Manual Técnico Software
GMRRNet

Danna Valentina Salazar Dubois

Universidad Nacional de Colombia
Sede Manizales

Índice

1. Descripción general del sistema de información desarrollado

El sistema de información GMRRNet, desarrollado para la clasificación de Imágenes Motoras (MI) basadas en señales de electroencefalografía (EEG), consiste en una red de conectividad funcional que utiliza una mezcla gaussiana con regularización α-Renyi. Este sistema está diseñado específicamente para mejorar la precisión en la clasificación de datos de EEG, utilizados en aplicaciones de interfaces cerebro-computadora (BCI). El objetivo del sistema es capturar de manera eficiente las relaciones neuronales tanto locales como globales a través de un conjunto de módulos que permiten extraer características espaciales y temporales de las señales EEG. Los principales componentes del sistema incluyen: i) la extracción de características usando capas convolucionales que capturan patrones en los datos EEG; ii) la aplicación de núcleos gaussianos de diferentes anchos de banda que modelan la conectividad neuronal; iii) el uso de regularización α-Renyi para mejorar la estabilidad y precisión del modelo; iv) la clasificación de las señales de EEG en base a los patrones identificados, mejorando la precisión incluso en sujetos con variabilidad alta o habilidades motoras más bajas. El sistema también permite una mejor interpretabilidad de las señales EEG, mostrando diferencias significativas entre ensayos de imágenes motoras, especialmente en las áreas cerebrales responsables del control motor (regiones C3 y C4). Su modularidad lo hace adaptable para futuras aplicaciones como la neurorehabilitación, el control de prótesis, y el monitoreo de salud mental, a través del uso de técnicas avanzadas de aprendizaje automático y análisis de patrones.

2. Requerimientos técnicos a nivel de hardware y software para instalar y operar el software desarrollado

El software GMRRNet requiere Python$\geq$ 3.8 y acceso a internet para descargar las librerías requeridas.

2.1. Instalar desde el código fuente

Para tener acceso al modelo por favor ingrese el siguiente comando en su código:

```
!pip install -U git+https://github.com/dannasalazar11/GMRRNet
```

2.2. Añadir la librería en su código

```
import gmrrnet.model as gm
```

3. Entorno y/o Lenguaje de desarrollo utilizado

El modelo GMRRNet está desarrollado utilizando Python 3.10.13. Además, el núcleo principal de este software se encuentra en la librería *TensorFlow*, diseñada para manejar tareas de aprendizaje automático y redes neuronales profundas de manera eficiente.

4. Descripción de los diferentes módulos que hacen parte del software GMRRNet

A continuación se hace la descripción de los módulos usados en el modelo GMRRNet, que se encargan de realizar las funciones definidas en la red de la **??**. Dado que el modelo está en inglés, dicha descripción de realizará en inglés:

```
class gmrrnet.model.GMRRNet(nb\_classes=2, Chans=64, Samples=320, kernLength
    =64, norm\_rate=0.25, alpha=2)
```

Builds a convolutional neural network model based on Inception blocks with convolutional layers and $\alpha-$Rényi entropy regularizer.

	Argument	Description
Parameters	{nb_classes : int, optional (default=2)}	Number of output classes for classification.
	{Chans : int, optional (default=64)}	Number of input channels (spatial dimension).
	{Samples : int, optional (default=320)}	Number of input samples (temporal dimension).
	{kernLength : int, optional (default=64)}	Kernel length for the first convolutional layer.
	{norm_rate : float, optional (default=0.25)}	Normalization rate for regularization in dense layers.
	{alpha : int, optional (default=2)}	Order parameter for Rényi entropy, where alpha=2 represents quadratic Rényi entropy.
Returns	{model : tf.keras.Model}	The compiled neural network model ready for training.

```
class gmrrnet.model.GaussianKernelLayer(*args: Any, **kwargs: Any)
```

Custom Keras layer that applies a Gaussian kernel to the inputs. This layer calculates the squared Euclidean distance between pairs of points and then applies a Gaussian kernel function to transform those distances into similarities, which is useful in signal processing such as EEG or in constructing neural networks with kernel functions.

	Argument	Description
Parameters	{sigma: float, optional (default=1.0) }	Standard deviation of the Gaussian kernel function. Controls the range or "spread" of the Gaussian.

Methods:

- build(input_shape)

  Parameter: input_shape : tuple. The expected shape of the input to the layer.

- call(inputs)

  Parameters:

  - inputs : Tensor.

    Input tensor with shape (N, C, T, F) where: - N: Number of samples in the batch. - C: Number of channels or features. - T: Number of time steps. - F: Number of filters.

  Returns:

  - gaussian_kernel : Tensor.

    Output tensor where the Gaussian kernel has been applied, with shape (N, C, C, F).

```
class gmrrnet.model.NormalizedBinaryCrossentropy(*args: Any, **kwargs: Any)
```

This class implements a normalized version of the binary cross-entropy loss. Binary cross-entropy is a measure of dissimilarity between two probability distributions, commonly used as a loss function in binary classification problems.
In this implementation, the binary cross-entropy loss is normalized using the theoretical losses associated with the labels [1, 0] and [0, 1], which represent the two possible classes. The normalization aims to adjust the loss to be more robust against skewed probability distributions.
Methods:

- call(inputs)

  Parameters:

  - kwargs : dict.

    Optional arguments passed to the base Loss class.

Returns:

- gaussian_kernel : Tensor. Output tensor where the Gaussian kernel has been applied, with shape (N, C, C, F).

- call(y_true, y_pred)

  Parameters:

  - y_true : Tensor.
    Tensor of true labels with shape (N, 2), where: - N: Number of samples in the batch. - 2: Corresponds to binary classes (0 or 1).
  - y_pred : Tensor
    Tensor of predictions with shape (N, 2), where: - N: Number of samples in the batch. - 2: Probability predictions for the two binary classes.

  Returns:

  - Tensor.
    A loss tensor with shape (N,), containing the normalized binary cross-entropy loss for each sample.

```
class gmrrnet.model.RenyiMutualInformation(*args: Any, **kwargs: Any)
```

This class implements a loss based on Rényi mutual information, which is a measure of the amount of information shared between two or more variables. In this case, it is used to evaluate how well the model predictions reflect the dependency between different features.
Methods:

- ___init___(C, **kwargs)

  Parameters:

  - C : int or float
    The number of channels (dimension C) used for normalization in the mutual information calculation.
  - kwargs : dict
    Other optional arguments passed to the base Loss class.

- call(y_true, y_pred)

  Parameters:

  - y_true : Tensor
    True labels, not used in this loss calculation but required to comply with the Keras API.
  - y_pred : Tensor
    Prediction tensor of shape (N, F+1), where: - N: Number of samples in the batch. - F: Number of marginal entropies. - F+1: The last column contains the joint entropy.

  Returns:

  - Tensor
    A loss tensor of shape (N, 1), which contains the calculated Rényi mutual information for each sample.

Notes:
This class is designed to work in conjunction with a model that outputs both the marginal entropies and the joint entropy. Rényi mutual information is useful in tasks where it is important to evaluate the amount of shared information between different features or signals.

```
gmrrnet.model.inception_block(x, filters, sigmas)
```

Builds a custom Inception block that includes convolution layers and Gaussian kernel layers.
This Inception block creates three branches, each applying a Gaussian kernel followed by a 2D convolution layer. Finally, the outputs of these branches are concatenated along the channel axis.

	Argument	Description
Parameters	{x : Tensor}	Input tensor of shape (N, C, T, F) where: - N: Number of samples in the batch. - C: Number of channels or features. - T: Number of time steps. - F: Number of additional filters or features.
	{filters : list of int}	List containing the number of filters for each branch of the Inception block. It should be a list of three integers [f1, f2, f3] where f1, f2, and f3 are the number of filters for branches 1, 2, and 3, respectively.
	{sigmas: list of float}	List containing the sigma values for each GaussianKernelLayer in the branches of the Inception block. It should be a list of three values [sigma1, sigma2, sigma3] where sigma1, sigma2, and sigma3 correspond to branches 1, 2, and 3, respectively.
Returns	{branch_k1, branch_k2, branch_k3 : Tensors}	The outputs of the GaussianKernelLayer in the three branches, with shape (N, C, C, F).
	{output : Tensor}	The concatenated output of the three branches after the convolution layer, with shape (N, C, T, f1 + f2 + f3).

```
gmrrnet.model.joint_renyi_entropy(K, alpha)
```

This function calculates the joint $\alpha-$Rényi entropy, which measures the amount of uncertainty in a system by considering multiple variables together. Joint entropy is useful for evaluating the dependency or interrelation between variables.

	Argument	Description
Parameters	{K : Tensor}	An input tensor of shape (N, F, C, C), where: - N: Number of samples in the batch. - F: Number of filters or features. - C: Number of channels or dimensions of the square matrices within the tensor.
	{alpha : float}	The $\alpha-$Rényi entropy parameter. Controls the sensitivity of the metric to different probability distributions.
Returns	{Tensor}	An output tensor of shape (N, 1), which contains the calculated joint Rényi entropy for each sample.

```
gmrrnet.model.renyi_entropy(K, alpha=2)
```

$\alpha-$Rényi entropy is a generalization of Shannon entropy, which depends on the parameter alpha. This function calculates the $\alpha-$Rényi entropy for each of the normalized square matrices in the tensor K.

	Argument	Description
Parameters	{K : Tensor}	An input tensor of shape (N, F, C, C), where: - N: Number of samples in the batch. - F: Number of filters or features. - C: Number of channels or dimensions of the square matrices within the tensor.
	{alpha : float, optional}	The $\alpha-$Rényi entropy parameter. Default is 2.0. - When alpha=2, a specific optimization is applied for this value.
Returns	{Tensor}	An output tensor of shape (N, F), which contains the calculated Rényi entropy for each sample and filter.

5. Diagrama de clase del software GMRRNet

El diagrama de entidad-relación representa la estructura y funcionalidades de la clase GMRRNet , diseñada para la clasificación de señales de EEG permitiendo el conservar la interpretabilidad. A continuación se detalla su diagrama:

```
+-----------------------+
|        GMRRNet        |
+-----------------------+
| - input_shape: tuple  |
| - num_classes: int    |
| - dropout_rate: float |
| - base_filters: int   |
| - num_blocks: int     |
| - optimizer: str      |
| - loss_function: str  |
+-----------------------+
| + build_model(): Model |
| + residual_block(x: Tensor, filters: int, kernel_size: int) |
| + multires_block(x: Tensor, filters: int) |
| + global_max_pooling(x: Tensor): Tensor |
| + compile_model(): None |
| + fit_model(X_train: np.ndarray, y_train: np.ndarray, epochs: int, batch_size: int): History |
+-----------------------+
```

6. Caso de uso

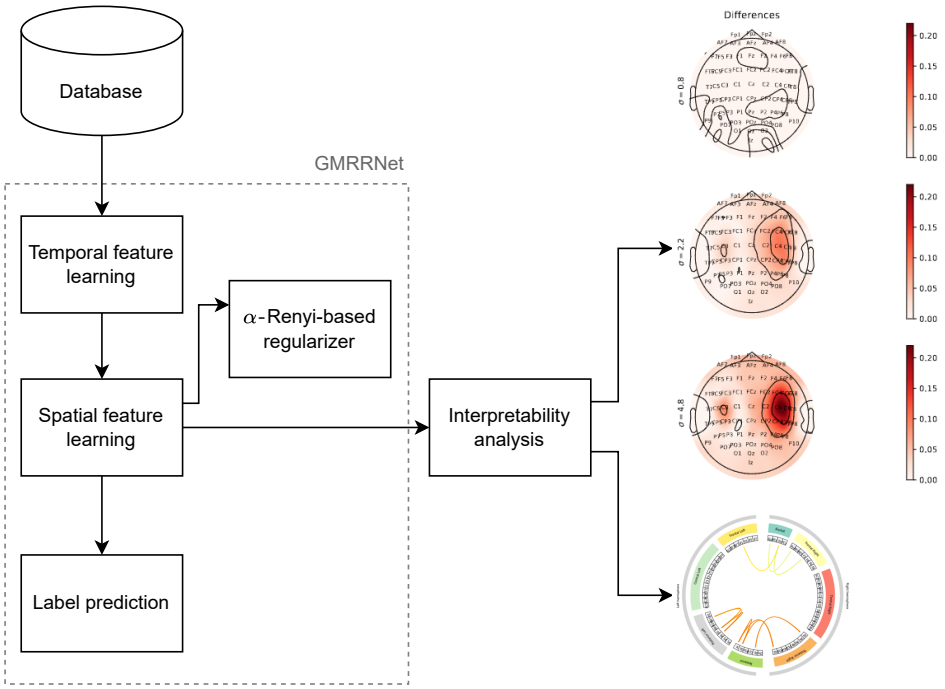


Figura 1: Diagrama de bloques del funcionamiento

Como se puede evidenciar en el diagrama de bloques de la 1, se importa la base de datos, se entrena la red (GMRRNet), se realiza la clasificación y se puede hacer un análisis de interpretabilidad.

6.1. Instalando la librería

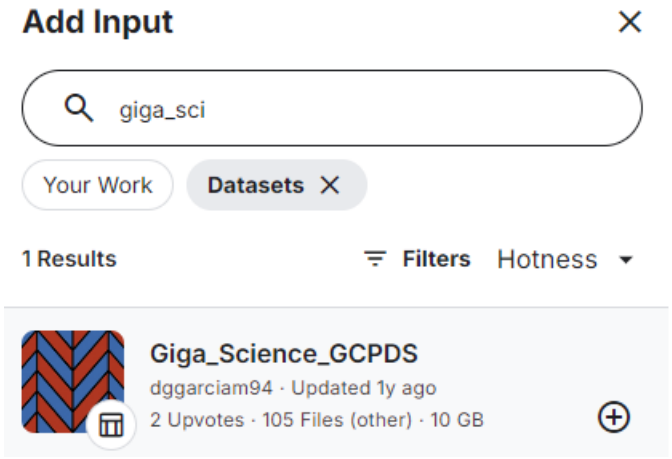
Para poder tener acceso al software basta con tener la siguiente linea dentro de tu cuaderno de *Kaggle*

[2]:
```
!pip install -U git+https://github.com/dannasalazar11/GMRRNet
```

Una vez instalada, se podrá llamar la clase con la siguiente linea

[2]:
```
from gmrrnet.model import GMRRNet
```

Además, para el ejemplo propuesto usaremos la base de datos "GIGA_MI_ME". Para ello, debemos agregarla como input



Y para poderla cargar y ver los resultados de la interpretabilidad, instalaremos las dependencias necesarias y usaremos una de las herramientas proporcionadas por la librería

[2]:
```
!pip install -U git+https://github.com/UN-GCPDS/python-gcpds.databases
!pip install gmrrnet[dev]
```

```
ae6714916
  Preparing metadata (setup.py) ... done
Requirement already satisfied: numpy in /opt/conda/lib/python3.10/site-packages (from gcpds-databases==
0.2) (1.26.4)
Requirement already satisfied: scipy in /opt/conda/lib/python3.10/site-packages (from gcpds-databases==
0.2) (1.11.4)
Requirement already satisfied: matplotlib in /opt/conda/lib/python3.10/site-packages (from gcpds-databa
ses==0.2) (3.7.5)
Requirement already satisfied: mne in /opt/conda/lib/python3.10/site-packages (from gcpds-databases==0.
2) (1.7.1)
Requirement already satisfied: tables in /opt/conda/lib/python3.10/site-packages (from gcpds-databases=
```

[4]:
```
from gmrrnet.utils import load_GIGA
from gmrrnet.utils import topoplot
```

6.2. Agregando todo lo necesario para nuestro código

Importamos las librerías necesarias como *numpy* y *sklearn*, y la herramienta *GIGA_MI_ME* para el correcto funcionamiento del código

[5]:
```
import numpy as np
import tensorflow as tf

from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from gcpds.databases import GIGA_MI_ME
from gcpds.visualizations.connectivities import CircosConnectivity
import matplotlib.pyplot as plt
```

6.3. Importación de los datos

Dado que esta red está diseñada para la clasificación de señales de EEG, vamos a usar la base de datos *GIGA_MI_ME*, en específico los datos de un sujeto que realiza dos actividades de imaginación motora (pensar en mover la mano izquierda y pensar en mover la mano derecha). Para cargar los datos de este sujeto, hacemos el siguiente procedimiento

[7]:
```
db = GIGA_MI_ME('/kaggle/input/giga-science-gcpds/GIGA_MI_ME')
fs = db.metadata['sampling_rate']
# 64 canales
eeg_ch_names = ['Fp1','Fpz','Fp2',
                'AF7','AF3','AFz','AF4','AF8',
                'F7','F5','F3','F1','Fz','F2','F4','F6','F8',
                'FT7','FC5','FC3','FC1','FCz','FC2','FC4','FC6','FT8',
                'T7','C5','C3','C1','Cz','C2','C4','C6','T8',
                'TP7','CP5','CP3','CP1','CPz','CP2','CP4','CP6','TP8',
                'P9','P7','P5','P3','P1','Pz','P2','P4','P6','P8','P10',
                'PO7','PO3','POz','PO4','PO8',
                'O1','Oz','O2',
                'Iz']
load_args = dict(db = db,
                 eeg_ch_names = eeg_ch_names,
                 fs = fs,
                 f_bank = np.asarray([[4., 40.]]), # bandpass
                 vwt = np.asarray([[2.5, 5]]),
                 new_fs = 128.)

sbj=43
X, y = load_GIGA(sbj=sbj, **load_args)
encoder = OneHotEncoder(sparse_output=True)
y = encoder.fit_transform(y.reshape(-1,1)).toarray()

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
Resampling from 512.000000 to 128.000000 Hz.
```

6.4. Entrenamiento del modelo

Ahora, entrenamos el modelo con el sujeto escogido

[17]:
```
model = GMRRNet()

history = model.fit(X_train,y_train, epochs=150, batch_size=32, verbose=1,
                    validation_data=(X_test, y_test))
```

6.5. Resultados del modelo e intratabilidad

Con los datos de test, podemos verificar cómo es el rendimiento de la red (en este caso se acertó en el 100 % de los datos de test)

[36]:
```
y_pred = model.predict(X_test)[0]
```

```
2/2 [==============================] - 0s 9ms/step
```

[40]:
```
accuracy_score(np.argmax(y_test, axis=1), np.argmax(y_pred, axis=1))
```

[40]:
```
1.0
```

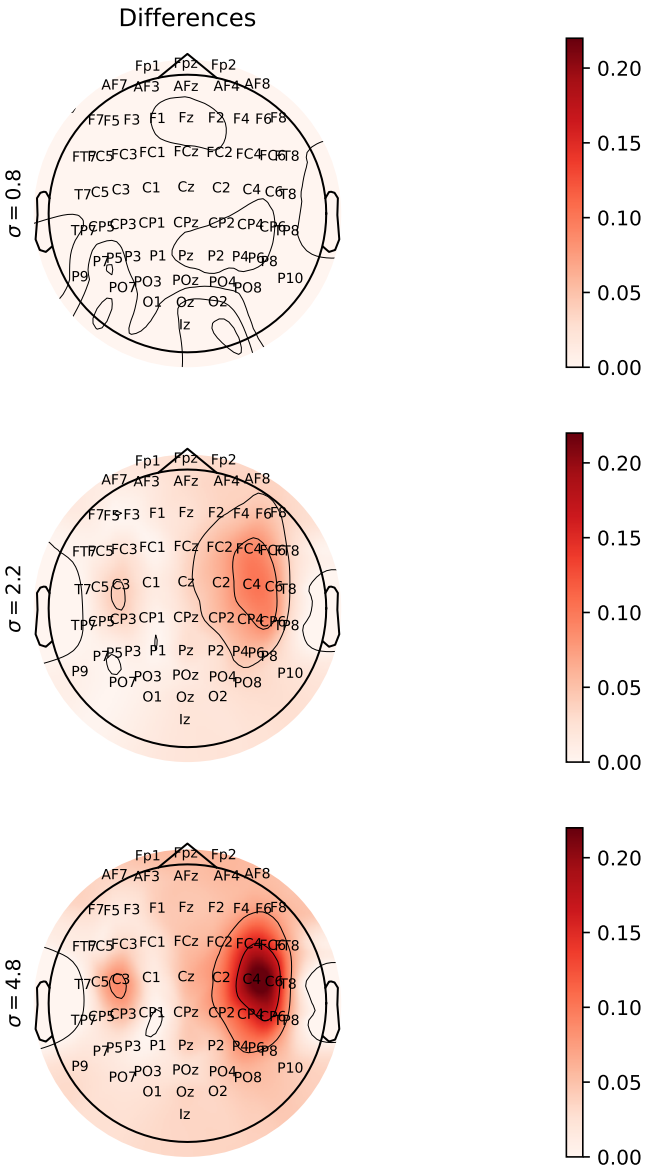
Para saber qué es lo que está teniendo en cuenta el modelo para realizar la clasificación, veremos un gráfico de topoplots. Para ello necesitamos la información de los kernels lo cual se realiza a continuación

[44]:
```
kernel1 = tf.keras.Model(inputs=model.inputs,outputs=model.get_layer('gaussian_layer_1').output)
kernel2 = tf.keras.Model(inputs=model.inputs,outputs=model.get_layer('gaussian_layer_2').output)
kernel3 = tf.keras.Model(inputs=model.inputs,outputs=model.get_layer('gaussian_layer_3').output)

idx_left = tf.squeeze(tf.where(np.argmax(y_train, axis=1)==0))
idx_right = tf.squeeze(tf.where(np.argmax(y_train, axis=1)==1))

### kernel 1
X_k1 = kernel1.predict(tf.expand_dims(X_train[0], axis=0))
X_k1 = tf.reduce_mean(X_k1, axis=-1) # promedio por filtros

X_k1_left = tf.reduce_mean(tf.gather(X_k1, idx_left), axis=0) # promedio de clase izq
X_k1_right = tf.reduce_mean(tf.gather(X_k1, idx_right), axis=0) # promedio de clase der

### Kernel 2
X_k2 = kernel2.predict(X_train)
X_k2 = tf.reduce_mean(X_k2, axis=-1) # promedio por filtros

X_k2_left = tf.reduce_mean(tf.gather(X_k2, idx_left), axis=0) # promedio de clase izq
X_k2_right = tf.reduce_mean(tf.gather(X_k2, idx_right), axis=0) # promedio de clase der

### Kernel 3
X_k3 = kernel3.predict(X_train)
X_k3 = tf.reduce_mean(X_k3, axis=-1) # promedio por filtros

X_k3_left = tf.reduce_mean(tf.gather(X_k3, idx_left), axis=0) # promedio de clase izq
X_k3_right = tf.reduce_mean(tf.gather(X_k3, idx_right), axis=0) # promedio de clase der
```

Y ya para poder ver el comportamiento de cada kernel vemos la imagen de los topoplots

[56]:

```
fig, axs = plt.subplots(3,1,figsize=[40,10])

axs[0].set_title("Differences")

axs[0].set_ylabel("$\sigma=0.8$")
axs[1].set_ylabel("$\sigma=2.2$")
axs[2].set_ylabel("$\sigma=4.8$")

# diferencias
diferencia1 = tf.abs(tf.subtract(tf.reduce_mean(X_k1_left,axis=0).numpy() , tf.reduce_mean(X_k1_right,axis=0).numpy()))
diferencia2 = tf.abs(tf.subtract(tf.reduce_mean(X_k2_left,axis=0).numpy() , tf.reduce_mean(X_k2_right,axis=0).numpy()))
diferencia3 = tf.abs(tf.subtract(tf.reduce_mean(X_k3_left,axis=0).numpy() , tf.reduce_mean(X_k3_right,axis=0).numpy()))
max_dif = tf.reduce_max(tf.stack([diferencia1, diferencia2, diferencia3]))

vmax = tf.reduce_max(tf.stack([tf.reduce_mean(X_k1_left,axis=0),tf.reduce_mean(X_k1_right,axis=0),
                               tf.reduce_mean(X_k2_left,axis=0),tf.reduce_mean(X_k2_right,axis=0),
                               tf.reduce_mean(X_k3_left,axis=0),tf.reduce_mean(X_k3_right,axis=0)], axis=0))

topoplot(diferencia1, eeg_ch_names, contours=3, names=eeg_ch_names, sensors=False, ax= axs[0], vlim=(0,max_dif))
topoplot(diferencia2, eeg_ch_names, contours=3, names=eeg_ch_names, sensors=False, ax=axs[1], vlim=(0,max_dif))
topoplot(diferencia3, eeg_ch_names, contours=3, names=eeg_ch_names, sensors=False, ax=axs[2], vlim=(0,max_dif))

plt.show()
```

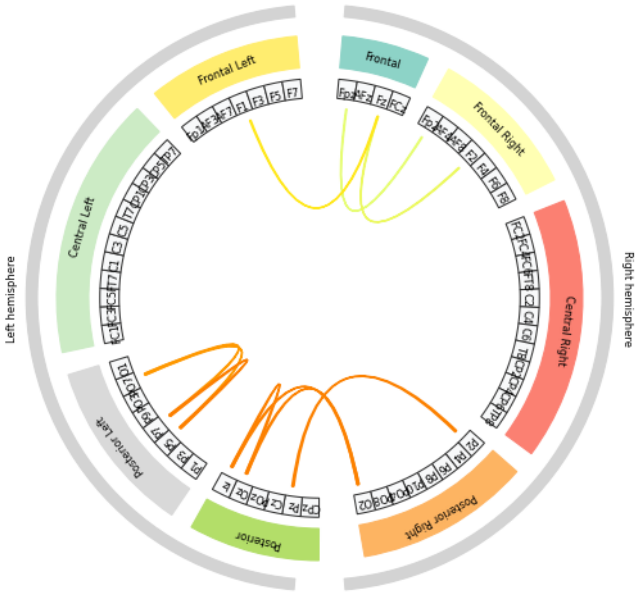


Además, podemos ver el comportamiento de la conectividad funcional. Para ello necesitamos las áreas del cerebro

[86]:

```
areas = {
    'Frontal': ['Fpz', 'AFz', 'Fz', 'FCz'],
    'Frontal Right': ['Fp2','AF4','AF8','F2','F4','F6','F8',],
    'Central Right': ['FC2','FC4','FC6','FT8','C2','C4','C6','T8','CP2','CP4','CP6','TP8',],
    'Posterior Right': ['P2','P4','P6','P8','P10','PO4','PO8','O2',],
    #'Central': ['Cz'],
    'Posterior': ['CPz','Pz', 'Cz','POz','Oz','Iz',],
    'Posterior Left': ['P1','P3','P5','P7','P9','PO3','PO7','O1',],
    'Central Left': ['FC1','FC3','FC5','FT7','C1','C3','C5','T7','CP1','CP3','CP5','TP7',],
    'Frontal Left': ['Fp1','AF3','AF7','F1','F3','F5','F7',],
}
```

Así, especificando el kernel que queremos ver (k2) y la clase que se está clasificando (izquierda : left), se pueden ver las siguientes conectividades


Frontal
Frontal Left
Frontal Right
Central Left
Central Right
Left hemisphere
Right hemisphere
Posterior Left
Posterior
Posterior Right