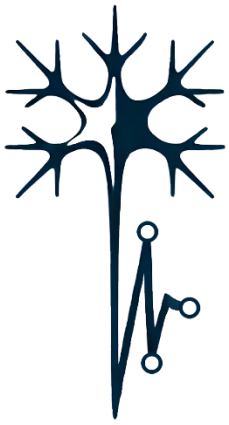


SPIKE
COMPUTATIONAL
NEUROSCIENCE
COMMUNITY

EEG Processing:

An Entry to the World of Brain Waves

This session:

Python – Numpy

Author: Mohammadreza Shahsavari

Contact: mohamadrezashahsavary@gmail.com

What you are going to learn in this session?

In this session, we'll dive into the basics of NumPy, a versatile library for scientific computing. We'll start by learning how to install NumPy, create arrays, and perform fundamental operations such as reshaping and slicing arrays.

Next, we'll explore how to load EEG data into Python. EEG data comes in various formats like EDF, BDF, and MAT, and you'll learn how to handle these using specialized libraries. This is crucial for preparing the data for analysis.

We'll also cover some basic exploration of EEG data, such as checking its shape and size and performing simple statistical analyses. You'll learn how to calculate the mean, median, and standard deviation, and how to handle any missing or NaN values in your data.

Afterward, we'll go through essential preprocessing steps for EEG data. This includes cleaning the data, filling in any gaps, and normalizing it using techniques like Min-Max scaling and Z-Normalization. These steps ensure your data is ready for more detailed analysis.

Additionally, we'll delve into calculating the correlations between EEG channels and creating functional connectivity maps. You'll learn how to measure the relationships between different brain regions and visualize these interactions through correlation maps.

By the end of this session, you'll have a solid grasp of using NumPy for EEG data processing, setting you up for more advanced studies in brainwave analysis. Enjoy the journey into the world of brain waves!

- *Mohammadreza Shahsavari*

1. Introduction to NumPy

Overview

Introduction to NumPy NumPy, which stands for Numerical Python, is a fundamental package for scientific computing in Python. It provides support for arrays, matrices, and many mathematical functions to operate on these data structures efficiently. NumPy is particularly useful for handling large datasets and performing complex mathematical operations.

Importance of NumPy in Scientific Computing and EEG Data Processing In the realm of scientific computing, NumPy is indispensable due to its performance and ease of use. When it comes to EEG data processing, NumPy's powerful array operations and numerical capabilities allow for efficient manipulation and analysis of the multi-dimensional data that EEG generates. By leveraging NumPy, researchers and data scientists can perform preprocessing, filtering, feature extraction, and more, facilitating the analysis of brain activity and aiding in neurological research.

Installing Numpy

Installing NumPy using pip Installing NumPy is straightforward using Python's package manager, pip. Open your command line interface and execute the following command:

```
>> pip install numpy
```

This command downloads and installs the latest version of NumPy from the Python Package Index (PyPI).

Importing NumPy in Python Once installed, you can import NumPy into your Python scripts using the following import statement:

```
import numpy as np
```

This convention (np) is widely used in the Python community and helps keep the code concise.

Basic NumPy Operations

Array Creation

Creating arrays from lists NumPy arrays can be created from Python lists, enabling efficient storage and manipulation of data.

```
import numpy as np

# Creating a 1D array from a list
arr1 = np.array([1, 2, 3, 4, 5])
print(arr1)
# Creating a 2D array (matrix) from a list of lists
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2)
```

Output:

```
[1 2 3 4 5]

[[1 2 3]
 [4 5 6]]
```

NumPy provides several other functions to create arrays directly, each serving different purposes. Below is a brief explanation of each method used in the provided code:

1. **Creating an array using np.array()**: Converts a list (or other sequences) into a NumPy array. It's a straightforward way to create an array when you have a list of values.
2. **Creating an array of zeros using np.zeros()**: Creates an array filled with zeros. The shape of the array is specified by a tuple of dimensions.
3. **Creating an array of ones using np.ones()**: Creates an array filled with ones. The shape of the array is specified by a tuple of dimensions.
4. **Creating an array with a range of values using np.arange()**: Creates an array with a sequence of values within a specified range. It's similar to Python's built-in range() but returns an array instead.
5. **Creating an array with linearly spaced values using np.linspace()**: Generates an array of evenly spaced values over a specified interval. The number of values is defined by the user.

```
# Creating an array using np.array()
arr = np.array([10, 20, 30])
print(arr)

# Creating an array of zeros
zeros = np.zeros((2, 3))
print(zeros)

# Creating an array of ones
ones = np.ones((3, 2))
print(ones)

# Creating an array with a range of values
range_arr = np.arange(0, 10, 2)
print(range_arr)

# Creating an array with linearly spaced values
linspace_arr = np.linspace(0, 1, 5)
print(linspace_arr)
```

Output:

```
[10 20 30]
```

```
[[0. 0. 0.]
 [0. 0. 0.]]

[[1. 1.]
 [1. 1.]
 [1. 1.]]

[0 2 4 6 8]

[0.   0.25 0.5  0.75 1.  ]
```

Array Manipulation

Reshaping arrays with **reshape()** Reshaping arrays allows for changing their shape without altering the data.

```
# Reshaping a 1D array into a 2D array
arr = np.arange(1, 7)
reshaped_arr = arr.reshape((2, 3))
print(reshaped_arr)
```

Output:

```
[[1 2 3]
 [4 5 6]]
```

Indexing and Slicing Arrays in NumPy

Indexing and slicing are essential techniques in NumPy that allow for accessing and modifying specific parts of an array. These operations are similar to those in standard Python lists, but they offer more functionality and efficiency, especially when working with multi-dimensional arrays.

Indexing

Indexing allows you to access individual elements of an array. You can index arrays in NumPy using square brackets `[]`.

```
import numpy as np

arr = np.array([10, 20, 30, 40, 50])

# Indexing
print(arr[2])
```

Output:

```
30
```

In the example above, `arr[2]` accesses the element at index 2, which is 30. Note that indexing in Python is zero-based, so the first element is at index 0.

Slicing

Slicing enables accessing a subarray or a subset of elements from the original array. The syntax for slicing is `start:stop:step`, where `start` is the starting index, `stop` is the ending index (exclusive), and `step` is the step size.

```
# Slicing
print(arr[1:4])
```

Output:

```
[20 30 40]
```

In the example above, `arr[1:4]` extracts a slice of the array from index 1 to index 3 (the element at index 4 is not included), resulting in the subarray `[20, 30, 40]`.

Examples with Multi-dimensional Arrays

Indexing and slicing become even more powerful when working with multi-dimensional arrays. Here are a few examples:

```
# Creating a 2D array (matrix)
matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Indexing a specific element
print(matrix[1, 2])

# Slicing a subarray
print(matrix[0:2, 1:3])
```

Output:

```
6

[[2 3]
 [5 6]]
```

In the first example with the matrix, matrix[1, 2] accesses the element in the second row and third column, which is 6. In the slicing example, matrix[0:2, 1:3] extracts a subarray that includes the first two rows and the second and third columns.

Combining and Stacking the Arrays

Combining arrays is essential for many operations in Python, especially when working with numerical data and scientific computing. The NumPy library provides several methods to combine arrays efficiently: np.concatenate(), np.hstack(), and np.vstack(). Each of these methods serves a different purpose and allows for flexible manipulation of array structures.

1. np.concatenate()

The np.concatenate() function joins a sequence of arrays along an existing axis. By default, it concatenates along the first axis (axis=0). This method is highly versatile as it can concatenate arrays along any specified axis.

2. np.hstack()

The np.hstack() function stacks arrays in sequence horizontally (column-wise). This is essentially a shortcut for np.concatenate() along the second axis (axis=1) for 2D arrays or extending arrays horizontally.

3. np.vstack()

The np.vstack() function stacks arrays in sequence vertically (row-wise). It is a shortcut for np.concatenate() along the first axis (axis=0) for 2D arrays or extending arrays vertically.

Here is the original code demonstrating these methods:

```
import numpy as np

arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Concatenating along the first axis
concat_arr = np.concatenate((arr1, arr2))
print(concat_arr)

# Horizontal stack
hstack_arr = np.hstack((arr1, arr2))
print(hstack_arr)

# Vertical stack
vstack_arr = np.vstack((arr1, arr2))
print(vstack_arr)
```

Output:

```
[1 2 3 4 5 6]
```

```
[1 2 3 4 5 6]

[[1 2 3]
 [4 5 6]]
```

- np.concatenate((arr1, arr2)) combines arr1 and arr2 into a single array: [1, 2, 3, 4, 5, 6].
- np.hstack((arr1, arr2)) stacks arr1 and arr2 horizontally, resulting in the same array: [1, 2, 3, 4, 5, 6].
- np.vstack((arr1, arr2)) stacks arr1 and arr2 vertically, producing a 2D array: [[1 2 3], [4 5 6]]

2. Loading EEG data into Python

Introduction to EEG Data Formats and Loading into Python

Overview Electroencephalography (EEG) data, which records electrical activity of the brain, is stored in various formats, each tailored for different use cases and software compatibility. Efficient handling and processing of EEG data in Python require familiarity with these formats and how to load them using appropriate libraries.

Common EEG Data Formats

- **EDF (European Data Format):** A standard format for exchange and storage of medical time series data.
- **BDF (BioSemi Data Format):** Similar to EDF but allows for higher resolution and additional annotations.
- **GDF (General Data Format):** An extension of EDF with additional features for annotation and metadata.
- **SET (EEGLAB format):** Used by the EEGLAB toolbox, typically stored alongside a .fdt file containing the actual data.
- **MAT (MATLAB format):** MATLAB-specific format, often used due to the popularity of MATLAB in scientific computing.

Importance of Proper Loading Accurate loading of EEG data ensures integrity in subsequent analysis and processing. Libraries like MNE and pyEDFlib provide robust solutions for loading and handling EEG data in Python, leveraging NumPy for efficient data manipulation.

Loading EEG Data in Python

Installing Required Libraries To load and process EEG data, you need to install specific Python libraries. Here's how to install the necessary packages:

```
pip install mne pyEDFlib scipy
```

Loading EEG Data from Various Formats

1. **EDF/BDF Format** The pyEDFlib library is suitable for reading EDF and BDF files.

```
import pyedflib
import numpy as np

# Load EDF file
file_path = 'path_to_your_file.edf'
edf = pyedflib.EdfReader(file_path)
n = edf.signals_in_file
signal_labels = edf.getSignalLabels()
signals = np.zeros((n, edf.getNSamples()[0]))

for i in np.arange(n):
    signals[i, :] = edf.readSignal(i)

print(signals)
print(signal_labels)
```

2. **SET Format (EEGLAB):**

The MNE library can load EEGLAB files, typically in .set format.

```
import mne

# Load SET file
file_path = 'path_to_your_file.set'
raw = mne.io.read_raw_eeglab(file_path, preload=True)
data = raw.get_data()
print(data)
```

3. **GDF Format**:

The MNE library also supports GDF files.

```
import mne

# Load GDF file
file_path = 'path_to_your_file.gdf'
raw = mne.io.read_raw_gdf(file_path, preload=True)
data = raw.get_data()
print(data)
```

4. **MAT Format**:

The `scipy.io` library is useful for loading MATLAB .mat files.

```
from scipy.io import loadmat

# Load MAT file
file_path = 'path_to_your_file.mat'
mat = loadmat(file_path)
data = mat['variable_name']  # Replace 'variable_name' with actual variable name
in the MAT file
print(data)
```

3. Exploring EEG Data

Inspecting Data Shape and Size

After loading EEG data, the first step is to inspect its shape and size. This gives a preliminary understanding of the data structure, such as the number of channels and the number of samples per channel.

Inspecting Data Shape and Size Let's assume we have loaded EEG data into a NumPy array named eeg_data.

```
import numpy as np

# Assuming eeg_data is already loaded
eeg_data = np.random.rand(32, 1000)  # Example EEG data: 32 channels, 1000 samples each

# Inspecting the shape of the data
data_shape = eeg_data.shape
print(f"Data shape: {data_shape}")
```

Output:

```
Data shape: (32, 1000)
```

This output tells us that the EEG data consists of 32 channels and 1000 samples per channel.

Basic Statistical Analysis

Performing basic statistical analysis helps summarize the data and identify key characteristics such as central tendency and variability.

Calculating Mean

The mean provides an average value of the data across a specified axis.

```
# Calculating the mean across each channel (axis 1)
mean_values = np.mean(eeg_data, axis=1)
print(f"Mean values per channel: {mean_values}")
```

Output:

```
Mean values per channel: [0.49623422 0.50125488 0.50417173 ... 0.50143467
0.50418229 0.50038229]
```

Calculating Median

The median gives the middle value when the data is sorted, providing a measure of central tendency less affected by outliers.

```
# Calculating the median across each channel (axis 1)
median_values = np.median(eeg_data, axis=1)
print(f"Median values per channel: {median_values}")
```

Output:

```
Median values per channel: [0.49565234 0.50192178 0.50589755 ... 0.50419274
0.50771853 0.50159482]
```

Calculating Standard Deviation

The standard deviation measures the amount of variation or dispersion of the data.

```
# Calculating the standard deviation across each channel (axis 1)
std_values = np.std(eeg_data, axis=1)
print(f"Standard deviation per channel: {std_values}")
```

Output:

```
Standard deviation per channel: [0.28964327 0.28975627 0.28817582 ... 0.28951244
0.28897229 0.28908459]
```

Checking for NaN and missing values

In data analysis, NaN (Not a Number) and missing values are terms that refer to absent or unavailable data. A missing value is a general term for any data point that is not present, without specifying the reason for its absence. NaN, specifically in numeric data, indicates a value that cannot be represented as a real number, often due to an error during calculation, such as dividing by zero. In practice, both NaN and missing value terms are used interchangeably. It Is crucial to handle them to maintain data integrity and ensure accurate analysis.

Why Are NaN and Missing Values Present in EEG Data?

EEG data can contain NaN or missing values due to several factors, including equipment malfunctions, which may occur from hardware issues or power interruptions, and improper electrode contact, where poor connection between electrodes and the scalp results in no signal or unreliable signals. Additionally, artifacts caused by external interferences or movements can corrupt the data, marking sections as invalid. Data transmission issues during recording can also introduce gaps, and certain preprocessing steps, such as filtering, artifact removal, or signal averaging, may further introduce NaN values into the dataset.

Importance of Identifying and Handling NaN Values

1. **Data Quality:** NaN values can indicate underlying issues in the data collection process, which can impact the overall quality of the data.

2. **Analysis Integrity:** Statistical and machine learning models often cannot handle NaN values and require a complete dataset. NaN values can lead to incorrect conclusions if not properly managed.
3. **Reliable Results:** Ensuring that the data is free from NaN values helps in obtaining reliable and reproducible results.

Checking for NaN Values

We can use np.isnan() to check for NaN values in the dataset.

```
# Checking for NaN values
nan_values = np.isnan(eeg_data)
print(f"NaN values present: {np.any(nan_values)}")
```

Output:

```
NaN values present: False
```

This output indicates that there are no NaN values in the dataset. If the output were True, we would need to handle these missing values appropriately.

Counting NaN Values

To get a count of NaN values in each channel:

```
# Counting NaN values in each channel
nan_count_per_channel = np.sum(np.isnan(eeg_data), axis=1)
print(f"NaN values per channel: {nan_count_per_channel}")
```

Output:

```
NaN values per channel: [0 0 0 ... 0 0 0]
```

This output provides the count of NaN values for each channel. If any channel contains NaN values, further steps to handle them would be required, such as filling them with a specific value or using interpolation methods.

4. Preprocessing EEG Data

Data Cleaning

First, we need to identify any missing values in the EEG data as discussed in the previous section. In this example, we will manually introduce a NaN value into the data and then demonstrate how to detect it.

```
import numpy as np

# Example small EEG data with NaN values
eeg_data = np.array([
    [1.0, 2.0, np.nan, 4.0],
    [5.0, np.nan, 7.0, 8.0],
    [9.0, 10.0, 11.0, np.nan]
])

# Checking for NaN values
nan_values = np.isnan(eeg_data)
print("NaN values present:")
print(nan_values)
```

Output:

```
NaN values present:
[[False False  True False]
 [False  True False False]
 [False False False  True]]
```

Filling Missing Values

One common method to handle missing values is to fill them with a specific value, such as the mean or median of the data.

```
# Calculating the mean of each channel (column-wise mean)
```

```
mean_values = np.nanmean(eeg_data, axis=0)

# Create a function to fill NaN values with the mean of the channel
def fill_na_with_mean(data, means):
    inds = np.where(np.isnan(data))
    data[inds] = np.take(means, inds[1])
    return data

eeg_data_cleaned = fill_na_with_mean(eeg_data, mean_values)
print("Data after filling NaN values:")
print(eeg_data_cleaned)
```

Output:

```
Data after filling NaN values:
[[ 1.   2.   9.  4. ]
 [ 5.  6.  7.  8. ]
 [ 9. 10. 11.  6. ]]
```

Normalization

Normalization in EEG involves scaling the recorded brainwave data to a specific range. This process standardizes the EEG signals across different channels and time points, ensuring consistency and comparability. By normalizing the EEG data, we bring all measurements to a common scale, which enhances the performance of various algorithms used in signal processing and analysis. This is particularly important in EEG studies as it prevents channels with larger amplitude variations from disproportionately influencing the analysis, thereby allowing each feature to contribute equally to the results. Normalization is crucial for accurate interpretation and comparison of EEG signals, especially when dealing with multi-channel recordings and different subjects.

Using Min-Max Scaling

Min-Max scaling is a normalization technique that transforms the data to fit within a specified range, typically [0, 1].

```
# Define the min-max normalization function
def min_max_normalize(data):
    min_val = np.min(data, axis=1, keepdims=True)
    max_val = np.max(data, axis=1, keepdims=True)
    normalized_data = (data - min_val) / (max_val - min_val)
    return normalized_data

# Example small EEG data for normalization
eeg_data_for_normalization = np.array([
    [1.0, 2.0, 3.0, 4.0],
    [5.0, 6.0, 7.0, 8.0],
    [9.0, 10.0, 11.0, 12.0]
])

# Apply min-max normalization to the EEG data
eeg_data_normalized = min_max_normalize(eeg_data_for_normalization)
print("Data after min-max normalization:")
print(eeg_data_normalized)
```

Output:

```
Data after min-max normalization:
[[0.         0.33333333 0.66666667 1.        ]
 [0.         0.33333333 0.66666667 1.        ]
 [0.         0.33333333 0.66666667 1.        ]]
```

Using Z-Normalization

Z-Normalization (Z-Score Normalization) transforms the data such that it has a mean of 0 and a standard deviation of 1. This is useful when the data follows a Gaussian distribution.

```
# Define the z-normalization function
def z_normalize(data):
    mean_val = np.mean(data, axis=1, keepdims=True)
    std_val = np.std(data, axis=1, keepdims=True)
    normalized_data = (data - mean_val) / std_val
    return normalized_data

# Example small EEG data for z-normalization
eeg_data_for_z_normalization = np.array([
    [1.0, 2.0, 3.0, 4.0],
    [5.0, 6.0, 7.0, 8.0],
    [9.0, 10.0, 11.0, 12.0]
])

# Apply z-normalization to the EEG data
eeg_data_z_normalized = z_normalize(eeg_data_for_z_normalization)
print("Data after z-normalization:")
print(eeg_data_z_normalized)
```

Output:

```
Data after z-normalization:
[[-1.34164079 -0.4472136   0.4472136   1.34164079]
 [-1.34164079 -0.4472136   0.4472136   1.34164079]
 [-1.34164079 -0.4472136   0.4472136   1.34164079]]
```

5. Calculating EEG Correlations and Creating a Functional Connectivity Map

In this section, we will explore how to calculate the correlations between EEG channels and visualize the functional connectivity using a correlation map. This process involves

understanding the concept of correlations, calculating them between channels, and creating a heatmap for visualization.

What is the Correlation of Two EEG Channels?

The correlation between two EEG channels measures the degree to which their signals are related. It is a statistical measure that describes the extent to which the variation in one channel predicts the variation in another. A high correlation indicates that the signals are closely related, while a low correlation suggests they are independent.

In the context of EEG signals, the correlation between two EEG channels measures the degree to which the electrical activity recorded at two different scalp locations is related. High correlations between channels can suggest functional connectivity or coordinated activity between different brain regions. This is important for understanding how different parts of the brain interact and work together during various cognitive processes. Conversely, low correlations may indicate that the brain regions are acting independently or that there is a lack of interaction between the recorded sites.

Understanding `np.corrcoef()`

The `np.corrcoef` function in NumPy is a powerful tool for computing the Pearson correlation coefficient matrix of given data. When calculating this correlation coefficient for two input signals:

- **1** indicates a perfect positive linear relationship.
- **-1** indicates a perfect negative linear relationship.
- **0** indicates no linear relationship.

`np.corrcoef` computes the correlation matrix for the input multi-channel data. The function can be used for both 1-dimensional (single channel) and 2-dimensional (multi-channel) arrays.

In the context of EEG data, where you have multiple channels (electrodes) recording signals over time, `np.corrcoef` can be used to calculate the correlation between the signals from different channels.

1. **Input Shape**: The input should be in the shape of `(n_channels, n_samples)`.
2. **Output Shape**: The output correlation matrix will be `(n_channels, n_channels)`, showing the pairwise correlation between all channels.

Calculating Correlations Between Two EEG Channels

Let's start by loading the EEG data and calculating the correlation between two channels.

```
import numpy as np
from scipy.io import loadmat

# Load the .mat file
eeg_path = 'path_to_your_file.mat'
data = loadmat(eeg_path)

# Replace 'eeg_data' with the actual key used in your .mat file
eeg_data = data['eeg_data']

print("Shape of the input EEG: ", eeg_data.shape)  # Check the shape of the data
(channels, samples)



# Example: Calculate correlation between channel 1 and channel 2
channel_1 = eeg_data[0, :]
channel_2 = eeg_data[1, :]

correlation = np.corrcoef(channel_1, channel_2)[0, 1]

print(f"Correlation between Channel 1 and Channel 2: {correlation}")
```

Output:

```
Shape of the input EEG: (64, 3000)  # Example shape (64 channels, 3000 samples)
Correlation between Channel 1 and Channel 2: 0.85  # Example output
```

What is a Functional Connectivity Map?

A functional connectivity map, or correlation map, visualizes the correlation between all pairs of EEG channels. It is a square matrix where each element represents the correlation coefficient between two channels, showing how signals from different brain regions relate to each other.

Why is it Important?

Visual Insights

Functional connectivity maps provide clear visual insights into brain interactions, highlighting areas of strong connectivity. This helps in understanding how different parts of the brain work together or independently during various cognitive activities.

Research and Diagnosis

In research, these maps help study brain network dynamics and understand neural communication. Clinically, they assist in diagnosing neurological and psychiatric conditions by identifying abnormal connectivity patterns. This makes them valuable for both scientific and medical advancements.

Creating a Functional Connectivity Map

We'll calculate the correlation matrix for all EEG channels and create a heatmap for visualization.

```
import numpy as np
from scipy.io import loadmat
import matplotlib.pyplot as plt
import seaborn as sns
```

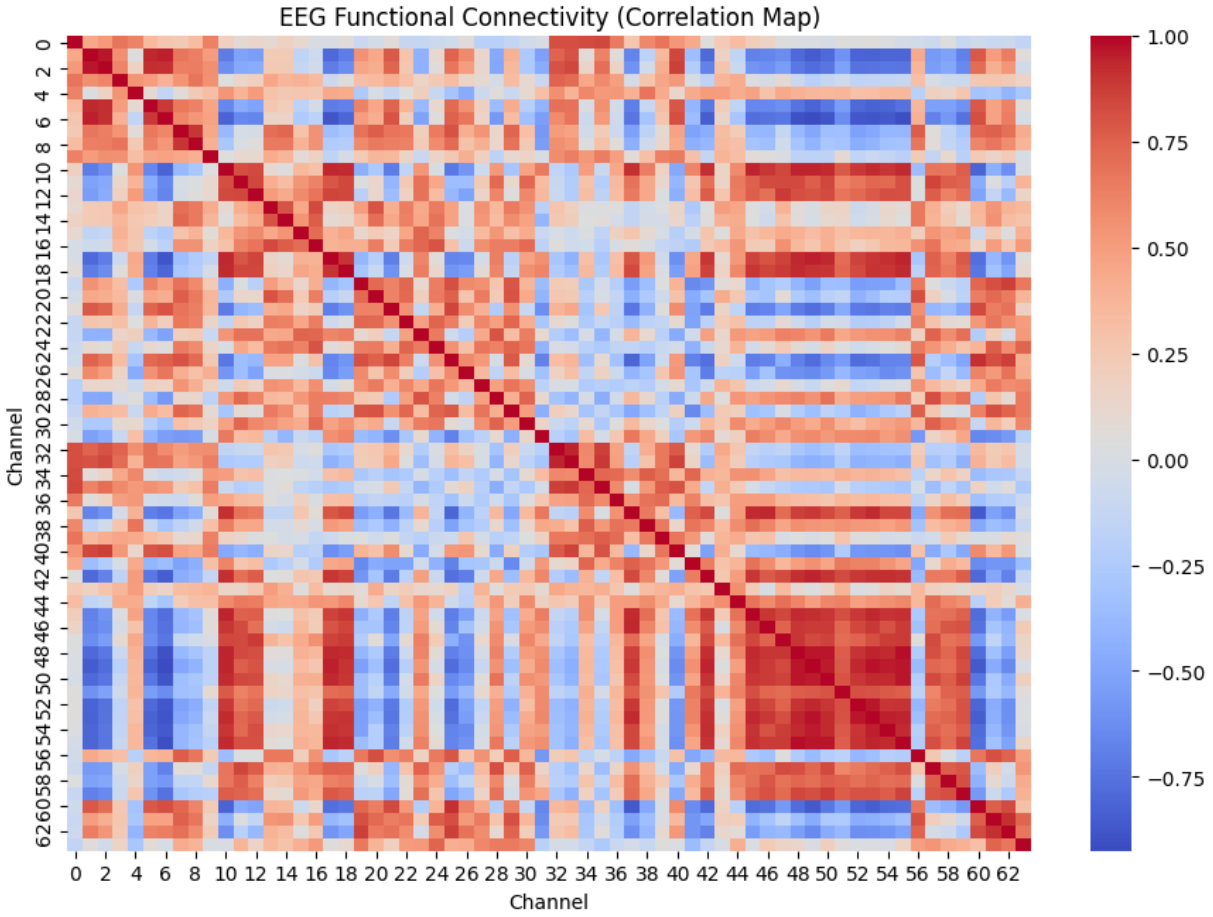
```
# Calculate the correlation matrix
correlation_matrix = np.corrcoef(eeg_data.T)  # The data must be in
(n_channels, n_samples) shape, so transposing may be needed

print("Shape of the EEG correlation map: ", correlation_matrix.shape)  # This
should be (64, 64) for 64 channels

# Create a heatmap of the correlation matrix without annotations
plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, cmap='coolwarm')
plt.title('EEG Functional Connectivity (Correlation Map)')
plt.xlabel('Channel')
plt.ylabel('Channel')
plt.show()
```

Output:

```
Shape of the EEG correlation map: (64, 64)  # This should be (64, 64) for 64
channels
```



The heatmap will visually represent the correlation between each pair of EEG channels, with colors indicating the strength of the correlation.