README file

Unravelling Brain Networks in Chronic Pain and Spinal Cord Stimulation through Magnetoencephalography and Graph Neural Networks

Caroline Witstok

Student number: 4883586

Contact: cfwitstok@gmail.com

15 August 2025

Master Thesis Feasibility Study (TM30002; 15 ECTS)

Department of Pain Medicine, Erasmus Medical Center

June 2025 - August 2025

Supervisors:

Dr. ir. Cecile de Vos

Drs. Laurien Reinders

Contents

Nomenclature

Notation	Description
AUC	Area Under the Curve
GNN	Graph Neural Network
MCTS	Monte Carlo Tree Search
MEG	Magnetoencephalography
MNI	Montreal Neurological Institute
ROC	Receiver Operating Characteristic
SCS	Spinal Cord Stimulation

Project Overview

In this project, a Graph Neural Network (GNN) model has been developed with the aim of classifying stimulation status (ON/OFF) in chronic pain patients with Spinal Cord Stimulation (SCS) using Magnetoencephalography (MEG) data. This project has been developed using Matlab's toolbox Brainstorm and Python. Furthermore, a cluster computer from TU Delft has been used for computational resources.

In this README file, the different scripts are described and a brief overview of the different steps is provided. The scripts can be found on [github repository].

1) **Processing Data:** Preprocess MEG data in Brainstorm and export to Python.
2) **Graph Datasets:** Creating graph datasets with different input settings.
3) **Graph Neural Network:** Train and test GNN models using different graph datasets.
4) **Explainability:** Implement explainability module on different GNN models.
5) **Dependencies:** The necessary software, requirements for running, and usage of the cluster computer.
6) **Plotting:** Creating plots to visualize parts of the analysis framework.

For a more elaborate overview of the methods of this project, as well as the rationale, refer to my Master Thesis report [1]. Furthermore, all Matlab and Python scripts include doc strings and comments to explain the code.

1 Repository

The repository for this project can be found on GitHub https://github.com/carolinefwistok/GNN_MEG. An overview of the folders and scripts contained in the repository is included below.

```
Project repository/
├── Python scripts
│  − generate_graphs.py
│  − dataset.py
│  − dataset_wrapper.py
│  − data_utils.py
│  − train_from_processed.py
│  − train.py
│  − model.py
│  − utils.py
│  − test_saved_model.py
│  − model_explainability.py
│  − explain.py
│  − subgraph_x.py
│  − mcts.py
│  − shapley.py
│  − task_enum.py
│  − export_ftp_python.py
│  − export_scouts.py
│  − plot_psd_analysis.py
│  − roc_plots.py
│
├── Matlab scripts
│  − export_bst_ftp.m
│  − process_raw_string.m
│  − out_fieldtrip_data.m
│
├── Data/
│       ├── raw
│       │    └── scout_files
│       ├── processed
│       ├── Raw_all
│       ├── Scout_all
│       └── Connectivity
│
└── Output/
```

2 Processing Data

This section describes the processing steps in Brainstorm and Python, and the integration step to export the preprocessed data from Brainstorm to Python for further processing steps.

MEG recordings were preprocessed in Brainstorm to prepare the data for graph construction and model training. This included anatomical co-registration, filtering, artifact rejection, and both sensor and source space analysis. The cleaned data was subsequently exported to Python for graph construction and classification analysis.

2.1. Overview

The scripts used to preprocess and export the data include both MATLAB and Python files:

- `export_bst_ftp.m` (MATLAB)
  - `process_raw_string.m`: Helper for file structure formatting.
  - `out_fieldtrip_data.m`: Brainstorm function to export to FieldTrip.
- `export_ftp_python.py` (Python): Converts exported mat-files to fif-files.
- `export_scouts.py` (Python): Converts Brainstorm source-space data to HDF5 file format.

2.2. Processing in Brainstorm

In the following subsections, the processing steps in Brainstorm are listed. These steps are to be performed inside the Brainstorm toolbox. Similar preprocessing steps can also be executed using the Python library MNE [2].

2.2.1. Anatomical Registration

1. **Warp anatomy:** CTF channels > MRI registration > Edit. Digitized head points > Warp > Deform default anatomy.
2. **Improve co-registration:** MRI registration > Refine using headpoints.
3. **Adjust reference position:** Adjust head position to median location via Process1 > Import > Channel file > Adjust coordinate system.

2.2.2. Data Cleaning and Preprocessing

1. **Convert recordings:** Import > Import recordings > Convert to continuous (CTF).
2. **Create PSD plot:** Frequency > Power Spectrum Density (Welch), with default window (4s), 50% overlap.
3. **Remove bad channels:** Identify via PSD/topography, right-click > Mark selected as bad.
4. **Mark stimulation cycles:** Use ECG, EEG, or Misc channels to visually annotate ON/OFF stimulation transitions.
5. **Identify stimulation frequency:** Use PSD on ECG/EEG/Misc signal to detect artifacts.

6. **Notch filtering:** Remove power line (50/60 Hz) and stimulation frequency via Pre-process > Notch filter.

2.2.3. Head Models and Source Estimation

1. **Noise covariance:** Use empty-room recordings > Compute from recordings.
2. **Data covariance:** Use MEG recording > Compute from recordings.
3. **Compute head model:** Sources > Compute head model > Cortex surface > MEG method: Overlapping spheres.
4. **Source estimation:** Sources > Compute sources [2018], Kernel only: one per file, Minimum norm imaging, Current density map, Unconstrained, MEG sensors only.

2.2.4. Scout Atlas and Time Series Extraction

The 'Destrieux' atlas was used as a starting point and adapted for the study. Scout time series were extracted as follows:

1. **Create/modify atlas:** Atlas > New atlas > Copy current atlas; grow or shrink scouts manually.
2. **Edit scouts:** Use Scout tab to merge, delete, or adjust scouts.
3. **Save atlas:** Atlas > Add scouts to atlas.
4. **Extract time series:** Process1 > Extract > Scout time series.

2.3. Export to Python

In order to use the preprocessed data from Brainstorm in a Python environment, an integration step is necessary. For each MEG recording, the preprocessed sensor data file, as well as the source estimate file, should be exported. For the sensor and source-based data, different steps are necessary for exporting, as described in the following subsections.

2.3.1. Sensor-Based Data

Sensor-based data were exported to Python as fif-files using the following workflow:

1. Use `export_bst_ftp.m` in MATLAB to create FieldTrip-compatible 'mat' files.
2. The script requires the subject folder (from Brainstorm) and the folder with filtered 'raw' MEG data.
3. Files are saved to a dedicated output folder (e.g., 'FT data').
4. Use `export_ftp_python.py` in Python to convert mat-files to fif-files.
5. Optional: visualize time series and PSDs in Python.

2.3.2. Source-Based Data

Source-based data were exported directly from Brainstorm as mat-files, and converted to HDF5 format for use in Python.

1. Export the time series from Brainstorm using the GUI (Figure 2.1).
2. Save these mat-files to a folder accessible by Python.
3. Use `export_scouts.py` to convert to HDF5.
4. This script checks for matching fif-files (needed for metadata alignment).

5. Optional: visualize time series and PSDs using built-in plotting functions.

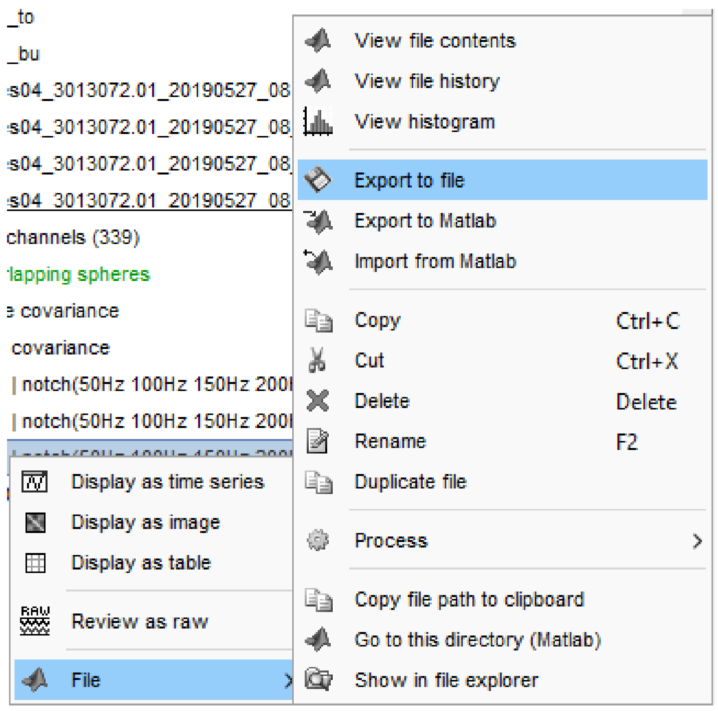
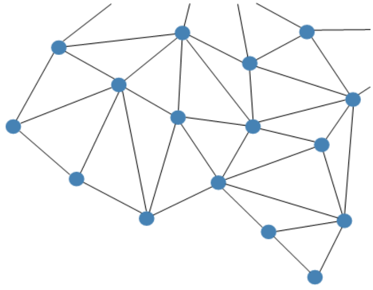


Figure 2.1: Exporting a file from Brainstorm as mat-file.



3 | Graph Datasets

This section describes the creation and structure of the graph datasets used for training and evaluating the GNN model. Each graph represents a subepoch of MEG data and encodes connectivity between brain regions either in sensor space or source space.

The input graph datasets are created with the Python script `generate_graphs.py`, that can be run on the cluster computer through the SLURM script `run_gnn_graphs.sh`.

3.1. Overview

The graph dataset creation is implemented using the following scripts:

- `generate_graphs.py`: Process files, and based on the input type and analysis, create graph dataset and save it in corresponding folder.
- `dataset.py`: Provides the MEGGraphs class for the graph dataset, in which the epoching process, graph attribute definition and calculations and artifact removal takes place.
- `dataset_wrapper.py`: A wrapper that creates a class object based on saved graph files to mimic the interface of the MEGGraphs class for downstream use.
- `run_gnn_graphs.sh`: SLURM script to create the graph datasets on a compute cluster.

The folder structure expected by the dataset generation pipeline is:

```
Data/
├── raw/
│   └── Scout_files/
├── processed/
│   └── Processed_<analysis>/
├── Raw_all
├── Scout_all
├── Connectivity/
├── MEG_PT_notex.xlsx
└── source_scouts_coordinates.xlsx
```

The folders `raw`, subfolder `Scout_files`, and the folder `processed` are primarily used for the graph creation process. Inside the folders `Raw_all` and `Scout_all`, a copy of all fif-files and source data files in HDF5 format are stored, respectively.

The `Connectivity` folder stores the calculations of the connectivity for every subepoch of data (representing all of the edges of one graph). This is implemented to decrease the run time necessary, since the connectivity calculations are a large proportion of the total run time. Whenever the connectivity computations for the subepochs of data are already saved, these files will be loaded and used, and further computations will be skipped. It is important to note that with a change to the time-series data, these saved computation files should be deleted to renew the connectivity computations to ensure the validity of this graph attribute.

The file `MEG_PT_notes.xlsx` contains metadata for each MEG recording, including:

- The stimulation frequency (used for notch filtering)
- Time points where stimulation was turned OFF

These OFF time points are used to define stimulation ON and OFF segments, from which graphs are constructed.

The file `source_scouts_coordinates.xlsx` is used for creating visualizations of the graphs, and provides 2D-coordinates for all of the sources.

3.2. Dataset Parameters

Graph generation is handled by the MEGGraphs class in `dataset.py`. The following parameters can be specified to control how graphs are created:

- **Input type:** Whether to use sensor-level data (`'fif'`) or source-level data (`'scout'`).
- **Root directory:** Path to the root data folder (typically `'Data/'`).
- **Filenames:** List of MEG recordings to include.
- **Stimulation info:** Dictionary extracted from `MEG_PT_notes.xlsx` to segment data.
- **Duration:** Length (in seconds) of each subepoch.
- **Overlap:** Overlap (in seconds) between consecutive subepochs.
- **Connectivity method:** Method used to define edges in the graph.
- **Minimum/maximum frequency:** Frequency range used to compute graph attributes.
- **Frequency resolution:** Frequency bin width (default: 1 Hz).
- **Ramp time:** Time (in seconds) excluded at the start and end of each stimulation cycle to allow ramp-up/down effects.
- **Scout data list:** Required when using source-level data; contains precomputed scout files.
- **Processed directory:** Where to save the generated graph files. When performing multiple analyses, separate directories are recommended.

3.3. Different Analyses

Multiple analyses are defined based on varying parameters or file subsets. Each analysis results in a separate graph dataset. The analyses include:

- **Full band:** 1–100 Hz (analysis name: ”Full”)
- **Delta band:** 1–4 Hz (analysis name: ”Delta”)
- **Theta band:** 4–8 Hz (analysis name: ”Theta”)
- **Alpha band:** 8–12 Hz (analysis name: ”Alpha”)
- **Beta band:** 12–30 Hz (analysis name: ”Beta”)
- **Gamma band:** 30–100 Hz (analysis name: ”Gamma”)
- **Low band:** 1-30 Hz (analysis name: ”Low”)
- **Tonic stimulation:** Only recordings with tonic stimulation (analysis name: ”TONIC”)

- **Burst stimulation:** Only recordings with burst stimulation (analysis name: ”BURST”)
- **Canada:** Only recordings from the MNI in Canada (analysis name: ”Canada”)
- **Nijmegen:** Only recordings from the Donders Institute in Nijmegen (analysis name: ”Nijmegen”)

For each analysis, the correct frequency band is automatically selected, and the relevant recordings are chosen based on filename structure.

To run an analysis on the cluster, the analysis name can be passed to `run_gnn_graphs.sh`, or a SLURM array can be used to process multiple analyses in parallel.

3.4. Processed Graphs

Each subepoch of MEG data is converted into a graph and saved using the naming convention:

`graph_<patient_prefix>_<stimulation_type>_<graph_index>.pt`

Where:

- `patient_prefix` represents the prefix of the dataset and the patient number e.g. `PT01`, `PTN03`, etc.
- `stimulation_type` is either `tonic` or `burst`, representing the stimulation type
- `graph_index` is the index of the subepoch for that MEG recording

The graphs are saved in the corresponding processed folder (`Processed_<analysis>/`). Graphs are also copied to the `all_graphs/` subdirectory of the processed folder.

3.4.1. Bad Subepoch Detection

During processing, subepochs with poor signal quality are automatically identified using a thresholding method described in the thesis. These subepochs are replaced with empty graphs and moved to the `bad_subepochs/` folder. These graphs are excluded from further steps, including model training and evaluation.

3.5. Remarks

- The graph generation process supports both sensor and source space analyses, which can be specified through the input type parameter.
  - For the sensor space, the MEG data from the fif-files is directly used, and scaled to femtoTesla (fT) using a scaling factor of 10^{15}.
  - When extracting the source data with the Minimum Norm Imaging approach, the unit is Ampere-meter (A-m), which is scaled with a factor of 10^{12} to acquire the unit picoAmpere-meter (pA-m) to obtain logically scaled values.
- Each analysis results in a fully independent graph dataset saved inside a separate folder, so different analysis can be processed in parallel.
- The pipeline is modular and easily adjustable with different parameters (such as subepoch duration, overlap, connectivity metric, etc.).
- The preprocessed dataset that was used in the most recent analysis for this project is located inside the folder ’`Data\ Scout_all\ Scout_58_MN`’.

4 | Graph Neural Network

This section describes the GNN model used in this project to classify MEG-based graph representations into stimulation ON and OFF states. The GNN is a graph classification model that takes brain connectivity graphs as input and outputs a binary prediction for each graph.

4.1. Overview

The training pipeline is implemented using the following scripts:

- `train_from_processed.py`: Loads a processed graph dataset, splits the data into training and test sets, and initiates training.
- `train.py`: Contains the model definition, hyperparameter optimization steps, training loop, loss computation, evaluation, and saving of outputs.
- `run_gnn_train.sh`: SLURM script to run the training on a compute cluster.
- `test_saved_model.py`: Loads a previously trained a saved model, to test a dataset with this model.
- `test_gnn_model.sh`: SLURM script to test a saved model with a specified dataset on a compute cluster.

The output includes the final model evaluation, performance metrics, model parameters, and visualizations.

4.2. Model Architecture

The GNN model architecture consists of the following layers:

1. **Input Layer:** Accepts a graph consisting of a node feature matrix, edge indices, and edge weights. Each graph corresponds to a MEG recording under a specific stimulation condition.
2. **Graph Convolutional Layers:** Multiple layers of graph convolutions using the operator proposed by Kipf and Welling [3]. These layers iteratively update node feature representations by aggregating information from neighbours and self-loops.
3. **Readout Layer:** A global mean pooling operation is applied to aggregate node embeddings into a single graph-level embedding.
4. **Output Layer:** A final linear layer maps the graph embedding to a binary output: stimulation ON or OFF.

4.3. Model Training

Before training, the dataset of graphs is split into a training and a test set. A stratified grouped K-fold split is used to ensure an even distribution of classes (in this case, stimulation states), and to prevent data leakage across groups. A five-fold cross-validation scheme is employed, where the model is trained five times on a different training set, and, subsequently, tested on the test sets.

The training set is split into a training and a validation set using a similar splitting approach. The validation set is used to evaluate the model training process and to evaluate the performance of different model parameters, called hyperparameters.

Some details on the model training that are used in this project are listed below.

- **Loss:** Binary Cross-Entropy Loss
- **Optimizer:** Adam
- **Evaluation:** Accuracy, Precision, Recall, F1 Score, Receiver Operating Characteristic (ROC) Area Under the Curve (AUC), confusion matrix
- **Hyperparameters:** tunable parameters including learning rate, dropout rate, hidden channels, batch size, number of layers, edge filtering approaches

4.4. Outputs

After training, the following outputs are stored inside the folder `Output/<model_name>/`:

- `training_results_<model_name>.xlsx`: Overall test results (across all five folds, average and standard deviation), overall confusion matrix, overall ROC curve data, hyperparameter results, and the best performing hyperparameters.
- `roc_data_<model_name>.xlsx`: Data for plotting ROC curves.
- `Output/<model_name>/plots/`: Plots of the cross-validation splits, training and validation loss and accuracy curves, ROC curves, and confusion matrix
- `best_model<model_name>.pt`: Trained model weights and checkpoints of the best performing model, saved after training all configurations.
- `best_result<model_name>.pt`: Saved parameters of the best performing model, saved after training all configurations.

4.5. Testing with External Test Set

After training a GNN model, the model's weights and parameters are saved, and can be reloaded to test a different dataset on this saved model. The Python script `test_saved_model.py` can be called through `test_gnn_model.sh`, and loads a saved model and tests it with a selected dataset. The path to the saved model and the path to the dataset to be tested are provided as inputs, and outputs are stored inside a '`cross_performance_<analysis_name>`' folder, in which the performance metrics are listed. Inside the SLURM script, only the name of the saved model (i.e. "TONIC", "BURST", "Nijmegen", "Canada") has to be specified, as well as the name of the graph dataset used for testing.

5 | Explainability

To gain insight into which regions of the brain most influenced the model's predictions, explainability analyses were performed on the test graphs using the SubgraphX approach. This method provides both local and global explanations by identifying key nodes and substructures that contributed to the classification outcomes.

5.1. Overview

The following Python scripts were used for model explainability:

- `model_explainability.py`: Main script to load the model, apply explainability, and save results.
- `explain.py`: Contains the core logic for SubgraphX-based explanation, plotting, and aggregation.
- `subgraph_x.py`: Explainability method, as proposed by Yuan et al. [4].
- `mcts.py`: Performs a Monte Carlo Tree Search (MCTS).
- `shapley.py`: Applies the Shapley scoring method within the MCTS context.
- `task_enum.py`: Provides configurations for the explainability method.
- `run_gnn_explain.sh`: SLURM script to run the explainability module on a saved GNN model.

5.2. Explainability Pipeline

The explainability pipeline consists of the following steps:

- **Load graphs and saved model:** Based on the specified directories, the script loads the corresponding graphs using the dataset wrapper class and retrieves the trained model for the corresponding analysis.
- **Explainability per file:** Apply explainability analysis on a per-file basis. The following inputs should be provided:
  - The directory to the saved graphs that will be explained through the explainability approach
  - A path to the saved model and its corresponding saved parameters
  - Path to an output Excel file
  - A list of specific files to analyse (i.e. "PT06_BURST" "PT06_TONIC" "PT07_BURST" ...)
  - The input type (sensor or sources)
  - The number of nodes to include in the subgraph (n_min=6)

  Due to the high computational resource demands of this process, only a limited number of files are analysed at a time.
- **Visualization outputs:** For each file, an explainability graph is saved as a PNG file, highlighting the identified subgraph of overall most important nodes.

- **Excel report:** Each explanation per file is also saved in an Excel file, containing average fidelity and sparsity scores corresponding to the identified subgraphs, and including the node labels of the overall most important nodes per file and in total.

5.3. Output Structure

Explanation results are saved to `Output/<model_name>/`, containing:

- `explanations_gnn_<analysis_name>.xlsx`: Excel sheet containing the outcomes per file, including the most frequently occurring important nodes, the average fidelity score, and the average sparsity score across all analysed samples for this file.
- PNG files for each file, highlighting the identified overall subgraph, saved inside the subfolder `/explainability_p`
- An overall plot summarizing the explainability outcomes across all graphs.

6 | Dependencies

6.1. Software

The software used during this project is the following:

- Brainstorm: open-source Matlab application, used with Matlab 2023b.
- Python 3.11.11, used with a variety of Python's scientific libraries, as included in the conda environment (`requirements_gnn_env.yaml`). Specific instructions on how to create this environment will be detailed below.

6.2. Installing requirements

All requirements listed in `requirements_gnn_env.yaml` will need to be installed to be able to run the Python code. First, Anaconda needs to be installed on the running system or compute cluster (see https://www.anaconda.com/download/success). To create a virtual environment in conda with all required packages ('meg_gnn_env'), use the following command in Anaconda Prompt:

```
conda env create --name meg_gnn_env --file requirements.yaml
```

The environment will be created in the Anaconda directory on your running system (i.e. `~anaconda3\envs\GNNenv`), which will also hold the preferred version of Python (in this case Python 3.11).

Then activate the virtual environment using:

```
conda activate meg_gnn_env
```

Add new packages using either conda or pip, as needed:

```
conda install <package_name>
pip install <package_name>
```

Saving these requirements as a '.yaml' file can be done using this command:

```
conda env export > <path-to-project>\requirements_gnn_env.yaml
```

6.3. Running on local environment

For running this project on a laptop, a code editor can be used to run the Python scripts. When opening the project files in VSCode (or any other preferred code editor), make sure to select the virtual environment when running the code. In VSCode, you can do this by pressing "ctrl"+"shift"+"P" and click on "Python: Select Interpreter" to select the 'meg_gnn_env' environment.
Note: ensure that your device can handle long directory paths (for saving the hyperparameters) using https://www.howtogeek.com/266621/how-to-make-windows-10-accept-file-paths-over-260-characters/.

6.4. Running on a Compute Cluster

A compute cluster provides significantly more processing power and memory compared to a local machine, which is especially beneficial for large-scale graph generation, model training, and explainability analyses in this project. All cluster jobs were managed using `SLURM` job scripts, which specify computing resources, conda environment setup, and task execution.

More information and the documentation of the cluster computer that was used for this project, including instructions on the command line usage for copying files to the cluster, can be found here: https://doc.dhpc.tudelft.nl/delftblue/. Furthermore, other useful command line instructions can be found here: https://swcarpentry.github.io/shell-novice/index.html. For a more thorough understanding, it is recommended to participate in a course focused on introducing high-performance computing and command line (https://www.tudelft.nl/cse/education/courses/linux-cli-101).

6.4.1. Workflow

The general workflow for running jobs on the cluster was as follows:

1. **Prepare input files:** Copy necessary datasets, trained models, and Python scripts to the appropriate directories on the cluster storage (be aware of the amount of storage available in the cluster, and if necessary, use the temporary storage folders, e.g., `/scratch/` folders).
2. **Edit the SLURM script:** Specify the correct input parameters inside the corresponding `.sh` script, such as the processed data directory, model name, or analysis type.
3. **Submit the job:** Run the script using the `sbatch` command:

   ```
   sbatch run_gnn_train.sh
   ```

4. **Monitor progress:** Track job status with the command '`squeue`' and review output or error logs stored in the `Output/` directory. Alternatively, use the command:

   ```
   sacct -j <JOB-ID> --format JobID, Elapsed, State
   ```

5. **Retrieve results:** Processed graphs, trained models, performance outputs, and explanation results are stored in their respective output folders for further analysis.

6.4.2. Available SLURM Scripts

The following SLURM scripts were used for this project:

- `run_gnn_graphs.sh`: Generates graph datasets from preprocessed MEG data for different analysis types (e.g., frequency bands, stimulation types, recording institutes).
- `run_gnn_train.sh`: Trains a GNN model for each analysis type using the generated graphs.
- `run_gnn_explain.sh`: Applies the SubgraphX explainability method to trained models for selected patient files, producing both plots and Excel reports.
- `test_gnn_model.sh`: Tests a saved model on a different dataset to evaluate cross-dataset performance, optionally using GPU resources.

6.4.3. Job Array Usage

For both graph generation and training, SLURM job arrays were used to efficiently run the same script on multiple analysis configurations in parallel. Each analysis type was stored in an array and accessed using:

```
ANALYSIS_NAME=${analysis_names[$SLURM_ARRAY_TASK_ID]}
```

This allowed parallel processing for all analysis, where all analyses can be started simultaneously by submitting the job. A single analysis can be run by changing the `ANALYSIS_NAME` parameter to any preferred analysis, i.e. `ANALYSIS_NAME="Full"`, and removing the SLURM array definition in the header.

6.4.4. Practical Notes

- The `Mobaxterm` terminal can be used for a convenient SSH interface to the cluster.
- Resource allocation (CPU cores, memory per core, time limit) is defined in the script header to optimize job performance. The existing SLURM scripts are somewhat optimized for their specific jobs.
- Output (`.out`) and error (`.err`) logs are automatically created in the `Output/` directory and are useful for debugging.
- For large explainability runs, only a few patient files are processed per job to avoid exceeding resource limits.

7 Plotting

For generating the plots included in the thesis report, two separate Python scripts and corresponding SLURM scripts are created.

- **`plot_graph_analysis.py`**: This script can be called through `plot_graphs.sh`, and creates plots for the averaged node features (for three selected nodes), the connectivity matrices for stimulation ON and OFF, averaged over all graphs, the difference in connectivity matrix between OFF and ON, and a graph visualization.
  - Input: directory to a folder containing saved graphs.
  - Output: inside a subfolder ’`plots`’, the PNG files are saved for each plot.
- **`roc_plots.py`**: The script can be called through `plot_roc.sh`, and created ROC curves after all GNN models have been trained and saved (for all analysis). The code automatically searches for the ’Output/<analysis_name>’ folders containing Excel sheets with the ROC curve data, and creates a folder ’Output/ROC_Analysis’ containing two plots.
  - `roc_curves_frequency_bands.png`: ROC curve plot containing the results for all frequency bands.
  - `roc_curves_stim_site.png`: ROC curve plot containing the results for the GNN models trained on tonic and burst stimulation separately, and the GNN models trained on recordings from either the Montreal Neurological Institute (MNI) or Donders Institute.

References

[1] C. Witstok, "Unravelling Brain Networks in Chronic Pain and Spinal Cord Stimulation through Magnetoencephalography and Graph Neural Networks," Master's thesis, Sep. 2025. [Online]. Available: https://resolver.tudelft.nl/uuid:28a426ae-582c-47d4-88e1-f96412f09610.

[2] *MNE — MNE 1.9.0 documentation*. [Online]. Available: https://mne.tools/stable/index.html (visited on 01/22/2025).

[3] T. N. Kipf and M. Welling, *Semi-Supervised Classification with Graph Convolutional Networks*, en, Feb. 2017.

[4] H. Yuan, H. Yu, J. Wang, K. Li, and S. Ji, "On Explainability of Graph Neural Networks via Subgraph Explorations," en, in *Proceedings of the 38th International Conference on Machine Learning*, ISSN: 2640-3498, PMLR, Jul. 2021, pp. 12 241–12 252.