Subject04_1.
    edf', 'Subject06_1.edf', 'Subject09_1.edf',
                   'Subject10_1.edf', 'Subject14_1.
                       edf', 'Subject19_1.edf', '
                       Subject21_1.edf',
                   'Subject22_1.edf', 'Subject30_1.
                       edf'] # 12 bad count quality
                       subjects
good_count_files = ['Subject01_1.edf', 'Subject02_1.
    edf', 'Subject03_1.edf', 'Subject05_1.edf',
                    'Subject07_1.edf', 'Subject08_1.
                        edf', 'Subject11_1.edf', '
                        Subject12_1.edf',
                    'Subject13_1.edf', 'Subject15_1.
                        edf', 'Subject16_1.edf', '
                        Subject17_1.edf',
                    'Subject18_1.edf', 'Subject20_1.
                        edf', 'Subject23_1.edf', '
                        Subject24_1.edf',
                    'Subject25_1.edf', 'Subject26_1.
                        edf', 'Subject27_1.edf', '
                        Subject28_1.edf',
                    'Subject29_1.edf', 'Subject31_1.
                        edf', 'Subject32_1.edf', '
                        Subject33_1.edf',
                    'Subject34_1.edf', 'Subject35_1.
                        edf'] # 24 good count
                        quality subjects

# Initialize lists to store raw objects for bad
    count and good count quality
raw_good_list = []
raw_bad_list = []

# Function to load edf files into a raw object
def load_edf_files(edf_files, folder_path):
    raw_list = []
    for edf_file in edf_files:
        edf_path = os.path.join(folder_path,
            edf_file) # Find folder path and edf
            file

        # Check if the file exists before proceeding
        if not os.path.exists(edf_path):
            print(f"File {edf_file} not found!")
            continue

        # Load the edf file into mne
        try:
            raw = mne.io.read_raw_edf(edf_path,
                preload=True)
            raw_list.append(raw)  # Append to the
                list of raw objects
            # Print whether import was successful
            print(f"Successfully loaded {edf_file}
                with {len(raw.ch_names)} channels.")
        except Exception as e:
            print(f"Error loading {edf_file}: {e}")
    return raw_list

# Load good and bad count quality files separately
    using function
raw_good_list = load_edf_files(good_count_files,
    edf_folder_path)
raw_bad_list = load_edf_files(bad_count_files,
    edf_folder_path)

# Concatenate raws for good and bad count conditions
     separately
raw_good = mne.concatenate_raws(raw_good_list) # Raw
     good count quality object
raw_bad = mne.concatenate_raws(raw_bad_list) # Raw
    bad count quality object
```

```
Extracting EDF parameters from /content/drive/MyDrive/eeg-mental-arithmetic/Subject01_1.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 90999  =      0.000 ...   181.998 secs...
Successfully loaded Subject01_1.edf with 21 channels.
Extracting EDF parameters from /content/drive/MyDrive/eeg-mental-arithmetic/Subject02_1.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 90999  =      0.000 ...   181.998 secs...
Successfully loaded Subject02_1.edf with 21 channels.
Extracting EDF parameters from /content/drive/MyDrive/eeg-mental-arithmetic/Subject03_1.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 90999  =      0.000 ...   181.998 secs...
Successfully loaded Subject03_1.edf with 21 channels.
Extracting EDF parameters from /content/drive/MyDrive/eeg-mental-arithmetic/Subject05_1.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 90999  =      0.000 ...   181.998 secs...
Successfully loaded Subject05_1.edf with 21 channels.
Extracting EDF parameters from /content/drive/MyDrive/eeg-mental-arithmetic/Subject07_1.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 90999  =      0.000 ...   181.998 secs...
Successfully loaded Subject07_1.edf with 21 channels.
Extracting EDF parameters from /content/drive/MyDrive/eeg-mental-arithmetic/Subject08_1.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
```

```
# Set the channel locations and types for both raw
    objects
for raw in [raw_good, raw_bad]:
    raw.rename_channels({'EEG Fp1':'Fp1','EEG Fp2':'
        Fp2','EEG F3':'F3','EEG F4':'F4','EEG F7':'
        F7','EEG F8':'F8',
                         'EEG T3':'T3','EEG T4':'T4
                             ','EEG C3':'C3','EEG C4
                             ':'C4','EEG T5':'T5','
                             EEG T6':'T6','EEG P3':'
                             P3',
                         'EEG P4':'P4','EEG O1':'O1
                             ','EEG O2':'O2','EEG Fz
                             ':'Fz','EEG Cz':'Cz','
                             EEG Pz':'Pz',
                         'EEG A2-A1':'A2','ECG ECG
                             ':'ECG'})
    raw.set_channel_types({'ECG': 'ecg'})
    raw.set_montage(mne.channels.
        make_standard_montage('standard_1020'))  #
        Set montage

# Get data from raw good count object
raw_data_1 = raw_good.get_data()
print("Number of channels: ", str(len(raw_data_1)))
    # print channels
print("Number of samples: ", str(len(raw_data_1[0]))
    ) # print samples
# Get Sampling Frequency
fs_g=raw_good.info['sfreq']
print("Sampling Frequency: ", str(fs_g))# Get data
    from raw task object
```

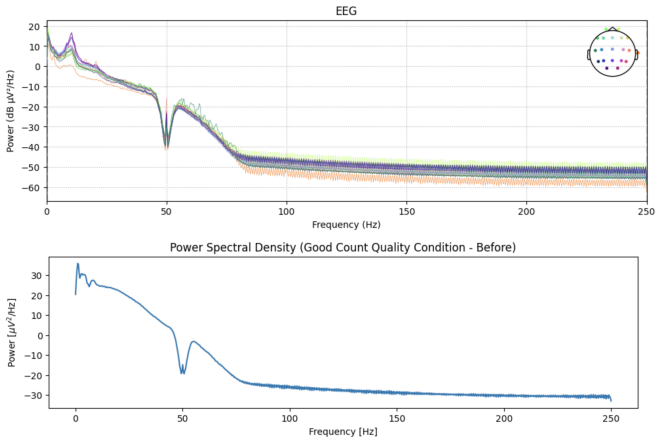
```
Number of channels:  21
Number of samples:  2315000
Sampling Frequency:  500.0
```

```
# Get data from raw bad count object
raw_data_2 = raw_bad.get_data()
print("Number of channels: ", str(len(raw_data_2)))
    # print channels
print("Number of samples: ", str(len(raw_data_2[0]))
    ) # print samples
# Get Sampling Frequency
fs_b=raw_bad.info['sfreq']
print("Sampling Frequency: ", str(fs_b))
```

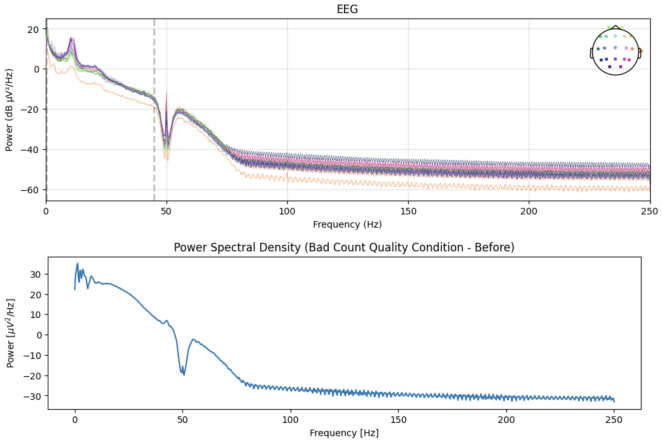
```
Number of channels:  21
Number of samples:  907000
```

```
Sampling Frequency:  500.0
```

```
# Import more packages
%matplotlib inline
import scipy as sp
from scipy import stats
import matplotlib.pyplot as plt
import copy

# Plot_psd function for plotting raw good count
    object Power Spectral Density (PSD)
raw_good.plot_psd()
# Alternative scipy plot with one line instead of
    multiple channels
f, Sxx = sp.signal.welch(raw_good._data[20,:], fs=
    fs_g, window='hann'),
nperseg=1500, noverlap=1000, scaling='density')
# The scale of Sxx use scaling='density' is V**2/Hz
# To change the scale to muV**2/Hz(Db) which mne
    uses, use the following conversion
plt.figure(figsize=(11.6,3))
plt.plot(f, 10*np.log10(Sxx*1e6**2))
plt.xlabel('Frequency [Hz]')
plt.ylabel('Power [$\mu V^2$/Hz]')
plt.title('Power Spectral Density (Good Count
    Quality Condition - Before)')
plt.show()
```




```
# Plot_psd function for plotting raw bad count
    object Power Spectral Density (PSD)
raw_bad.plot_psd()
# Alternative scipy plot with one line instead of
    multiple channels
f, Sxx = sp.signal.welch(raw_bad._data[20,:], fs=
    fs_b, window='hann'),
nperseg=1500, noverlap=1000, scaling='density')
# The scale of Sxx use scaling='density' is V**2/Hz
# To change the scale to muV**2/Hz(Db) which mne
    uses, use the following conversion
plt.figure(figsize=(11.6,3))
plt.plot(f, 10*np.log10(Sxx*1e6**2))
plt.xlabel('Frequency [Hz]')
plt.ylabel('Power [$\mu V^2$/Hz]')
plt.title('Power Spectral Density (Bad Count Quality
     Condition - Before)')
plt.show()
```



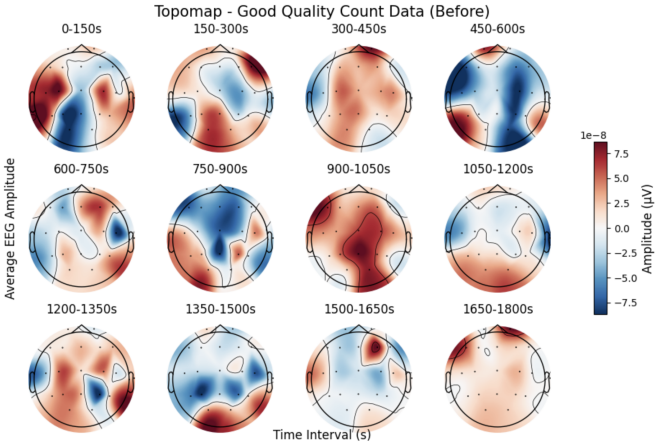

```
# Check how many seconds are in good data to
    determine time windows for topomap
total_duration_seconds = raw_good.times[-1]
print(f'Total duration: {total_duration_seconds}
    seconds')
```

Total duration: 4629.998 seconds

```
# Check how many seconds are in bad data to
    determine time windows for topomap
total_duration_seconds = raw_bad.times[-1]
print(f'Total duration: {total_duration_seconds}
    seconds')
```

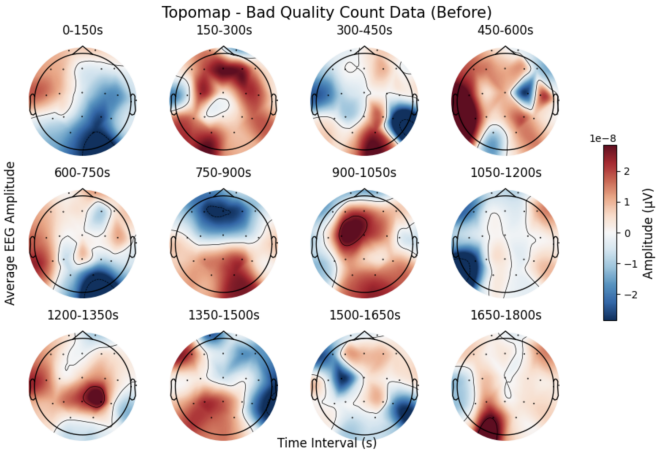
Total duration: 1813.998 seconds

```
# Define time intervals in seconds
time_intervals = [(0, 150), (150, 300), (300, 450),
    (450, 600), (600, 750), (750, 900),
 (900, 1050), (1050, 1200), (1200, 1350), (1350,
    1500), (1500, 1650), (1650, 1800)]

# Create a 3-row x 4-column subplot grid for 12 time
    windows
fig, axes = plt.subplots(3, 4, figsize=(9, 6))

# Flatten axes array for easier iteration
axes = axes.flatten()

# Create colorbar (for shared colorbar)
cbar_ax = fig.add_axes([0.92, 0.3, 0.02, 0.4])  # [
    left, bottom, width, height] placement

# Iterate over the time intervals and plot the
    topomaps
for i, (start, end) in enumerate(time_intervals):
    # Extract the data for the current time interval
    start_sample = int(start * raw_good.info['sfreq
        '])  # Convert time to samples
    end_sample = int(end * raw_good.info['sfreq'])
        # Convert time to samples
    interval_data = raw_good.get_data(picks='eeg',
        start=start_sample, stop=end_sample)  #
        Select 'eeg' channels

    # Compute the average signal over the interval
    avg_data = np.mean(interval_data, axis=1)  #
        Mean across time points for each channel

    # Plot the topomap for the current time interval
        in the corresponding subplot
    im, _ = mne.viz.plot_topomap(
        avg_data,
        pos=raw_good.info,
        ch_type='eeg',
        axes=axes[i],
        show=False,
        contours=1  # Show contours
    )
    axes[i].set_title(f'{start}-{end}s')  # Add
        topomap label for time window

# Add colorbar
cbar = fig.colorbar(im, cax=cbar_ax)
cbar.set_label('Amplitude ( V )', fontsize=12)

# Add axis labels and title to entire figure
fig.text(0.5, 0.01, 'Time Interval (s)', ha='center
    ', fontsize=12)
fig.text(0.01, 0.5, 'Average EEG Amplitude', va='
    center', rotation='vertical', fontsize=12)
fig.text(0.5, 0.99, 'Topomap - Good Quality Count
    Data (Before)', ha='center', fontsize=15)

# Adjust layout
plt.tight_layout(rect=[0, 0, 0.9, 1])  # Leave space
     for colorbar
plt.show()
```

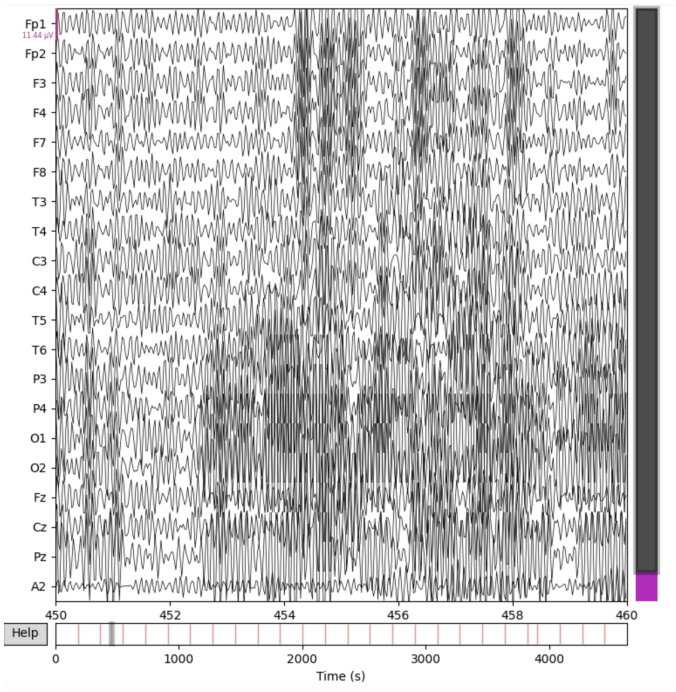



```
# Define time intervals in seconds
time_intervals = [(0, 150), (150, 300), (300, 450),
    (450, 600), (600, 750), (750, 900),
 (900, 1050), (1050, 1200), (1200, 1350), (1350,
    1500), (1500, 1650), (1650, 1800)]

# Create a 3-row x 4-column subplot grid for 12 time
    windows
fig, axes = plt.subplots(3, 4, figsize=(9, 6))

# Flatten axes array for easier iteration
axes = axes.flatten()

# Create common colorbar axis
cbar_ax = fig.add_axes([0.92, 0.3, 0.02, 0.4])  # [
    left, bottom, width, height]

# Iterate over the time intervals and plot the
    topomaps
for i, (start, end) in enumerate(time_intervals):
    # Extract the data for the current time interval
    start_sample = int(start * raw_bad.info['sfreq
        '])  # Convert time to samples
```

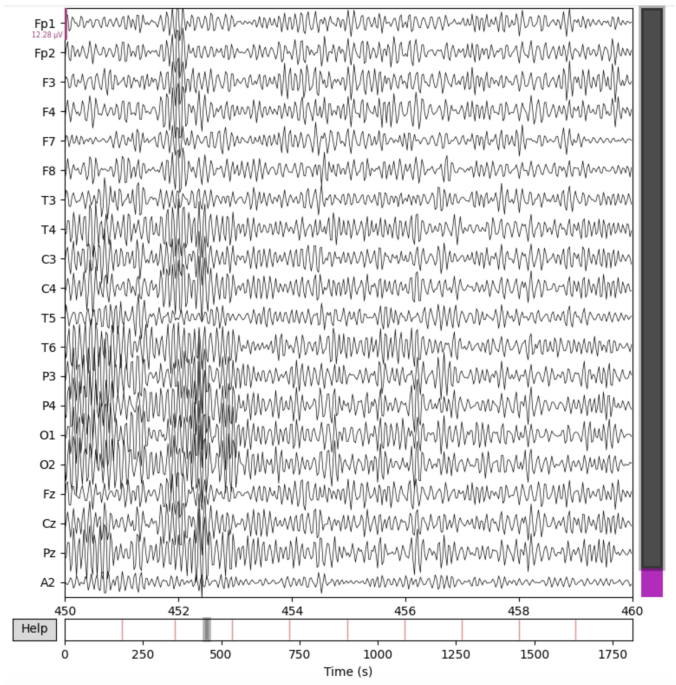
```
    end_sample = int(end * raw_bad.info['sfreq'])  #
        Convert time to samples
    interval_data = raw_bad.get_data(picks='eeg',
        start=start_sample, stop=end_sample)  #
        Select 'eeg' channels

    # Compute the average signal over the interval
    avg_data = np.mean(interval_data, axis=1)  #
        Mean across time points for each channel

    # Plot the topomap for the current time interval
        in the corresponding subplot
    im, _ = mne.viz.plot_topomap(
        avg_data,
        pos=raw_bad.info,
        ch_type='eeg',
        axes=axes[i],
        show=False,
        contours=1  # Show contours
    )
    axes[i].set_title(f'{start}-{end}s')  # Add
        topomap label for time window

# Add colorbar
cbar = fig.colorbar(im, cax=cbar_ax)
cbar.set_label('Amplitude ( V )', fontsize=12)

# Add axis labels and title to entire figure
fig.text(0.5, 0.01, 'Time Interval (s)', ha='center
    ', fontsize=12)
fig.text(0.01, 0.5, 'Average EEG Amplitude', va='
    center', rotation='vertical', fontsize=12)
fig.text(0.5, 0.99, 'Topomap - Bad Quality Count
    Data (Before)', ha='center', fontsize=15)

# Adjust layout
plt.tight_layout(rect=[0, 0, 0.9, 1])  # Leave space
    for the colorbar
plt.show()
```




```
# Plot the time-domain signal for raw_good between
    alpha band (common duing consciousness)
raw_good.filter(8., 12.) # Filter between alpha band
    (8-12 Hz)
# Plot the time-domain signal within time window
raw_good.plot(scalings='auto', title='Time-Domain
    Signal: Good Count Quality EEG (Before)',
                  start=450,  # Start time in
                      seconds
                  duration=10, # Duration of time
                      window
                  show=True)
```




```
# Plot the time-domain signal for raw_bad between
    alpha band (common during consciousness)
raw_bad.filter(8., 12.) # Filter between alpha band
    (8-12 Hz)
# Plot the time-domain signal within time window
raw_bad.plot(scalings='auto', title='Time-Domain
    Signal: Bad Count Quality EEG (Before)',
                  start=450,  # Start time in
                      seconds
                  duration=10, # Duration of time
                      window
                  show=True)
```



```
# Extract data and sampling frequency from the raw
    good and bad count quality objects
data_good, _ = raw_good[:, :]
data_bad, _ = raw_bad[:, :]
fs_g = int(raw_good.info['sfreq'])
fs_b = int(raw_bad.info['sfreq'])
print(fs_g, fs_b) # Print sampling frequencies of
    the raw objects
```

```
500 500
```

```
# Set segment size to 1000 sample data points for
    splitting (larger samples due to more before
    task data)
window_size = 1000

# Segment the data into non-overlapping windows
def segment_data(data, window_size):
    num_segments = data.shape[1] // window_size #
        Split into 1000-sample point segments
    segments = np.array_split(data[:, :num_segments
        * window_size], num_segments, axis=1) #
        Convert into numpy array
    return segments

# Segment the good and bad data
segments_good = segment_data(data_good, window_size)
segments_bad = segment_data(data_bad, window_size)

# Label the segments as 1 for good, 0 for bad as in
    csv
labels_good = np.ones(len(segments_good))
labels_bad = np.zeros(len(segments_bad))

# Combine the segments and labels into numpy dataset
segments = np.array(segments_good + segments_bad)  #
     All segmented data
labels = np.concatenate([labels_good, labels_bad])
    # All label data

from sklearn.metrics import accuracy_score,
    balanced_accuracy_score, f1_score # Evaluation
    metrics

# Print shape of all segments and labels
print(np.shape(segments))
print(np.shape(labels))
# Change dimension of data for ML Algorithm
X_all=np.mean(segments,axis=1)
Y_all=labels
X_all=sp.stats.zscore(X_all,axis=1)
print(np.shape(X_all))
print(np.shape(Y_all))
# Set to X and Y for training
X=X_all
Y=Y_all
print(np.shape(X))
print(np.shape(Y))
# Check for NaN values in X and Y
print(np.isnan(X).any())
print(np.isnan(Y).any())
```

```
(3222, 21, 1000)
(3222,)
(3222, 1000)
(3222,)
(3222, 1000)
(3222,)
False
False
```

```
# Since there are NaN values in X (sometimes there
    are no NaN values), impute using SimpleImputer
from sklearn.impute import SimpleImputer

# Handle NaN values using SimpleImputer
imputer = SimpleImputer(strategy='mean')  # Replace
    NaNs with the mean of each column
X = imputer.fit_transform(X) # Set X to newly
    imputed missing values

# Take counts of Y to see imbalance of labels
unique, counts = np.unique(Y, return_counts=True)
print(dict(zip(unique, counts)))
```

```
{0.0: 907, 1.0: 2315}
```

```
# Confirm shape of X and Y again
print(X.shape)
print(Y.shape)
```

```
(3222, 1000)
(3222,)
```

```
# Import train_test_split
from sklearn.model_selection import train_test_split

# Split the data into training and testing sets -
    70% training and 30% testing
X_train, X_test, Y_train, Y_test = train_test_split(
    X, Y, test_size=0.3, random_state=42, stratify=Y
    )

# Check shapes
print("Training set shape:", X_train.shape, Y_train.
    shape)
print("Testing set shape:", X_test.shape, Y_test.
    shape)
```

```
Training set shape: (2255, 1000) (2255,)
Testing set shape: (967, 1000) (967,)
```

```
# Convert data to PyTorch tensors
X_train_tensor = torch.FloatTensor(X_train)
Y_train_tensor = torch.LongTensor(Y_train)
X_test_tensor = torch.FloatTensor(X_test)
Y_test_tensor = torch.LongTensor(Y_test)

# Create TensorDataset objects for training and
    testing
train_dataset = TensorDataset(X_train_tensor,
    Y_train_tensor)
test_dataset = TensorDataset(X_test_tensor,
    Y_test_tensor)

# Define DataLoader batch size
batch_size = 32

# Create DataLoaders for training and testing sets
train_loader = DataLoader(train_dataset, batch_size=
    batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=
    batch_size, shuffle=False)

# Check if DataLoader works correctly by iterating
    over a batch
for X_batch, Y_batch in train_loader:
    print("X_batch shape:", X_batch.shape)
```

```
    print("Y_batch shape:", Y_batch.shape)
    break  # Stop showing after 1 batch
```

```
X_batch shape: torch.Size([32, 1000])
Y_batch shape: torch.Size([32])
```

```
# Import torch.nn
import torch.nn as nn

# Define the SNN model architecture
class SNNModel(nn.Module):
    def __init__(self, input_size):
        super(SNNModel, self).__init__()
        self.fc1 = nn.Linear(input_size, 100)  #
            Fully connected layer with 100 neurons
        self.lif1 = snn.Leaky(beta=0.9)  # LIF
            neuron with snn.Leaky to handle spikes
        self.fc2 = nn.Linear(100, 2)  # Output layer
             for binary classification

    def forward(self, x):
        # Pass through the first fully connected
            layer
        x = self.fc1(x)

        # Process with the LIF layer and generate
            spikes
        mem, spikes = self.lif1(x)

        # Pass through the second fully connected
            layer
        output = self.fc2(spikes)

        return output

# Initialize network, optimizer, and loss function
input_size = X_train.shape[1]
model = SNNModel(input_size)
optimizer = torch.optim.Adam(model.parameters(), lr
    =0.001)
criterion = nn.CrossEntropyLoss()

# Training loop
num_epochs = 50
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0  # Track loss for each epoch

    # Iterate over batches from the training loader
    for X_batch, Y_batch in train_loader:
        optimizer.zero_grad()  # Zero the gradients

        # Forward pass
        output = model(X_batch)
        loss = criterion(output, Y_batch)

        # Backward pass and optimization
        loss.backward(retain_graph=True)
        optimizer.step()

        running_loss += loss.item() * X_batch.size
            (0)  # Accumulate loss

    # Calculate average loss for the epoch
    epoch_loss = running_loss / len(train_loader.
        dataset)

    # Print training loss every epoch
    if (epoch + 1) % 1 == 0:
        print(f'Epoch [{epoch + 1}/{num_epochs}],
            Training Loss: {epoch_loss:.4f}')

    # Evaluation on test data
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for X_batch, Y_batch in test_loader:
            outputs = model(X_batch) # evaluates
                based on batch in X
            _, predicted = torch.max(outputs.data,
                1) # takes maximum score

            all_preds.extend(predicted.cpu().numpy()
                ) # appends to all_preds
            all_labels.extend(Y_batch.cpu().numpy())
                 # appends to all_labels

    # Calculate Accuracy, Balanced Accuracy, and F1-
        Score
    accuracy = accuracy_score(all_labels, all_preds)
    balanced_accuracy = balanced_accuracy_score(
        all_labels, all_preds)
    f1 = f1_score(all_labels, all_preds, average='
        weighted')  # 'weighted' accounts for class
        imbalance

    # Print evaluation metrics
    print(f'Epoch [{epoch + 1}/{num_epochs}], Test
        Accuracy: {accuracy*100:.2f}%')
    print(f'Epoch [{epoch + 1}/{num_epochs}],
        Balanced Accuracy: {balanced_accuracy:.2f}')
    print(f'Epoch [{epoch + 1}/{num_epochs}], F1-
        Score: {f1:.2f}')
```

```
Epoch [1/50], Training Loss: 0.8158
Epoch [1/50], Test Accuracy: 59.98%
Epoch [1/50], Balanced Accuracy: 0.50
Epoch [1/50], F1-Score: 0.60
Epoch [2/50], Training Loss: 0.6292
Epoch [2/50], Test Accuracy: 67.32%
Epoch [2/50], Balanced Accuracy: 0.50
Epoch [2/50], F1-Score: 0.62
Epoch [3/50], Training Loss: 0.6242
Epoch [3/50], Test Accuracy: 68.98%
Epoch [3/50], Balanced Accuracy: 0.51
Epoch [3/50], F1-Score: 0.63
Epoch [4/50], Training Loss: 0.6237
Epoch [4/50], Test Accuracy: 69.29%
Epoch [4/50], Balanced Accuracy: 0.49
Epoch [4/50], F1-Score: 0.60
Epoch [5/50], Training Loss: 0.6185
Epoch [5/50], Test Accuracy: 66.29%
Epoch [5/50], Balanced Accuracy: 0.49
Epoch [5/50], F1-Score: 0.60
Epoch [6/50], Training Loss: 0.6171
Epoch [6/50], Test Accuracy: 68.87%
Epoch [6/50], Balanced Accuracy: 0.50
Epoch [6/50], F1-Score: 0.62
Epoch [7/50], Training Loss: 0.6190
Epoch [7/50], Test Accuracy: 69.39%
Epoch [7/50], Balanced Accuracy: 0.50
Epoch [7/50], F1-Score: 0.61
Epoch [8/50], Training Loss: 0.6124
Epoch [8/50], Test Accuracy: 67.94%
Epoch [8/50], Balanced Accuracy: 0.48
```

```
Epoch [8/50], F1-Score: 0.59
Epoch [9/50], Training Loss: 0.6238
Epoch [9/50], Test Accuracy: 70.22%
Epoch [9/50], Balanced Accuracy: 0.50
Epoch [9/50], F1-Score: 0.61
Epoch [10/50], Training Loss: 0.6172
Epoch [10/50], Test Accuracy: 69.60%
Epoch [10/50], Balanced Accuracy: 0.50
Epoch [10/50], F1-Score: 0.61
Epoch [11/50], Training Loss: 0.6161
Epoch [11/50], Test Accuracy: 71.25%
Epoch [11/50], Balanced Accuracy: 0.50
Epoch [11/50], F1-Score: 0.61
Epoch [12/50], Training Loss: 0.6081
Epoch [12/50], Test Accuracy: 70.63%
Epoch [12/50], Balanced Accuracy: 0.50
Epoch [12/50], F1-Score: 0.60
Epoch [13/50], Training Loss: 0.6097
Epoch [13/50], Test Accuracy: 71.46%
Epoch [13/50], Balanced Accuracy: 0.50
Epoch [13/50], F1-Score: 0.60
Epoch [14/50], Training Loss: 0.6057
Epoch [14/50], Test Accuracy: 71.77%
Epoch [14/50], Balanced Accuracy: 0.50
Epoch [14/50], F1-Score: 0.60
Epoch [15/50], Training Loss: 0.6105
Epoch [15/50], Test Accuracy: 67.53%
Epoch [15/50], Balanced Accuracy: 0.49
Epoch [15/50], F1-Score: 0.60
Epoch [16/50], Training Loss: 0.6103
Epoch [16/50], Test Accuracy: 71.77%
Epoch [16/50], Balanced Accuracy: 0.50
Epoch [16/50], F1-Score: 0.60
Epoch [17/50], Training Loss: 0.6010
Epoch [17/50], Test Accuracy: 70.42%
Epoch [17/50], Balanced Accuracy: 0.50
Epoch [17/50], F1-Score: 0.60
Epoch [18/50], Training Loss: 0.6030
Epoch [18/50], Test Accuracy: 71.15%
Epoch [18/50], Balanced Accuracy: 0.50
Epoch [18/50], F1-Score: 0.60
Epoch [19/50], Training Loss: 0.6022
Epoch [19/50], Test Accuracy: 69.49%
Epoch [19/50], Balanced Accuracy: 0.50
Epoch [19/50], F1-Score: 0.61
Epoch [20/50], Training Loss: 0.6102
Epoch [20/50], Test Accuracy: 71.15%
Epoch [20/50], Balanced Accuracy: 0.50
Epoch [20/50], F1-Score: 0.60
Epoch [21/50], Training Loss: 0.6042
Epoch [21/50], Test Accuracy: 71.15%
Epoch [21/50], Balanced Accuracy: 0.50
Epoch [21/50], F1-Score: 0.60
Epoch [22/50], Training Loss: 0.6069
Epoch [22/50], Test Accuracy: 71.87%
Epoch [22/50], Balanced Accuracy: 0.50
Epoch [22/50], F1-Score: 0.60
Epoch [23/50], Training Loss: 0.6042
Epoch [23/50], Test Accuracy: 72.18%
Epoch [23/50], Balanced Accuracy: 0.51
Epoch [23/50], F1-Score: 0.61
Epoch [24/50], Training Loss: 0.6059
Epoch [24/50], Test Accuracy: 71.87%
Epoch [24/50], Balanced Accuracy: 0.50
Epoch [24/50], F1-Score: 0.60
Epoch [25/50], Training Loss: 0.6023
Epoch [25/50], Test Accuracy: 70.73%
Epoch [25/50], Balanced Accuracy: 0.49
Epoch [25/50], F1-Score: 0.60
Epoch [26/50], Training Loss: 0.6024
Epoch [26/50], Test Accuracy: 70.42%
Epoch [26/50], Balanced Accuracy: 0.50
Epoch [26/50], F1-Score: 0.61
Epoch [27/50], Training Loss: 0.6117
Epoch [27/50], Test Accuracy: 71.87%
Epoch [27/50], Balanced Accuracy: 0.50
Epoch [27/50], F1-Score: 0.61
Epoch [28/50], Training Loss: 0.5997
Epoch [28/50], Test Accuracy: 70.63%
Epoch [28/50], Balanced Accuracy: 0.49
Epoch [28/50], F1-Score: 0.60
Epoch [29/50], Training Loss: 0.6093
Epoch [29/50], Test Accuracy: 68.25%
Epoch [29/50], Balanced Accuracy: 0.51
Epoch [29/50], F1-Score: 0.62
Epoch [30/50], Training Loss: 0.6122
Epoch [30/50], Test Accuracy: 71.77%
Epoch [30/50], Balanced Accuracy: 0.50
Epoch [30/50], F1-Score: 0.60
Epoch [31/50], Training Loss: 0.6089
Epoch [31/50], Test Accuracy: 70.53%
Epoch [31/50], Balanced Accuracy: 0.50
Epoch [31/50], F1-Score: 0.61
Epoch [32/50], Training Loss: 0.6035
Epoch [32/50], Test Accuracy: 68.05%
Epoch [32/50], Balanced Accuracy: 0.50
Epoch [32/50], F1-Score: 0.62
Epoch [33/50], Training Loss: 0.6137
Epoch [33/50], Test Accuracy: 71.77%
Epoch [33/50], Balanced Accuracy: 0.50
Epoch [33/50], F1-Score: 0.60
Epoch [34/50], Training Loss: 0.6096
Epoch [34/50], Test Accuracy: 71.15%
Epoch [34/50], Balanced Accuracy: 0.50
Epoch [34/50], F1-Score: 0.60
Epoch [35/50], Training Loss: 0.6093
Epoch [35/50], Test Accuracy: 70.42%
Epoch [35/50], Balanced Accuracy: 0.50
Epoch [35/50], F1-Score: 0.61
Epoch [36/50], Training Loss: 0.6066
Epoch [36/50], Test Accuracy: 71.04%
Epoch [36/50], Balanced Accuracy: 0.50
```

```
Epoch [36/50], F1-Score: 0.61
Epoch [37/50], Training Loss: 0.6153
Epoch [37/50], Test Accuracy: 71.66%
Epoch [37/50], Balanced Accuracy: 0.50
Epoch [37/50], F1-Score: 0.60
Epoch [38/50], Training Loss: 0.6070
Epoch [38/50], Test Accuracy: 70.94%
Epoch [38/50], Balanced Accuracy: 0.50
Epoch [38/50], F1-Score: 0.61
Epoch [39/50], Training Loss: 0.6027
Epoch [39/50], Test Accuracy: 70.22%
Epoch [39/50], Balanced Accuracy: 0.49
Epoch [39/50], F1-Score: 0.60
Epoch [40/50], Training Loss: 0.6058
Epoch [40/50], Test Accuracy: 68.56%
Epoch [40/50], Balanced Accuracy: 0.50
Epoch [40/50], F1-Score: 0.62
Epoch [41/50], Training Loss: 0.6028
Epoch [41/50], Test Accuracy: 70.42%
Epoch [41/50], Balanced Accuracy: 0.51
Epoch [41/50], F1-Score: 0.62
Epoch [42/50], Training Loss: 0.6022
Epoch [42/50], Test Accuracy: 71.04%
Epoch [42/50], Balanced Accuracy: 0.50
Epoch [42/50], F1-Score: 0.60
Epoch [43/50], Training Loss: 0.6039
Epoch [43/50], Test Accuracy: 71.46%
Epoch [43/50], Balanced Accuracy: 0.51
Epoch [43/50], F1-Score: 0.61
Epoch [44/50], Training Loss: 0.6031
Epoch [44/50], Test Accuracy: 71.77%
Epoch [44/50], Balanced Accuracy: 0.50
Epoch [44/50], F1-Score: 0.60
Epoch [45/50], Training Loss: 0.6071
Epoch [45/50], Test Accuracy: 71.15%
Epoch [45/50], Balanced Accuracy: 0.51
Epoch [45/50], F1-Score: 0.62
Epoch [46/50], Training Loss: 0.6058
Epoch [46/50], Test Accuracy: 71.56%
Epoch [46/50], Balanced Accuracy: 0.50
Epoch [46/50], F1-Score: 0.61
Epoch [47/50], Training Loss: 0.6108
Epoch [47/50], Test Accuracy: 71.46%
Epoch [47/50], Balanced Accuracy: 0.50
Epoch [47/50], F1-Score: 0.60
Epoch [48/50], Training Loss: 0.6062
Epoch [48/50], Test Accuracy: 69.91%
Epoch [48/50], Balanced Accuracy: 0.50
Epoch [48/50], F1-Score: 0.61
Epoch [49/50], Training Loss: 0.6083
Epoch [49/50], Test Accuracy: 71.15%
Epoch [49/50], Balanced Accuracy: 0.50
Epoch [49/50], F1-Score: 0.61
Epoch [50/50], Training Loss: 0.6098
Epoch [50/50], Test Accuracy: 71.35%
Epoch [50/50], Balanced Accuracy: 0.50
Epoch [50/50], F1-Score: 0.60
```

B. During Task Data

```
# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')
%cd /content/drive/MyDrive/
```

```
Mounted at /content/drive
/content/drive/MyDrive
```

```
# For output file path
import os

# Path to the folder containing EEG for Mental
    Arithmetic Tasks dataset files in Google Drive
edf_folder_path = '/content/drive/MyDrive/eeg-mental
    -arithmetic/'

# Import packages
import pandas as pd
import pyedflib # For reading EEG data
import numpy as np
from torch.utils.data import DataLoader,
    TensorDataset # For dataloader when about to
    train SNN
import snntorch as snn # SNN package
import torch
# Import MNE packages
import mne # Reading EEG data
import mne.viz # EEG data visualizations
from mne.io import concatenate_raws, read_raw_edf #
    Reading in raw EDFs
from mne.channels import make_standard_montage # EDF
     montage

# Load the labels from the CSV file into a
    dictionary for matching with edf files
csv_path = '/content/drive/MyDrive/eeg-mental-
    arithmetic/subject-info.csv'
labels_df = pd.read_csv(csv_path)
# Look at labels for each subject
display(labels_df)
```

	Subject	Age	Gender	Recording year	Number of subtractions	Count quality
0	Subject00	21	F	2011	9.70	0
1	Subject01	18	F	2011	29.35	1
2	Subject02	19	F	2012	12.88	1
3	Subject03	17	F	2010	31.00	1
4	Subject04	17	F	2010	8.60	0
5	Subject05	16	F	2010	20.71	1
6	Subject06	18	M	2011	4.35	0
7	Subject07	18	F	2012	13.38	1
8	Subject08	26	M	2011	18.24	1
9	Subject09	16	F	2010	7.00	0
10	Subject10	17	F	2010	1.00	0
11	Subject11	18	F	2010	26.00	1
12	Subject12	17	F	2010	26.36	1
13	Subject13	24	M	2012	34.00	1
14	Subject14	17	F	2010	9.00	0
15	Subject15	17	F	2012	22.18	1
16	Subject16	17	F	2010	11.59	1
17	Subject17	17	F	2010	28.70	1
18	Subject18	17	F	2010	20.00	1

```
# Import all _2 edf files (during task), manually
    import files corresponding to each group

# Separate into bad quality count (label 0) and good
     quality count (label 1)
bad_count_files = ['Subject00_2.edf', 'Subject04_2.
    edf', 'Subject06_2.edf', 'Subject09_2.edf',
                   'Subject10_2.edf', 'Subject14_2.
                       edf', 'Subject19_2.edf', '
                       Subject21_2.edf',
                   'Subject22_2.edf', 'Subject30_2.
                       edf'] # 12 bad count quality
                       subjects
good_count_files = ['Subject01_2.edf', 'Subject02_2.
    edf', 'Subject03_2.edf', 'Subject05_2.edf',
                    'Subject07_2.edf', 'Subject08_2.
                        edf', 'Subject11_2.edf', '
                        Subject12_2.edf',
                    'Subject13_2.edf', 'Subject15_2.
                        edf', 'Subject16_2.edf', '
                        Subject17_2.edf',
                    'Subject18_2.edf', 'Subject20_2.
                        edf', 'Subject23_2.edf', '
                        Subject24_2.edf',
                    'Subject25_2.edf', 'Subject26_2.
                        edf', 'Subject27_2.edf', '
                        Subject28_2.edf',
                    'Subject29_2.edf', 'Subject31_2.
                        edf', 'Subject32_2.edf', '
                        Subject33_2.edf',
                    'Subject34_2.edf', 'Subject35_2.
                        edf'] # 24 good count
                        quality subjects

# Initialize lists to store raw objects for bad
    count and good count quality
raw_good_list = []
raw_bad_list = []

# Function to load edf files into a raw object
def load_edf_files(edf_files, folder_path):
    raw_list = []
    for edf_file in edf_files:
        edf_path = os.path.join(folder_path,
            edf_file) # Find folder path and edf
            file

        # Check if the file exists before proceeding
        if not os.path.exists(edf_path):
            print(f"File {edf_file} not found!")
            continue

        # Load the edf file into mne
        try:
            raw = mne.io.read_raw_edf(edf_path,
                preload=True)
            raw_list.append(raw)  # Append to the
                list of raw objects
            # Print whether import was successful
            print(f"Successfully loaded {edf_file}
                with {len(raw.ch_names)} channels.")
        except Exception as e:
            print(f"Error loading {edf_file}: {e}")
    return raw_list

# Load good and bad count quality files separately
    using function
raw_good_list = load_edf_files(good_count_files,
    edf_folder_path)
raw_bad_list = load_edf_files(bad_count_files,
    edf_folder_path)

# Concatenate raws for good and bad count conditions
     separately
raw_good = mne.concatenate_raws(raw_good_list) # Raw
     good count quality object
raw_bad = mne.concatenate_raws(raw_bad_list) # Raw
    bad count quality object
```

```
Extracting EDF parameters from /content/drive/MyDrive/eeg-mental-arithmetic/Subject01_2.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 30999  =      0.000 ...    61.998 secs...
Successfully loaded Subject01_2.edf with 21 channels.
Extracting EDF parameters from /content/drive/MyDrive/eeg-mental-arithmetic/Subject02_2.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 30999  =      0.000 ...    61.998 secs...
Successfully loaded Subject02_2.edf with 21 channels.
Extracting EDF parameters from /content/drive/MyDrive/eeg-mental-arithmetic/Subject03_2.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 30999  =      0.000 ...    61.998 secs...
Successfully loaded Subject03_2.edf with 21 channels.
Extracting EDF parameters from /content/drive/MyDrive/eeg-mental-arithmetic/Subject05_2.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 30999  =      0.000 ...    61.998 secs...
Successfully loaded Subject05_2.edf with 21 channels.
Extracting EDF parameters from /content/drive/MyDrive/eeg-mental-arithmetic/Subject07_2.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 30999  =      0.000 ...    61.998 secs...
Successfully loaded Subject07_2.edf with 21 channels.
Extracting EDF parameters from /content/drive/MyDrive/eeg-mental-arithmetic/Subject08_2.edf...
EDF file detected
Setting channel info structure...
Creating raw.info structure...
Reading 0 ... 30999  =      0.000 ...    61.998 secs...
Successfully loaded Subject08_2.edf with 21 channels.
Extracting EDF parameters from /content/drive/MyDrive/eeg-mental-arithmetic/Subject11_2.edf...
EDF file detected
```

```
# Set the channel locations and types for both raw
    objects
for raw in [raw_good, raw_bad]:
    raw.rename_channels({'EEG Fp1':'Fp1','EEG Fp2':'
        Fp2','EEG F3':'F3','EEG F4':'F4','EEG F7':'
        F7','EEG F8':'F8',
                         'EEG T3':'T3','EEG T4':'T4
                             ','EEG C3':'C3','EEG C4
                             ':'C4','EEG T5':'T5','
                             EEG T6':'T6','EEG P3':'
                             P3',
                         'EEG P4':'P4','EEG O1':'O1
                             ','EEG O2':'O2','EEG Fz
                             ':'Fz','EEG Cz':'Cz','
                             EEG Pz':'Pz',
                         'EEG A2-A1':'A2','ECG ECG
                             ':'ECG'})
    raw.set_channel_types({'ECG': 'ecg'})
    raw.set_montage(mne.channels.
        make_standard_montage('standard_1020'))  #
        Set montage

# Get data from raw good count object
raw_data_1 = raw_good.get_data()
print("Number of channels: ", str(len(raw_data_1)))
    # print channels
print("Number of samples: ", str(len(raw_data_1[0]))
    ) # print samples
# Get Sampling Frequency
fs_g=raw_good.info['sfreq']
print("Sampling Frequency: ", str(fs_g))# Get data
    from raw task object
```

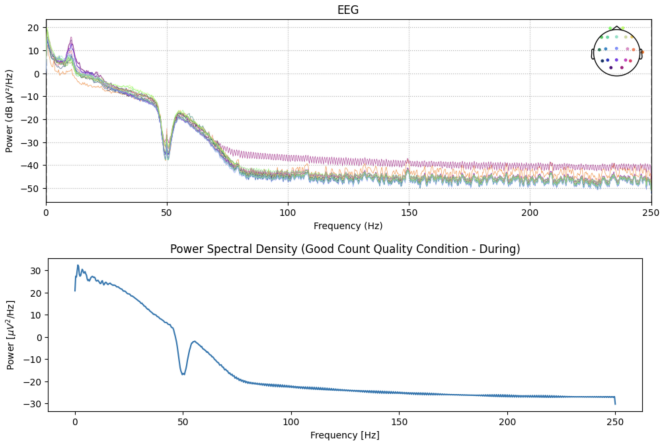
```
Number of channels:  21
Number of samples:  806000
Sampling Frequency:  500.0
```

```
# Get data from raw bad count object
raw_data_2 = raw_bad.get_data()
print("Number of channels: ", str(len(raw_data_2)))
    # print channels
```

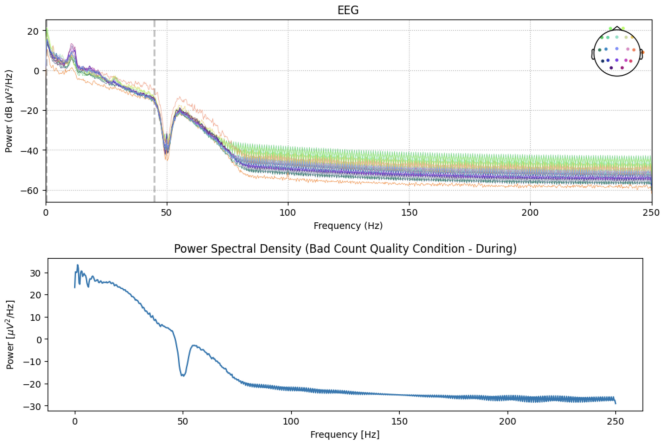
```
print("Number of samples: ", str(len(raw_data_2[0]))
    ) # print samples
# Get Sampling Frequency
fs_b=raw_bad.info['sfreq']
print("Sampling Frequency: ", str(fs_b))
```

```
Number of channels:  21
Number of samples:  310000
Sampling Frequency:  500.0
```

```
# Import more packages
%matplotlib inline
import scipy as sp
from scipy import stats
import matplotlib.pyplot as plt
import copy

# Plot_psd function for plotting raw good count
    object Power Spectral Density (PSD)
raw_good.plot_psd()
# Alternative scipy plot with one line instead of
    multiple channels
f, Sxx = sp.signal.welch(raw_good._data[20,:], fs=
    fs_g, window='hann'),
nperseg=1500, noverlap=1000, scaling='density')
# The scale of Sxx use scaling='density' is V**2/Hz
# To change the scale to muV**2/Hz(Db) which mne
    uses, use the following conversion
plt.figure(figsize=(11.6,3))
plt.plot(f, 10*np.log10(Sxx*1e6**2))
plt.xlabel('Frequency [Hz]')
plt.ylabel('Power [$\mu V^2$/Hz]')
plt.title('Power Spectral Density (Good Count
    Quality Condition - During)')
plt.show()
```




```
# Plot_psd function for plotting raw bad count
    object Power Spectral Density (PSD)
raw_bad.plot_psd()
# Alternative scipy plot with one line instead of
    multiple channels
f, Sxx = sp.signal.welch(raw_bad._data[20,:], fs=
    fs_b, window='hann'),
nperseg=1500, noverlap=1000, scaling='density')
# The scale of Sxx use scaling='density' is V**2/Hz
# To change the scale to muV**2/Hz(Db) which mne
    uses, use the following conversion
plt.figure(figsize=(11.6,3))
plt.plot(f, 10*np.log10(Sxx*1e6**2))
plt.xlabel('Frequency [Hz]')
plt.ylabel('Power [$\mu V^2$/Hz]')
plt.title('Power Spectral Density (Bad Count Quality
    Condition - During)')
plt.show()
```




```
# Check how many seconds are in good data to
    determine time windows for topomap
total_duration_seconds = raw_good.times[-1]
print(f'Total duration: {total_duration_seconds}
    seconds')
```

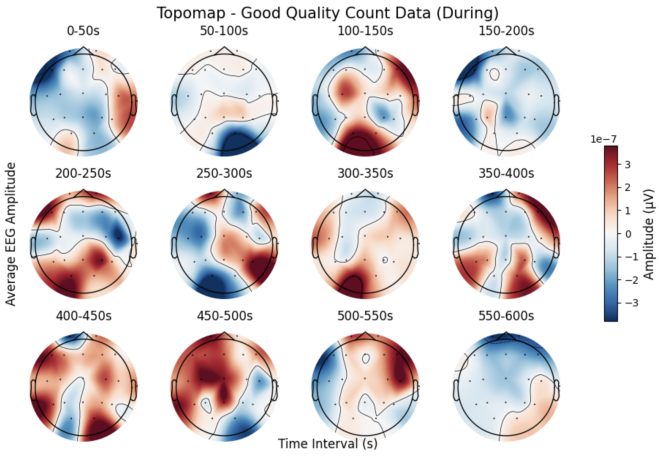
```
Total duration: 1611.998 seconds
```

```
# Check how many seconds are in bad data to
    determine time windows for topomap
total_duration_seconds = raw_bad.times[-1]
print(f'Total duration: {total_duration_seconds}
    seconds')
```

```
Total duration: 619.998 seconds
```

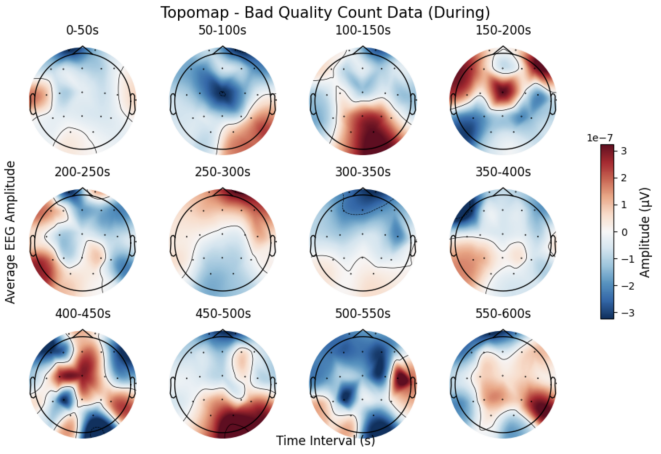
```
# Define time intervals in seconds
time_intervals = [(0, 50), (50, 100), (100, 150),
    (150, 200), (200, 250), (250, 300), (300, 350),
    (350, 400), (400, 450), (450, 500), (500, 550),
    (550, 600)]

# Create a 3-row x 4-column subplot grid for 12 time
     windows
fig, axes = plt.subplots(3, 4, figsize=(9, 6))

# Flatten axes array for easier iteration
axes = axes.flatten()

# Create colorbar (for shared colorbar)
cbar_ax = fig.add_axes([0.92, 0.3, 0.02, 0.4])  # [
    left, bottom, width, height] placement

# Iterate over the time intervals and plot the
    topomaps
for i, (start, end) in enumerate(time_intervals):
    # Extract the data for the current time interval
    start_sample = int(start * raw_good.info['sfreq
        '])  # Convert time to samples
    end_sample = int(end * raw_good.info['sfreq'])
        # Convert time to samples
    interval_data = raw_good.get_data(picks='eeg',
        start=start_sample, stop=end_sample)  #
        Select 'eeg' channels
```

```
    # Compute the average signal over the interval
    avg_data = np.mean(interval_data, axis=1)  #
        Mean across time points for each channel

    # Plot the topomap for the current time interval
         in the corresponding subplot
    im, _ = mne.viz.plot_topomap(
        avg_data,
        pos=raw_good.info,
        ch_type='eeg',
        axes=axes[i],
        show=False,
        contours=1  # Show contours
    )
    axes[i].set_title(f'{start}-{end}s')  # Add
        topomap label for time window

# Add colorbar
cbar = fig.colorbar(im, cax=cbar_ax)
cbar.set_label('Amplitude ( V )', fontsize=12)

# Add axis labels and title to entire figure
fig.text(0.5, 0.01, 'Time Interval (s)', ha='center
    ', fontsize=12)
fig.text(0.01, 0.5, 'Average EEG Amplitude', va='
    center', rotation='vertical', fontsize=12)
fig.text(0.5, 0.99, 'Topomap - Good Quality Count
    Data (Before)', ha='center', fontsize=15)

# Adjust layout
plt.tight_layout(rect=[0, 0, 0.9, 1])  # Leave space
     for colorbar
plt.show()
```



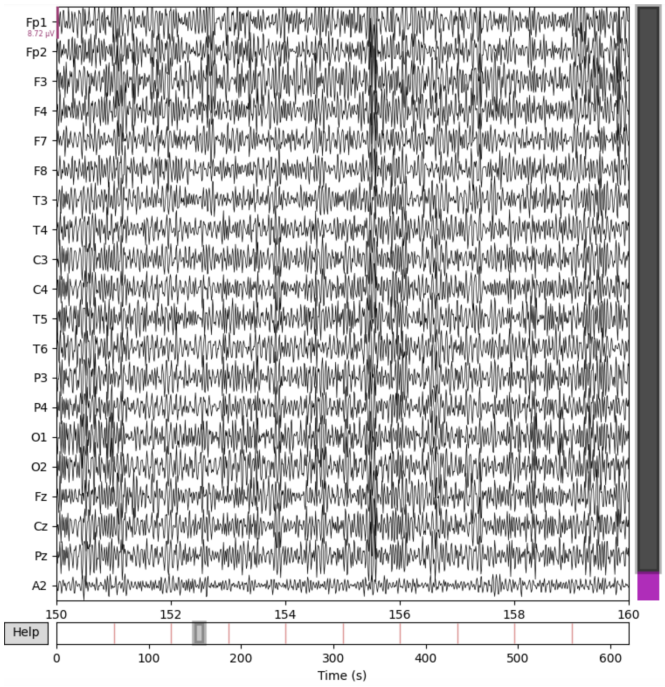

```
# Define time intervals in seconds
time_intervals = [(0, 50), (50, 100), (100, 150),
    (150, 200), (200, 250), (250, 300), (300, 350),
    (350, 400), (400, 450), (450, 500), (500, 550),
    (550, 600)]

# Create a 3-row x 4-column subplot grid for 12 time
     windows
fig, axes = plt.subplots(3, 4, figsize=(9, 6))

# Flatten axes array for easier iteration
axes = axes.flatten()

# Create common colorbar axis
cbar_ax = fig.add_axes([0.92, 0.3, 0.02, 0.4])  # [
    left, bottom, width, height]
```

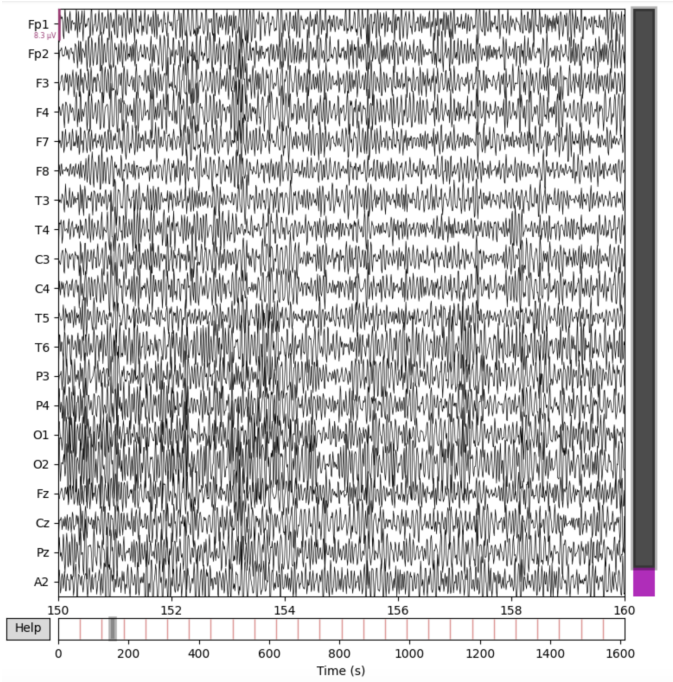
```
# Iterate over the time intervals and plot the
    topomaps
for i, (start, end) in enumerate(time_intervals):
    # Extract the data for the current time interval
    start_sample = int(start * raw_bad.info['sfreq
        '])  # Convert time to samples
    end_sample = int(end * raw_bad.info['sfreq'])  #
         Convert time to samples
    interval_data = raw_bad.get_data(picks='eeg',
        start=start_sample, stop=end_sample)  #
        Select 'eeg' channels

    # Compute the average signal over the interval
    avg_data = np.mean(interval_data, axis=1)  #
        Mean across time points for each channel

    # Plot the topomap for the current time interval
         in the corresponding subplot
    im, _ = mne.viz.plot_topomap(
        avg_data,
        pos=raw_bad.info,
        ch_type='eeg',
        axes=axes[i],
        show=False,
        contours=1  # Show contours
    )
    axes[i].set_title(f'{start}-{end}s')  # Add
        topomap label for time window

# Add colorbar
cbar = fig.colorbar(im, cax=cbar_ax)
cbar.set_label('Amplitude ( V )', fontsize=12)

# Add axis labels and title to entire figure
fig.text(0.5, 0.01, 'Time Interval (s)', ha='center
    ', fontsize=12)
fig.text(0.01, 0.5, 'Average EEG Amplitude', va='
    center', rotation='vertical', fontsize=12)
fig.text(0.5, 0.99, 'Topomap - Bad Quality Count
    Data (Before)', ha='center', fontsize=15)

# Adjust layout
plt.tight_layout(rect=[0, 0, 0.9, 1])  # Leave space
     for the colorbar
plt.show()
```




```
# Plot the time-domain signal for raw_good between
    beta band (common duing mental activity)
raw_good.filter(13., 30.) # Filter between beta band
     (13-30 Hz)
# Plot the time-domain signal within time window
```

```
raw_good.plot(scalings='auto', title='Time-Domain
    Signal: Good Count Quality EEG (During)',
                  start=150,  # Start time in
                      seconds
                  duration=10, # Duration of time
                      window
                  show=True)
```






```
# Extract data and sampling frequency from the raw
    good and bad count quality objects
data_good, _ = raw_good[:, :]
data_bad, _ = raw_bad[:, :]
fs_g = int(raw_good.info['sfreq'])
fs_b = int(raw_bad.info['sfreq'])
print(fs_g, fs_b) # Print sampling frequencies of
    the raw objects
```

```
500 500
```

```
# Plot the time-domain signal for raw_bad between
    beta band (common during mental activity)
raw_bad.filter(13., 30.) # Filter between beta band
    (13-30 Hz)
# Plot the time-domain signal within time window
raw_bad.plot(scalings='auto', title='Time-Domain
    Signal: Bad Count Quality EEG (During)',
                  start=150,  # Start time in
                      seconds
                  duration=10, # Duration of time
                      window
                  show=True)
```

```
# Set segment size to 500 sample data points for
    splitting
window_size = 500

# Segment the data into non-overlapping windows
def segment_data(data, window_size):
    num_segments = data.shape[1] // window_size #
        Split into 500-sample point segments
    segments = np.array_split(data[:, :num_segments
        * window_size], num_segments, axis=1) #
        Convert into numpy array
    return segments

# Segment the good and bad data
segments_good = segment_data(data_good, window_size)
segments_bad = segment_data(data_bad, window_size)

# Label the segments as 1 for good, 0 for bad as in
    csv
labels_good = np.ones(len(segments_good))
labels_bad = np.zeros(len(segments_bad))

# Combine the segments and labels into numpy dataset
segments = np.array(segments_good + segments_bad)  #
     All segmented data
labels = np.concatenate([labels_good, labels_bad])
    # All label data

from sklearn.metrics import accuracy_score,
    balanced_accuracy_score, f1_score # Evaluation
    metrics
```

```
# Print shape of all segments and labels
print(np.shape(segments))
print(np.shape(labels))
# Change dimension of data for ML Algorithm
X_all=np.mean(segments,axis=1)
Y_all=labels
X_all=sp.stats.zscore(X_all,axis=1)
print(np.shape(X_all))
print(np.shape(Y_all))
# Set to X and Y for training
X=X_all
Y=Y_all
print(np.shape(X))
print(np.shape(Y))
# Check for NaN values in X and Y
print(np.isnan(X).any())
print(np.isnan(Y).any())
```

```
(2232, 21, 500)
(2232,)
(2232, 500)
(2232,)
(2232, 500)
(2232,)
False
False
```

```
# Since there are NaN values in X (sometimes there
    are no NaN values), impute using SimpleImputer
from sklearn.impute import SimpleImputer

# Handle NaN values using SimpleImputer
imputer = SimpleImputer(strategy='mean')  # Replace
    NaNs with the mean of each column
X = imputer.fit_transform(X) # Set X to newly
    imputed missing values

# Take counts of Y to see imbalance of labels
unique, counts = np.unique(Y, return_counts=True)
print(dict(zip(unique, counts)))
```

```
{0.0: 620, 1.0: 1612}
```

```
# Confirm shape of X and Y again
print(X.shape)
print(Y.shape)
```

```
(2232, 500)
(2232,)
```

```
# Import train_test_split
from sklearn.model_selection import train_test_split

# Split the data into training and testing sets -
    70% training and 30% testing
X_train, X_test, Y_train, Y_test = train_test_split(
    X, Y, test_size=0.3, random_state=42, stratify=Y
    )

# Check shapes
print("Training set shape:", X_train.shape, Y_train.
    shape)
print("Testing set shape:", X_test.shape, Y_test.
    shape)
```

```
Training set shape: (1562, 500) (1562,)
Testing set shape: (670, 500) (670,)
```

```
# Convert data to PyTorch tensors
X_train_tensor = torch.FloatTensor(X_train)
Y_train_tensor = torch.LongTensor(Y_train)
X_test_tensor = torch.FloatTensor(X_test)
Y_test_tensor = torch.LongTensor(Y_test)

# Create TensorDataset objects for training and
    testing
train_dataset = TensorDataset(X_train_tensor,
    Y_train_tensor)
test_dataset = TensorDataset(X_test_tensor,
    Y_test_tensor)

# Define DataLoader batch size
batch_size = 32

# Create DataLoaders for training and testing sets
train_loader = DataLoader(train_dataset, batch_size=
    batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=
    batch_size, shuffle=False)

# Check if DataLoader works correctly by iterating
    over a batch
for X_batch, Y_batch in train_loader:
    print("X_batch shape:", X_batch.shape)
    print("Y_batch shape:", Y_batch.shape)
    break  # Stop showing after 1 batch
```

```
X_batch shape: torch.Size([32, 500])
Y_batch shape: torch.Size([32])
```

```
# Import torch.nn
import torch.nn as nn

# Define the SNN model architecture
class SNNModel(nn.Module):
    def __init__(self, input_size):
        super(SNNModel, self).__init__()
        self.fc1 = nn.Linear(input_size, 100)  #
            Fully connected layer with 100 neurons
        self.lif1 = snn.Leaky(beta=0.9)  # LIF
            neuron with snn.Leaky to handle spikes
        self.fc2 = nn.Linear(100, 2)  # Output layer
             for binary classification

    def forward(self, x):
        # Pass through the first fully connected
            layer
        x = self.fc1(x)

        # Process with the LIF layer and generate
            spikes
        mem, spikes = self.lif1(x)

        # Pass through the second fully connected
            layer
        output = self.fc2(spikes)

        return output

# Initialize network, optimizer, and loss function
input_size = X_train.shape[1]
model = SNNModel(input_size)
optimizer = torch.optim.Adam(model.parameters(), lr
    =0.001)
criterion = nn.CrossEntropyLoss()

# Training loop
num_epochs = 50
```

```
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0  # Track loss for each epoch

    # Iterate over batches from the training loader
    for X_batch, Y_batch in train_loader:
        optimizer.zero_grad()  # Zero the gradients

        # Forward pass
        output = model(X_batch)
        loss = criterion(output, Y_batch)

        # Backward pass and optimization
        loss.backward(retain_graph=True)
        optimizer.step()

        running_loss += loss.item() * X_batch.size
            (0)  # Accumulate loss

    # Calculate average loss for the epoch
    epoch_loss = running_loss / len(train_loader.
        dataset)

    # Print training loss every epoch
    if (epoch + 1) % 1 == 0:
        print(f'Epoch [{epoch + 1}/{num_epochs}],
            Training Loss: {epoch_loss:.4f}')

    # Evaluation on test data
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for X_batch, Y_batch in test_loader:
            outputs = model(X_batch) # evaluates
                based on batch in X
            _, predicted = torch.max(outputs.data,
                1) # takes maximum score

            all_preds.extend(predicted.cpu().numpy()
                ) # appends to all_preds
            all_labels.extend(Y_batch.cpu().numpy())
                 # appends to all_labels

    # Calculate Accuracy, Balanced Accuracy, and F1-
        Score
    accuracy = accuracy_score(all_labels, all_preds)
    balanced_accuracy = balanced_accuracy_score(
        all_labels, all_preds)
    f1 = f1_score(all_labels, all_preds, average='
        weighted')  # 'weighted' accounts for class
        imbalance

    # Print evaluation metrics
    print(f'Epoch [{epoch + 1}/{num_epochs}], Test
        Accuracy: {accuracy*100:.2f}%')
    print(f'Epoch [{epoch + 1}/{num_epochs}],
        Balanced Accuracy: {balanced_accuracy:.2f}')
    print(f'Epoch [{epoch + 1}/{num_epochs}], F1-
        Score: {f1:.2f}')
```

```
Epoch [1/50], Training Loss: 0.8998
Epoch [1/50], Test Accuracy: 57.61%
Epoch [1/50], Balanced Accuracy: 0.49
Epoch [1/50], F1-Score: 0.58
Epoch [2/50], Training Loss: 0.6427
Epoch [2/50], Test Accuracy: 65.97%
Epoch [2/50], Balanced Accuracy: 0.50
Epoch [2/50], F1-Score: 0.62
Epoch [3/50], Training Loss: 0.6100
Epoch [3/50], Test Accuracy: 65.97%
Epoch [3/50], Balanced Accuracy: 0.48
Epoch [3/50], F1-Score: 0.60
Epoch [4/50], Training Loss: 0.6124
Epoch [4/50], Test Accuracy: 69.40%
Epoch [4/50], Balanced Accuracy: 0.51
Epoch [4/50], F1-Score: 0.63
Epoch [5/50], Training Loss: 0.6032
Epoch [5/50], Test Accuracy: 69.10%
Epoch [5/50], Balanced Accuracy: 0.50
Epoch [5/50], F1-Score: 0.62
Epoch [6/50], Training Loss: 0.6108
Epoch [6/50], Test Accuracy: 67.76%
Epoch [6/50], Balanced Accuracy: 0.50
Epoch [6/50], F1-Score: 0.62
Epoch [7/50], Training Loss: 0.6103
Epoch [7/50], Test Accuracy: 68.06%
Epoch [7/50], Balanced Accuracy: 0.49
Epoch [7/50], F1-Score: 0.61
Epoch [8/50], Training Loss: 0.6138
Epoch [8/50], Test Accuracy: 68.51%
Epoch [8/50], Balanced Accuracy: 0.49
Epoch [8/50], F1-Score: 0.61
Epoch [9/50], Training Loss: 0.6017
Epoch [9/50], Test Accuracy: 69.40%
Epoch [9/50], Balanced Accuracy: 0.50
Epoch [9/50], F1-Score: 0.62
Epoch [10/50], Training Loss: 0.6090
Epoch [10/50], Test Accuracy: 70.45%
Epoch [10/50], Balanced Accuracy: 0.50
Epoch [10/50], F1-Score: 0.61
Epoch [11/50], Training Loss: 0.6035
Epoch [11/50], Test Accuracy: 72.09%
Epoch [11/50], Balanced Accuracy: 0.50
Epoch [11/50], F1-Score: 0.61
Epoch [12/50], Training Loss: 0.6081
Epoch [12/50], Test Accuracy: 71.04%
Epoch [12/50], Balanced Accuracy: 0.49
Epoch [12/50], F1-Score: 0.60
Epoch [13/50], Training Loss: 0.6009
Epoch [13/50], Test Accuracy: 71.49%
Epoch [13/50], Balanced Accuracy: 0.50
Epoch [13/50], F1-Score: 0.61
Epoch [14/50], Training Loss: 0.5958
Epoch [14/50], Test Accuracy: 71.94%
Epoch [14/50], Balanced Accuracy: 0.50
Epoch [14/50], F1-Score: 0.60
Epoch [15/50], Training Loss: 0.6025
Epoch [15/50], Test Accuracy: 71.94%
Epoch [15/50], Balanced Accuracy: 0.50
Epoch [15/50], F1-Score: 0.61
Epoch [16/50], Training Loss: 0.5972
Epoch [16/50], Test Accuracy: 71.04%
Epoch [16/50], Balanced Accuracy: 0.49
Epoch [16/50], F1-Score: 0.60
Epoch [17/50], Training Loss: 0.6079
```

```
Epoch [17/50], Test Accuracy: 72.24%
Epoch [17/50], Balanced Accuracy: 0.50
Epoch [17/50], F1-Score: 0.61
Epoch [18/50], Training Loss: 0.5930
Epoch [18/50], Test Accuracy: 71.79%
Epoch [18/50], Balanced Accuracy: 0.50
Epoch [18/50], F1-Score: 0.61
Epoch [19/50], Training Loss: 0.5983
Epoch [19/50], Test Accuracy: 72.09%
Epoch [19/50], Balanced Accuracy: 0.50
Epoch [19/50], F1-Score: 0.61
Epoch [20/50], Training Loss: 0.5920
Epoch [20/50], Test Accuracy: 71.34%
Epoch [20/50], Balanced Accuracy: 0.50
Epoch [20/50], F1-Score: 0.61
Epoch [21/50], Training Loss: 0.6017
Epoch [21/50], Test Accuracy: 72.09%
Epoch [21/50], Balanced Accuracy: 0.50
Epoch [21/50], F1-Score: 0.61
Epoch [22/50], Training Loss: 0.6047
Epoch [22/50], Test Accuracy: 70.30%
Epoch [22/50], Balanced Accuracy: 0.51
Epoch [22/50], F1-Score: 0.62
Epoch [23/50], Training Loss: 0.6000
Epoch [23/50], Test Accuracy: 71.94%
Epoch [23/50], Balanced Accuracy: 0.50
Epoch [23/50], F1-Score: 0.60
Epoch [24/50], Training Loss: 0.6053
Epoch [24/50], Test Accuracy: 72.09%
Epoch [24/50], Balanced Accuracy: 0.50
Epoch [24/50], F1-Score: 0.61
Epoch [25/50], Training Loss: 0.5999
Epoch [25/50], Test Accuracy: 72.24%
Epoch [25/50], Balanced Accuracy: 0.50
Epoch [25/50], F1-Score: 0.61
Epoch [26/50], Training Loss: 0.5991
Epoch [26/50], Test Accuracy: 72.24%
Epoch [26/50], Balanced Accuracy: 0.50
Epoch [26/50], F1-Score: 0.61
Epoch [27/50], Training Loss: 0.6000
Epoch [27/50], Test Accuracy: 72.24%
Epoch [27/50], Balanced Accuracy: 0.50
Epoch [27/50], F1-Score: 0.61
Epoch [28/50], Training Loss: 0.5967
Epoch [28/50], Test Accuracy: 72.24%
Epoch [28/50], Balanced Accuracy: 0.50
Epoch [28/50], F1-Score: 0.61
Epoch [29/50], Training Loss: 0.5968
Epoch [29/50], Test Accuracy: 72.09%
Epoch [29/50], Balanced Accuracy: 0.50
Epoch [29/50], F1-Score: 0.61
Epoch [30/50], Training Loss: 0.5973
Epoch [30/50], Test Accuracy: 72.24%
Epoch [30/50], Balanced Accuracy: 0.50
Epoch [30/50], F1-Score: 0.61
Epoch [31/50], Training Loss: 0.5976
Epoch [31/50], Test Accuracy: 72.24%
Epoch [31/50], Balanced Accuracy: 0.50
Epoch [31/50], F1-Score: 0.61
Epoch [32/50], Training Loss: 0.5914
Epoch [32/50], Test Accuracy: 72.39%
Epoch [32/50], Balanced Accuracy: 0.50
Epoch [32/50], F1-Score: 0.61
Epoch [33/50], Training Loss: 0.5928
Epoch [33/50], Test Accuracy: 71.79%
Epoch [33/50], Balanced Accuracy: 0.50
Epoch [33/50], F1-Score: 0.61
Epoch [34/50], Training Loss: 0.5994
Epoch [34/50], Test Accuracy: 72.39%
Epoch [34/50], Balanced Accuracy: 0.50
Epoch [34/50], F1-Score: 0.61
Epoch [35/50], Training Loss: 0.5932
Epoch [35/50], Test Accuracy: 71.94%
Epoch [35/50], Balanced Accuracy: 0.50
Epoch [35/50], F1-Score: 0.60
Epoch [36/50], Training Loss: 0.6007
Epoch [36/50], Test Accuracy: 72.24%
Epoch [36/50], Balanced Accuracy: 0.50
Epoch [36/50], F1-Score: 0.61
Epoch [37/50], Training Loss: 0.5953
Epoch [37/50], Test Accuracy: 72.09%
Epoch [37/50], Balanced Accuracy: 0.50
Epoch [37/50], F1-Score: 0.61
Epoch [38/50], Training Loss: 0.5970
Epoch [38/50], Test Accuracy: 71.94%
Epoch [38/50], Balanced Accuracy: 0.50
Epoch [38/50], F1-Score: 0.60
Epoch [39/50], Training Loss: 0.5965
Epoch [39/50], Test Accuracy: 71.64%
Epoch [39/50], Balanced Accuracy: 0.50
Epoch [39/50], F1-Score: 0.61
Epoch [40/50], Training Loss: 0.6024
Epoch [40/50], Test Accuracy: 72.24%
Epoch [40/50], Balanced Accuracy: 0.50
Epoch [40/50], F1-Score: 0.61
Epoch [41/50], Training Loss: 0.5987
Epoch [41/50], Test Accuracy: 72.09%
Epoch [41/50], Balanced Accuracy: 0.50
Epoch [41/50], F1-Score: 0.61
Epoch [42/50], Training Loss: 0.5949
Epoch [42/50], Test Accuracy: 72.24%
Epoch [42/50], Balanced Accuracy: 0.50
Epoch [42/50], F1-Score: 0.61
Epoch [43/50], Training Loss: 0.5910
Epoch [43/50], Test Accuracy: 72.24%
Epoch [43/50], Balanced Accuracy: 0.50
Epoch [43/50], F1-Score: 0.61
Epoch [44/50], Training Loss: 0.5948
Epoch [44/50], Test Accuracy: 72.09%
Epoch [44/50], Balanced Accuracy: 0.50
Epoch [44/50], F1-Score: 0.61
Epoch [45/50], Training Loss: 0.5980
```

```
Epoch [45/50], Test Accuracy: 71.94%
Epoch [45/50], Balanced Accuracy: 0.50
Epoch [45/50], F1-Score: 0.61
Epoch [46/50], Training Loss: 0.5900
Epoch [46/50], Test Accuracy: 72.24%
Epoch [46/50], Balanced Accuracy: 0.50
Epoch [46/50], F1-Score: 0.61
Epoch [47/50], Training Loss: 0.5971
Epoch [47/50], Test Accuracy: 72.39%
Epoch [47/50], Balanced Accuracy: 0.51
Epoch [47/50], F1-Score: 0.61
Epoch [48/50], Training Loss: 0.5994
Epoch [48/50], Test Accuracy: 72.24%
Epoch [48/50], Balanced Accuracy: 0.50
Epoch [48/50], F1-Score: 0.61
Epoch [49/50], Training Loss: 0.5933
Epoch [49/50], Test Accuracy: 71.79%
Epoch [49/50], Balanced Accuracy: 0.50
Epoch [49/50], F1-Score: 0.61
Epoch [50/50], Training Loss: 0.5939
Epoch [50/50], Test Accuracy: 72.09%
Epoch [50/50], Balanced Accuracy: 0.50
Epoch [50/50], F1-Score: 0.61
```