Research Internship Report

Machine Learning for Seizure Onset Zone Detection from sEEG-data

Zacharie Rodière

May 2025

Physics Laboratory, ENS de Lyon

Abstract

For drug-resistant focal epilepsy, neurosurgical resection of the epileptogenic zone (EZ) offers a curative treatment. Accurate localization of the Seizure Onset Zone (SOZ) via stereo-EEG (sEEG) analysis—intracranial recordings from multi-contact electrodes—is crucial for surgical planning. We develop a novel SOZ detection model using a Transformer encoder with spatial attention, complemented by a new spatial contrastive pre-training framework designed to learn channel-specific representations. This framework, applied to heterogeneous sEEG records (ictal and interictal) from diverse patients, encourages distinct representations for SOZ versus non-SOZ channels. Subsequent fine-tuning for classification employs a channel-wise focal class-balanced Binary Cross-Entropy loss to effectively address the inherent class imbalance between the few SOZ channels and numerous non-SOZ channels.

Acknowledgments

I sincerely thank my advisors, Pierre Borgnat and Paulo Gonçalves, for the exceptional opportunity to immerse myself in the research environment at ENS de Lyon. My gratitude also extends to the vibrant Ockham team PhD students – Arthur, Çan, Anne, Maël, and Edgar – and my fellow intern, Gabriel Sanchez. These six months were filled with learning, camaraderie, and stimulating challenges. The dedication, intellectual curiosity, and passion for research I observed within the team have left a lasting impression on me, and I am deeply thankful for that experience.

마지막으로 사랑하는 가희에게, 이 보고서를 완성하는 동안 인내심과 격려를 보내주셔서 감사합니다. 이 일이 얼마나 중요한 일인지 네가 이해해주고 곁에 있어줘서 정말 큰 힘이 되었어.

Contents

1 Introduction

Epilepsy is a significant neurological disorder affecting millions globally. For patients with drug-resistant epilepsy, surgical resection of the Seizure Onset Zone (SOZ) offers a potential cure, but its success hinges on the accurate localization of this zone. Stereo-electroencephalography (sEEG) provides high spatio-temporal resolution intracranial recordings crucial for SOZ identification. However, the visual analysis of sEEG signals is a complex, time-consuming task, susceptible to inter-observer variability, underscoring the need for automated, data-driven methods.

This report details a 6-month research internship at ENS de Lyon. The primary objective was to develop and evaluate machine learning techniques for robust SOZ detection from sEEG data. Initial investigations explored anomaly detection using learned representations from a Diffusion Convolutional Gated Recurrent Unit (DCGRU), a model designed to exploit graph structures computed from sEEG channel connectivity measures. While this approach provided valuable insights, it was ultimately not pursued to completion.

The core contribution of this internship, and the main focus of this report, is the development of a novel approach utilizing a transformer encoder architecture applied to sequentialized sEEG channels. This model leverages a balanced contrastive pre-training strategy to learn effective representations from the sEEG signals, followed by a balanced classification fine-tuning stage for the final SOZ detection task. Preliminary results from this approach are promising and were presented at the Graph Signal Processing Workshop (MILA, Montréal, 2025).

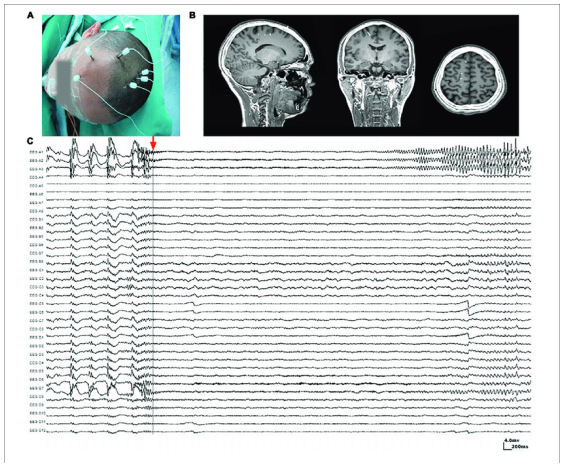


Figure 1: sEEG electrode setup and example recordings. Figure from [1]

2 Internship Organization and Context

The internship was formally conducted under a *convention de stage* with ENS de Lyon from Nov. 5 2024 to May 17 2025. My primary advisor was Pierre Borgnat, a Senior Research Director at CNRS (*Centre National de la Recherche Scientifique*) within the SIgnaux, SYstèmes et PHysique (SISYPHE) team of the Physics Laboratory. I was co-advised by Paulo Gonçalves from Inria (*Institut National de Recherche en Informatique et Automatique*).

Computing resources for large-scale dataset management and model training were provided by ENS de Lyon, with technical support from Emmanuel Quemener. The research environment at the Physics Laboratory facilitated rich scientific exchange. Interactions included participation in PhD defenses, weekly Machine Learning and Signal Processing seminars, Monday morning mathematical meetings, selected ENS classes, and valuable informal discussions with PhD students specializing in machine learning. Regular progress meetings, typically held weekly, were conducted with my advisors, supplemented by email updates on significant developments.

This internship contributes to a larger multiyear project funded by the French National Research Agency (ANR, *Agence Nationale de la Recherche*), project SEIZURE. This ANR project involves collaboration with researchers from the Lyon Neuroscience Research Center (CRNL, *Centre de Recherche en Neurosciences de Lyon*), particularly Julien Jung, who was instrumental in bringing this project to fruition through his essential contributions to data acquisition. The overarching goal of this collaborative effort is to develop a multi-modal model for improved SOZ detection, aiming to radically enhance surgical outcomes for epilepsy patients. Within this framework, the specific focus of my work was the development of an effective model for the sEEG modality. ANR project meetings were held three times a month in a CRNL library in the eastern Lyon hospitals.

The internship lead to the submission of an extended abstract at the Graph Signal Processing Workshop (GSP Workshop), held in MILA, Montréal this year, where I was able to present my work in the form of a poster.

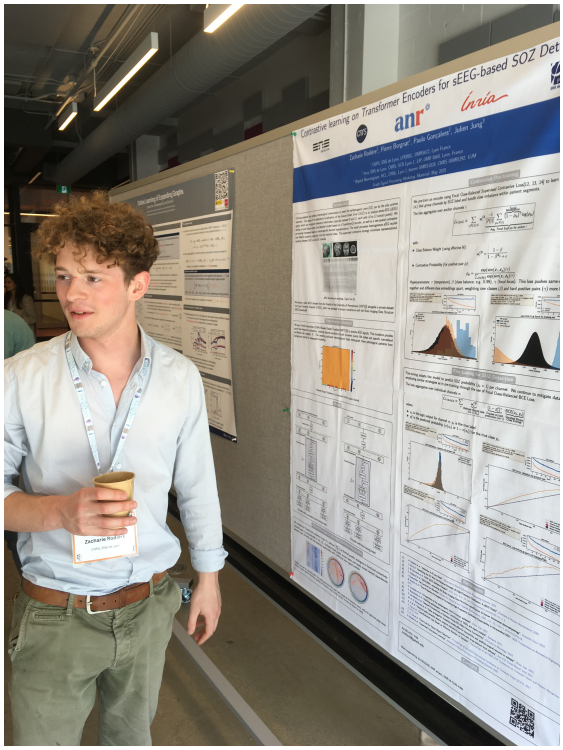


Figure 2: Poster presentation at the GSP Workshop in Montréal. Thanks to Pierre Borgnat for the picture!

3 Working with sEEG data

3.1 Understanding the sEEG modality

Stereotactic electroencephalography (sEEG) is an invasive neurophysiological monitoring technique that involves implanting multiple thin electrodes directly into deep brain structures. Unlike scalp EEG, which measures electrical activity from the surface of the brain, sEEG provides highly localized and precise recordings of neuronal activity from within the brain parenchyma. This allows clinicians to pinpoint the exact origin and spread of epileptic seizures, a critical step in surgical planning for patients with drug-resistant epilepsy. The three-dimensional placement of electrodes, guided by pre-operative MRI and CT scans, enables the mapping of complex seizure networks that might be missed by less invasive methods, offering a detailed understanding of the seizure onset zone (SOZ) which is the area of the brain where seizures originate.

Detecting the SOZ using sEEG relies on analyzing the unique electrophysiological signatures of seizure activity. Traditional analysis techniques often involve visual inspection of raw sEEG traces by experienced epileptologists, looking for characteristic patterns such as low-voltage fast activity, rhythmic spiking, or high-frequency oscillations that precede clinical seizure onset. Beyond visual assessment, quantitative methods are employed to enhance SOZ localization. For instance, spectral analysis can identify changes in specific frequency bands (e.g., increased gamma or ripple activity) associated with seizure initiation. Connectivity analysis, using metrics like coherence or Granger causality, helps to map the propagation of seizure activity by identifying brain regions that become synchronously active or drive activity in other areas. Furthermore, source localization techniques attempt to infer the precise spatial origin of electrical activity from the sEEG signals, providing a more refined anatomical localization of the SOZ. These combined approaches help to triangulate the SOZ, guiding neurosurgeons in resecting or ablating the epileptogenic tissue while sparing eloquent brain regions.

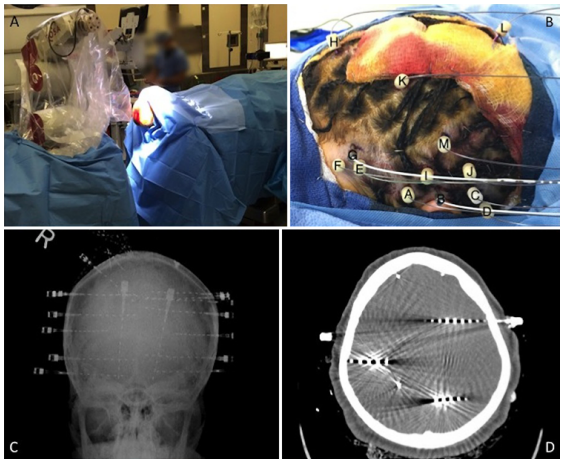


Figure 3: Pediatric sEEG implantation with a ROSA surgical robot. Figure from [2]

The development of stereo-electroencephalography (SEEG) as a precise method for exploring deep brain activity, particularly in epilepsy, is inextricably linked to the pioneering work of French neurosurgeon Jean Talairach and neurologist/neurophysiologist Jean Bancaud in Paris, primarily from the 1950s through the 1970s[3]. Talairach developed the stereotactic methodology and the proportional grid system (the basis of the Talairach Atlas) that allowed for the precise and reproducible three-dimensional targeting and implantation of electrodes deep within the brain. Bancaud then championed the clinical application of this technique, meticulously correlating the intracerebral electrical recordings with clinical seizure semiology and anatomical localization to define the epileptogenic zone. Their collaborative approach at Sainte-Anne Hospital revolutionized the presurgical evaluation of patients with drug-resistant focal epilepsy, moving beyond surface EEG limitations to directly investigate the origin and spread of seizures within complex brain networks, thereby resulting in more effective surgical interventions.

3.2 Data formats

A variety of recording formats exist for stereo-EEG (sEEG), with the most prominent detailed in the subsequent table. The sEEG data accessed from Lyon University Hospital (CHUL) was originally in Micromed (.trc) format, supplemented by Brainstorm[4] exports containing additional metadata. SOZ information was provided in an Excel spreadsheet, before being included in the Brain Imaging Data Structure-compliant dataset. All data was subsequently reorganized according to the iEEG-BIDS standard [5] and converted to the BIDS-compatible EDF format. This semi-manual conversion, involving both automated and manual steps, required approximately one week to complete. The primary model, as described

in the DCGRU section, was first implemented using Micromed data, necessitating the creation of specific PyTorch Datasets to accommodate its original organizational scheme. A subsequent and significant effort involved transitioning to a more standardized dataset. This refined dataset is intended to serve as a reusable asset for future laboratory members, including interns and PhD candidates, with the aspiration that its eventual enlargement and publication will benefit the broader research community. BIDS compliance was verified using the BIDS validator. An other open-access dataset from the Hospital of the University of Pennsylvania (HUP)[6] was used to complement our French dataset.

Format Name	File Ext(s).	Structure	Data Res.	Max Ch. & Rate	Metadata Handling	Key Specificities
Micromed (.trc)	.trc	Proprietary, single binary file.	Typically 16-bit.	Dependent on hardware (e.g., up to 256 ch, 16 kHz).	Basic channel labels. Limited std. metadata for electrode coords.	Widely used in clinical settings with Micromed systems. Conversion often needed for advanced analysis.
Brainvision Core	.vhdr, .vmrk, .eeg	Three separate files: text header, text marker, binary data.	16-bit or 32-bit float.	High; up to 256+ ch, up to 100 kHz (hw dep.).	Flexible; basic info in header. Often supplemented for detailed coords.	BIDS compatible. Multi-file for rich event/metadata. 32-bit float preserves fidelity.
European Data Format (EDF/EDF+)	.edf, .edf+	Single binary file, text header.	16-bit signed int.	Up to 256 ch (pract.). Rate varies per ch.	EDF+ allows annotations for basic events/metadata. Electrode locs not std.	Open standard. Simple, widely supported. 16-bit res. can limit. EDF+ crucial for events.
Natus Nicolet	.e, .eeg	Proprietary binary format.	Typ. 16-bit or 32-bit.	Dependent on hw (e.g., up to 256 ch, 2 kHz+).	Stores basic channel info. Detailed sEEG metadata often separate.	Common in clinics with Natus/Nicolet. Data often needs conversion for research.
Blackrock Microsystems	.nsx, .nev	Two files: .nsx (continuous), .nev (spike/event).	16-bit int.	Very high; hundreds of ch, up to 30 kS/s per ch.	Basic header. Detailed metadata (electrode maps) via software.	High-density. Optimized for high-ch, high-rate research. Dual-file separates data.
Neurodata Without Borders (NWB)	.nwb	Single HDF5-based binary file.	32-bit or 64-bit float.	Virtually unlimited; for very large datasets.	Comprehensive standard. Dedicated structures for electrode groups, localizations (x,y,z), regions.	Future of neurophys. Modern, open-std for FAIR data. Stores raw, processed, extensive metadata. Ideal for complex sEEG.

Table 1: Comparison of Different Data Formats for Stereo-EEG Recordings

3.3 Overview of the iEEG-BIDS CHUL dataset

The CHUL dataset adheres to iEEG-BIDS[5], a clear hierarchical structure designed for standardized neuroimaging data organization and sharing. At the dataset root level, essential descriptive files are present, including `dataset_description.json` (providing global metadata), `participants.tsv` and `participants.json` (detailing subject-level information), and dataset-wide metadata like `channels.json` or

`events.json`, applicable to all subjects and sessions. Data for each individual participant is then segregated into uniquely labeled directories, such as `sub-CHUL01`, `sub-CHUL02`, and so on. Within each of these participant directories (e.g., `sub-CHUL01`), data is further organized by imaging modality. In this specific case, all data falls under the `ieeg` subdirectory, indicating intracranial EEG recordings. Inside the `ieeg` directory, individual recording runs are distinguished by standardized filenames incorporating key-value pairs like `sub-CHUL01_task-ictal_run-01_ieeg.edf`. These filenames clearly specify the subject, task (e.g., ictal, interictal), and run number. Accompanying each primary iEEG data file (e.g., `.edf`) are associated metadata files, such as `_channels.tsv` (describing electrode properties), `_events.tsv` (annotating experimental events), and `_ieeg.json` (a "sidecar" JSON file containing acquisition parameters specific to that recording). Additionally, each subject directory contains a `_scans.tsv` file, which lists all acquisition files for that participant and can link to their specific metadata. This consistent, multi-level organization ensures that data and its associated metadata are logically grouped and easily discoverable, facilitating both manual inspection and automated processing pipelines.
The reference manual for iEEG-BIDS can be found here.

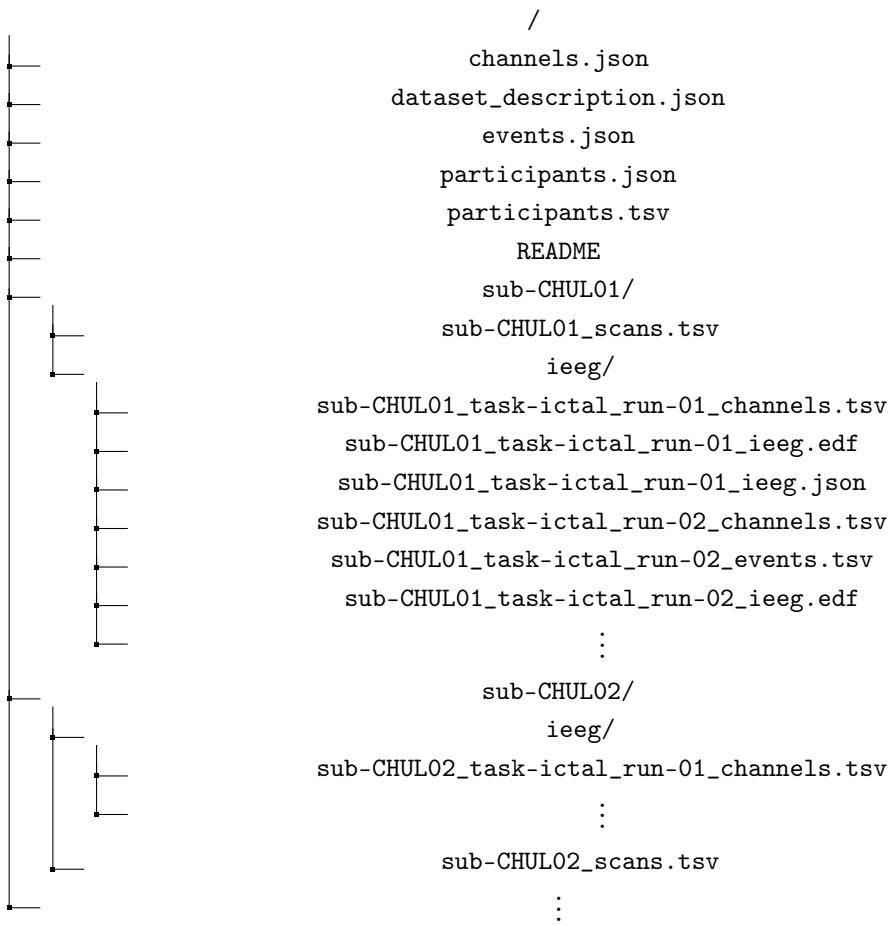


Figure 4: Hierarchical Structure of the CHUL iEEG-BIDS Dataset

3.4 Time-Frequency Features as Effective Inputs

We chose to use the Daubechies-4 (Db4) Wavelet Packet Transform (WPT)[7] as time-frequency features for processing sEEG signals from each channel, employing 5 levels of decomposition. This approach offers a highly effective methodology for analyzing seizure-related activity due to its inherent ability to generate localized time-frequency representations, which are indispensable for identifying key epileptiform biomarkers such as spikes, ripples[8], and High-Frequency Oscillations (HFOs)[9]. These biomarkers are often transient and spectrally overlapping phenomena, and their precise characterization in both time and frequency is paramount for understanding the underlying neural dynamics. The WPT achieves this by providing a detailed tiling of the time-frequency plane, allowing for the precise determination of when a specific frequency component occurs—a critical ca-

pability for analyzing such short-lived events.

The Daubechies-4 wavelet itself contributes specific advantages to this analysis. Its compact support, meaning its short duration in the time domain, is particularly well-suited for capturing the sharp, transient nature of epileptic spikes, enabling excellent temporal resolution by closely matching their morphology. Simultaneously, the Db4 wavelet's regularity, or smoothness, aids in distinguishing these pathological activities from the more regular background brain rhythms, effectively suppressing smoother, non-epileptic components and thereby enhancing the prominence of abnormal signals. The implicit orthogonality of Daubechies wavelets further ensures that the extracted wavelet coefficients represent distinct, non-redundant pieces of information, which benefits subsequent feature extraction and modeling.

Beyond the properties of the Db4 wavelet, the Wavelet Packet Transform structure significantly enhances spectral resolution, especially for HFOs and ripples. Unlike the standard Discrete Wavelet Transform, the WPT decomposes both approximation and detail coefficients at each level, resulting in a more uniform and finer-grained partitioning of the frequency spectrum. With 5 levels of decomposition, the spectrum is divided into 32 sub-bands. This fine partitioning enables the isolation of narrowband oscillations into distinct WPT nodes. Such granularity is crucial for disentangling spectrally overlapping components, which might otherwise obscure these subtle biomarkers in raw signal analysis or traditional spectral methods.

Ultimately, the Db4 WPT transforms the sEEG signal into a rich feature space where pathological patterns are simultaneously time-localized and frequency-resolved. This allows machine learning models to identify more discriminative patterns: ripples and HFOs can manifest as sustained energy in specific high-frequency subbands, while spikes appear as transient energy bursts across adjacent bands. This enhanced representation offers superior discriminative power compared to methods relying only on static spectral features or time-domain morphology.

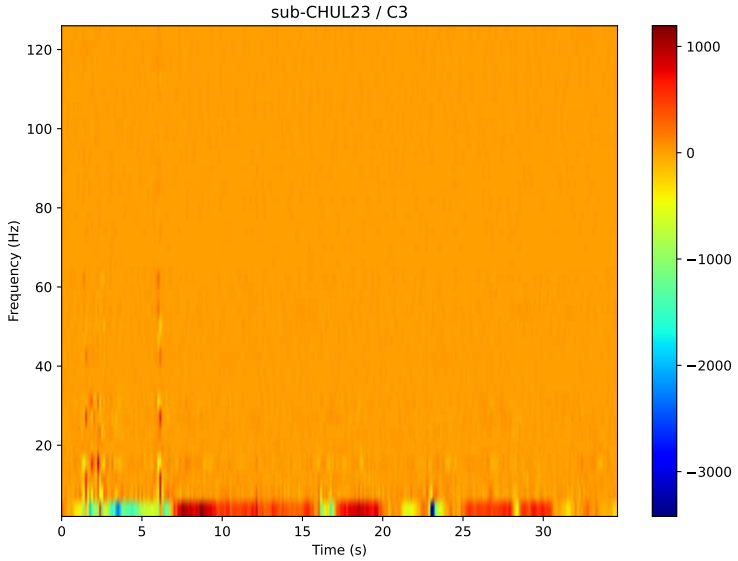


Figure 5: Db-4 WPT of a 35 second window of a given channel for patient sub-CHUL23

3.5 PyTorch implementation

To facilitate this work, PyTorch [10] implementations of the Daubechies-4 Wavelet Packet Transform (WPT) for an arbitrary number of decomposition levels (see Appendix A.1.1) and a custom Dataset class (Appendix A.1.2) were developed. This custom class handles notch-filtering, global indexing of fixed-length clips from the HUP and CHUL datasets, and on-the-fly WPT computation. For each retrieved item, it provides the patient ID, WPT features, channel names, and SOZ multi-hot labels. Such data engineering was foundational to the project's success. A special collate function is also provided, to ensure compatibility with Dataloaders.

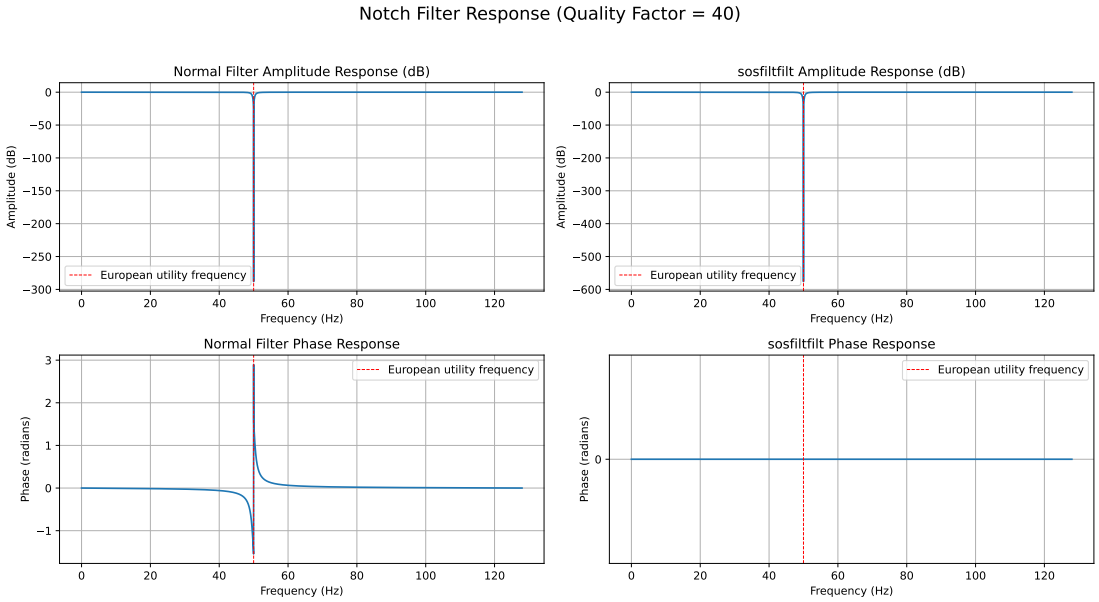


Figure 6: Mains frequency interference in CHUL signals is removed using a notch filter applied with a forward-backward technique (filtering again the time-reversed filtered signal) to ensure zero phase distortion.

4 First approach: Modeling Anomalies on a self-supervised DCGRU model

We began by training a Diffusion Convolutional Gated Recurrent Unit[11] (DCGRU), a recurrent graph neural network based on the Diffusion Convolutional Recurrent Neural Network [12]. This model was designed for a self-supervised prediction task, where it minimized a regression loss between the Discrete Fourier Transforms (DFTs) of actual and predicted signals. The subsequent step intended to utilize normalizing flows [13]–[16] for anomaly detection on the encoder's output. However, despite successfully training the self-supervised prediction model, our experiments did not progress significantly in this direction. Initial observations of regression prediction error peaks showed no consistent correlation with Seizure Onset Zone (SOZ) localization, as the error appeared constant over time.

Consequently, we shifted to a second approach that proved considerably more promising. This involved exploiting label information and capturing long-distance dependencies using a full-attention transformer encoder, enhanced with spatial attention and contrastive pre-training. This methodology is detailed in the subsequent section.

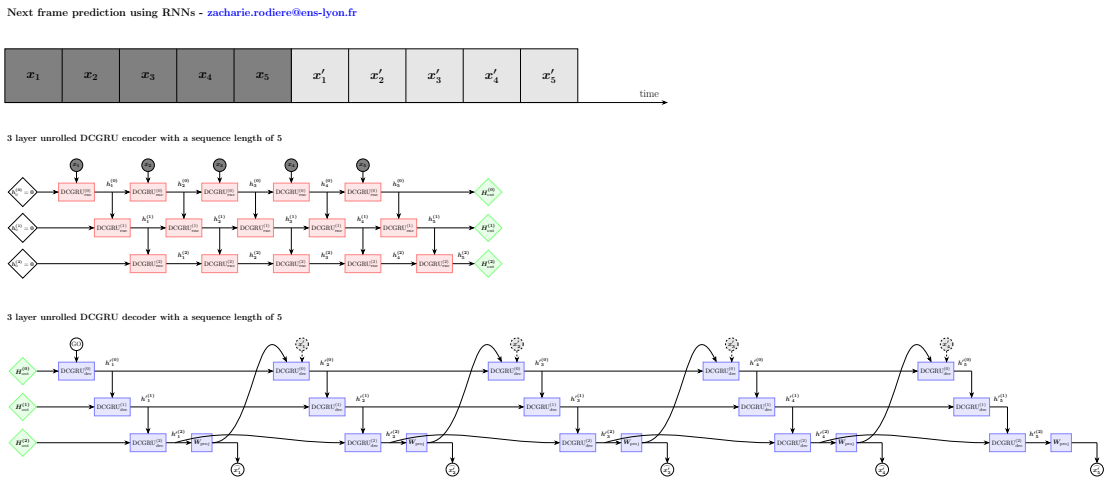


Figure 7: An unrolled DCGRU recurrent neural network, with a sequence length of 5 and a network depth of 3. In practice, I trained a network with a sequence length of 12 and a depth of 3.

4.1 Diffusion convolution operation

This model, developed by Siyi Tang and collaborators from Stanford University, incorporates a diffusion process over a graph linking EEG electrodes, allowing for self-supervised spatio-temporal modeling of EEG signals. This approach naturally adapts to the sEEG modality, a crucial step for the implementation of which is to compute a prior connectivity graph from the sEEG signals themselves. This is where the core of the model's spatial reasoning comes in: the diffusion process it employs effectively models spatial dependencies within this graph by accounting for the influence of a random walk over a K-hop neighborhood.

At its heart, this mechanism relies on the diffusion convolution process, a fundamental operation in many Graph Neural Networks (GNNs). This process is specifically designed to capture and aggregate information by mimicking a diffusion-like spreading of features across a graph's structure. In essence, it generalizes the concept of convolution, which we typically associate with grid-structured data like images, to the more complex and arbitrary connections found in graph data.

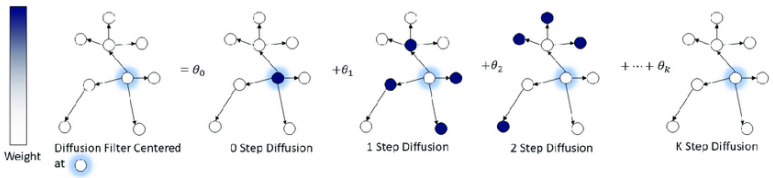


Figure 8: Diffusion Convolution process, as illustrated in [17]

4.1.1 Undirected graphs

A diffusion convolution aims to compute a new representation for each node by combining its own features with those of its neighbors, and its neighbors' neighbors, up to a certain "diffusion depth" or "receptive field" of K hops. This process effectively allows information to "diffuse" outwards from each node.

The mathematical formulation for the diffusion convolution of the m-th feature across the graph $\mathcal{G}$ can be expressed as:

$$\boldsymbol{X}_{:,m} \star_{\mathcal{G}} f_\theta = \boldsymbol{\Phi}\left(\sum_{k=0}^{K-1} \theta_k \boldsymbol{\Lambda}^k\right)\boldsymbol{\Phi}^\intercal \boldsymbol{X}_{:,m} = \sum_{k=0}^{K-1} \theta_k \boldsymbol{L}^k \boldsymbol{X}_{:,m} = \sum_{k=0}^{K-1} \tilde{\theta}_k T_k(\tilde{\boldsymbol{L}})\boldsymbol{X}_{:,m} \quad \text{for } m \in \{1, ..., M\} \tag{1}$$

With:

- $\boldsymbol{X}_{:,m}$: This denotes the m-th feature vector across all nodes in the graph. If $\boldsymbol{X} \in \mathbb{R}^{N \times M}$ is the input feature matrix, where N is the number of nodes and M is the number of input features, then X:,m is the m-th column of $\boldsymbol{X}$. The diffusion convolution operates on each feature dimension independently.
- f_θ: This represents the convolution filter parameterized by θ.
- $\boldsymbol{L}$: This is the **Graph Laplacian matrix**. For an undirected graph, the unnormalized Laplacian is defined as $\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A}$, where $\boldsymbol{D}$ is the degree matrix (a diagonal matrix with node degrees on the diagonal) and $\boldsymbol{A}$ is the adjacency matrix. The Laplacian is a discrete analogue of the Laplace operator and plays a crucial role in spectral graph theory, characterizing the graph's connectivity and diffusion properties.

Different ways of seeing the diffusion convolution operator:

- **Spectral domain** - $\boldsymbol{\Phi}\left(\sum_{k=0}^{K-1} \theta_k \boldsymbol{\Lambda}^k\right)\boldsymbol{\Phi}^\intercal \boldsymbol{X}_{:,m}$: this is the definition of diffusion convolution in the spectral domain, with:
  - $\boldsymbol{\Phi}$ the matrix of eigenvectors of the graph Laplacian $\boldsymbol{L}$. These eigenvectors form an orthonormal basis for the graph, akin to Fourier basis functions for grid-structured data.
  - $\boldsymbol{\Lambda}$ the diagonal matrix of eigenvalues, such that $\boldsymbol{L} = \boldsymbol{\Phi}\boldsymbol{\Lambda}\boldsymbol{\Phi}^\top$. The eigenvalues correspond to frequencies in the graph spectrum.
  - $\boldsymbol{\Phi}^\top \boldsymbol{X}_{:,m}$: This operation projects the input feature vector $\boldsymbol{X}_{:,m}$ onto the eigenbasis of the Laplacian, transforming it into the spectral domain (similar to a Graph Fourier Transform).
  - $\sum_{k=0}^{K-1} \theta_k \boldsymbol{\Lambda}^k$: This term represents the learnable filter applied in the spectral domain. It's a polynomial of the eigenvalues $\boldsymbol{\Lambda}$. Each θ_k is a learnable parameter that dictates how much influence information corresponding to a particular 'frequency' (eigenvalue) contributes to the output.
  - $\boldsymbol{\Phi}\left(\dots\right)$ This projects the filtered spectral representation back to the original node domain.
- **Spatial domain** - $\sum_{k=0}^{K-1} \theta_k \boldsymbol{L}^k \boldsymbol{X}_{:,m}$. This equality directly stems from the spectral decomposition of the Laplacian: since $\boldsymbol{L} = \boldsymbol{\Phi}\boldsymbol{\Lambda}\boldsymbol{\Phi}^\top$.
  - $\boldsymbol{L}^k \boldsymbol{X}_{:,m}$: This term represents applying the Laplacian operator k times to the feature vector. Intuitively, applying $\boldsymbol{L}$ once aggregates information from immediate neighbors (1-hop). Applying $\boldsymbol{L}^2$ aggregates infor-

mation from 2-hop neighbors, and so on. Higher powers of L correspond to aggregating information from nodes further away in the graph.

- $\sum_{k=0}^{K-1} \theta_k (\dots)$: This is a linear combination of these multi-hop aggregated features. The learnable parameters θ_k act as weights, allowing the model to learn the importance of information from different "distances" (hops) away from a node. This directly corresponds to a diffusion process, where information spreads out in discrete steps.

- **Chebyshev Polynomial Approximation** - $\sum_{k=0}^{K-1} \tilde{\theta}_k T_k(\tilde{\boldsymbol{L}}) \boldsymbol{X}_{:,m}$. While the previous form is intuitive, directly computing powers of $\boldsymbol{L}$ can be computationally expensive for large graphs, especially for high K. Furthermore, the eigenvalues of $\boldsymbol{L}$ can span a wide range, which might lead to numerical instability or issues with filter localization. To address this, the spectral filter (the polynomial of eigenvalues) is often approximated using a truncated expansion of Chebyshev polynomials of the first kind, denoted by $T_k(x)$.

  - $\tilde{\boldsymbol{L}} = \boldsymbol{I} - \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}$: This is the symmetric normalized Laplacian. It's used because its eigenvalues are guaranteed to lie within the range [0,2] (or scaled to [1,1] for Chebyshev polynomials), which is ideal for the stability and properties of Chebyshev approximations. $\boldsymbol{D}^{-1/2}$ is the inverse square root of the degree matrix.
  - $T_k(\tilde{\boldsymbol{L}})$: These are the Chebyshev polynomials evaluated at the normalized Laplacian. They are defined recursively: $T_0(x) = 1$, $T_1(x) = x$, and $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$. This recursive property means that $T_k(\tilde{L})$ can be computed efficiently without explicit matrix exponentiation or eigenvalue decomposition.
  - $\tilde{\theta_k}$: These are the new learnable coefficients (parameters) in the Chebyshev basis. These coefficients effectively define the shape of the convolution filter in the spectral domain. This approximation makes graph convolutional networks computationally tractable and localized, as the filter is a K-localized polynomial that only considers information up to K hops away.

In summary, the diffusion convolution process uses the mathematical properties of the graph Laplacian to provide an efficient method for aggregating information across graph neighborhoods.

4.1.2 Directed graphs

$$\boldsymbol{X}_{:,m \star \mathcal{G}} f_\theta = \sum_{k=0}^{K-1} \left(\theta_{k,1}(\boldsymbol{D}_O^{-1}\boldsymbol{W})^k + \theta_{k,2}(\boldsymbol{D}_I^{-1}\boldsymbol{W}^\intercal)^k\right)\boldsymbol{X}_{:,m} \text{ for } m \in \{1, ..., M\} \quad (2)$$

The diffusion equation for directed graphs, as utilized in models like the Diffusion Convolutional Gated Recurrent Unit (DCGRU), extends standard graph convolution to account for the inherent directionality of edges. It involves two distinct state transition matrices: $\boldsymbol{P}_{\text{out}} = \boldsymbol{D}_O^{-1}\boldsymbol{W}$ for outward diffusion and $\boldsymbol{P}_{\text{in}} = \boldsymbol{D}_I^{-1}\boldsymbol{W}^\intercal$ for inward diffusion. Here, $\boldsymbol{W}$ represents the edge-weighted adjacency matrix, $\boldsymbol{D}_O$ is the diagonal out-degree matrix, and $\boldsymbol{D}_I$ is the diagonal in-degree matrix. Each term captures the weighted sum of features propagating either from a node to its out-neighbors (outward) or from its in-neighbors to the node itself (inward), normalized by the respective degrees. The overall diffusion process combines these directional propagations over multiple steps, effectively modeling information flow or influence spreading across the directed graph structure.

4.2 Connectivity Graphs

The effectiveness of the diffusion process is highly dependent on the underlying graph structure. We explored two distinct approaches for constructing connectivity graphs among EEG electrodes: one based on physical proximity and another on functional connectivity.

4.2.1 Distance-Based Graph

This approach leverages the physical Euclidean distance between electrode placements. Similarity s_{ij} between electrodes i and j is calculated using a Gaussian kernel:

$$s_{ij} = e^{-\frac{d_{ij}}{\sigma^2}}$$

where d_{ij} is the Euclidean distance and σ is a scaling parameter. A higher similarity indicates closer proximity. A critical limitation for this dataset, however, was the incomplete availability of real-life electrode coordinates, which are typically inferred from MRI scans. Consequently, this method could not be uniformly applied to all patients. The resulting graph is then thresholded, with all edges falling below a similarity value of $\kappa \in [0, 1]$ being removed to ensure sparsity.

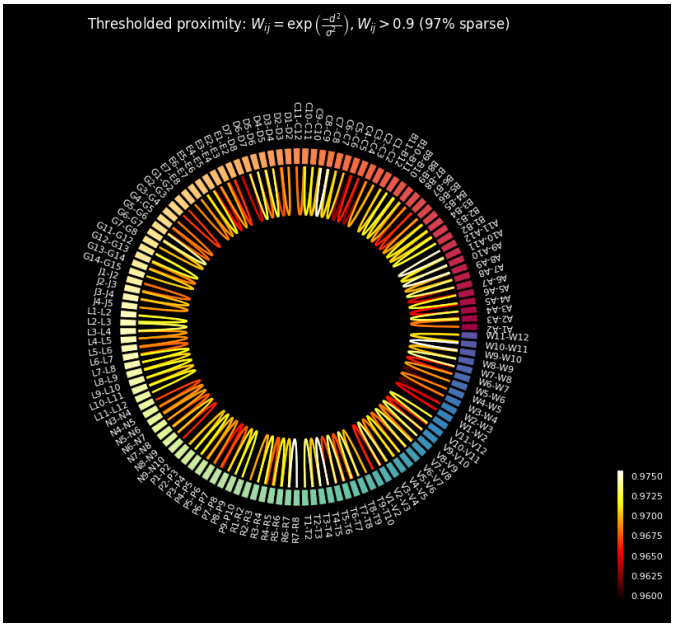


Figure 9: Example of a distance-based connectivity graph.

4.2.2 Connectivity-Based Graph (Phase Locking Value)

To overcome the limitations of physical distance, we also utilized functional connectivity measures, specifically the **Phase Locking Value (PLV)**. PLV quantifies the phase synchronicity between two signals and is well-suited for capturing dynamic interactions between brain regions.
The PLV between two channels in a given time window is defined as:

$$\left| \mathbb{E}\left\{ e^{i(\phi_1(t) - \phi_2(t))} \right\} \right|$$

where $\phi_1(t)$ and $\phi_2(t)$ are the instantaneous phases of the two signals. These instantaneous phases are derived via the **Hilbert transform**. For a real signal $x(t)$, its analytic signal $z(t)$ is given by:

$$z(t) = x(t) + i\mathcal{H}\{x(t)\}$$

Here, $\mathcal{H}\{x(t)\}$ represents the Hilbert transform of $x(t)$. Conceptually, the analytic signal corresponds to the inverse Fourier transform of $2U(f)X(f)$, where $X(f) = \mathcal{F}\{x(t)\}$ (the Fourier transform of $x(t)$) and $U(f)$ is the Heaviside step function, which removes negative frequencies. The factor of two preserves signal energy. The presence of a non-zero imaginary component in the inverse Fourier transform

allows for the definition of the instantaneous amplitude $A(t)$ and phase $\phi(t)$:

$$A(t) = \sqrt{x^2(t) + \mathcal{H}\{x(t)\}^2}$$

$$\phi(t) = \text{atan2}(\mathcal{H}\{x(t)\}, x(t))$$

We used the average PLV computed over a defined time period as the measure of connectivity between two nodes. A key advantage of PLV is its focus solely on phase relationships, making it robust to variations in instantaneous amplitude that might arise from electrode placement or signal gain differences. Connectivity graphs derived from PLV can either be directly thresholded or made sparse using methods like the graphical lasso.

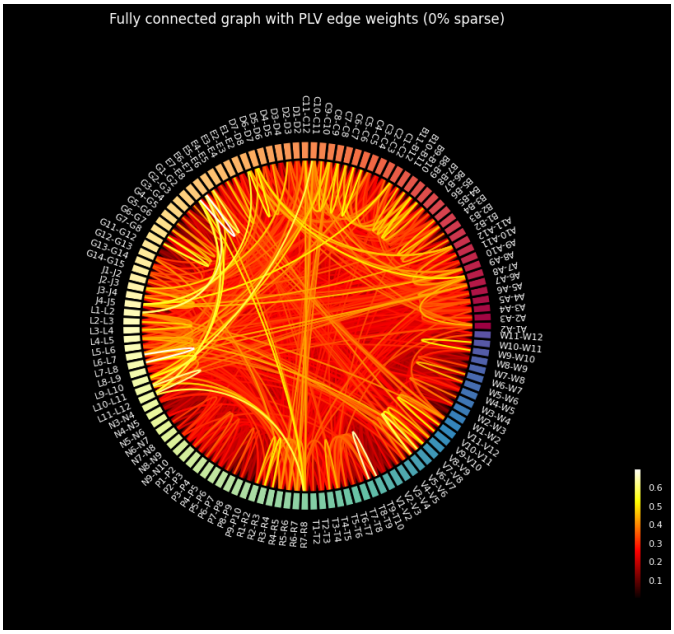


Figure 10: Unthresholded Phase Locking Value (PLV) connectivity for a single patient within a defined time period

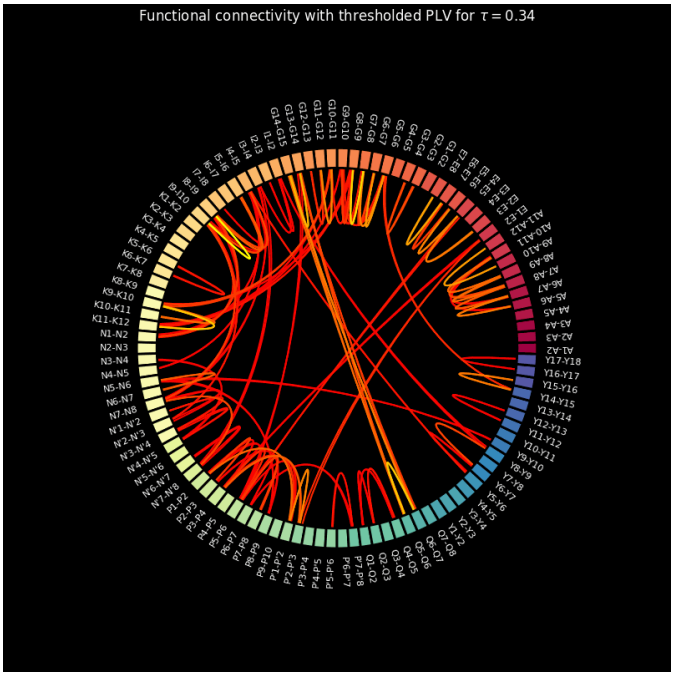


Figure 11: Thresholded PLV connectivity

4.3 DCGRU cell

The Diffusion Convolutional Gated Recurrent Unit (DCGRU) cell extends the standard Gated Recurrent Unit (GRU) by incorporating the previously presented diffusion convolutions within its core operations. This modification is crucial for effectively capturing spatial dependencies inherent in graph-structured data, such as EEG signals. Specifically, diffusion convolutions, represented by $\boldsymbol{\Theta}_{\cdot\star\mathcal{G}}$, are applied to the inputs of both the **reset gate** ($\boldsymbol{r}^{(t)}$) and the **update gate** ($\boldsymbol{u}^{(t)}$). These gates, defined as:

$$\boldsymbol{r}^{(t)} = \sigma\big(\boldsymbol{\Theta}_{r\star\mathcal{G}}[\boldsymbol{X}^{(t)}, \boldsymbol{H}^{(t-1)}] + \mathbf{b}_r\big) \qquad \boldsymbol{u}^{(t)} = \sigma\big(\boldsymbol{\Theta}_{u\star\mathcal{G}}[\boldsymbol{X}^{(t)}, \boldsymbol{H}^{(t-1)}] + \mathbf{b}_u\big)$$

regulate the flow of information from the previous hidden state ($\boldsymbol{H}^{(t-1)}$) and the current input ($\boldsymbol{X}^{(t)}$). The diffusion convolution is also applied during the calculation of the **candidate hidden state** ($\boldsymbol{C}^{(t)}$), which determines potential new information to be incorporated. The candidate is computed as:

$$\boldsymbol{C}^{(t)} = \tanh\big(\boldsymbol{\Theta}_{C\star\mathcal{G}}[\boldsymbol{X}^{(t)}, (\boldsymbol{r}^{(t)} \odot \boldsymbol{H}^{(t-1)})] + \mathbf{b}_C\big)$$

Here, $[\boldsymbol{X}^{(t)}, \boldsymbol{H}^{(t-1)}]$ denotes the concatenation of the input vector and the previous hidden state, while $\odot$ signifies the Hadamard (element-wise) product. Finally, the

new **hidden state** ($\boldsymbol{H}^{(t)}$) is generated by selectively combining the previous hidden state and the candidate hidden state using the update gate:

$$\boldsymbol{H}^{(t)} = \boldsymbol{u}^{(t)} \odot \boldsymbol{H}^{(t-1)} + (1 - \boldsymbol{u}^{(t)}) \odot \boldsymbol{C}^{(t)}$$

This architecture allows the DCGRU to effectively model spatio-temporal dependencies by enabling the gates and candidate state to leverage neighborhood information propagated through the diffusion convolutions.

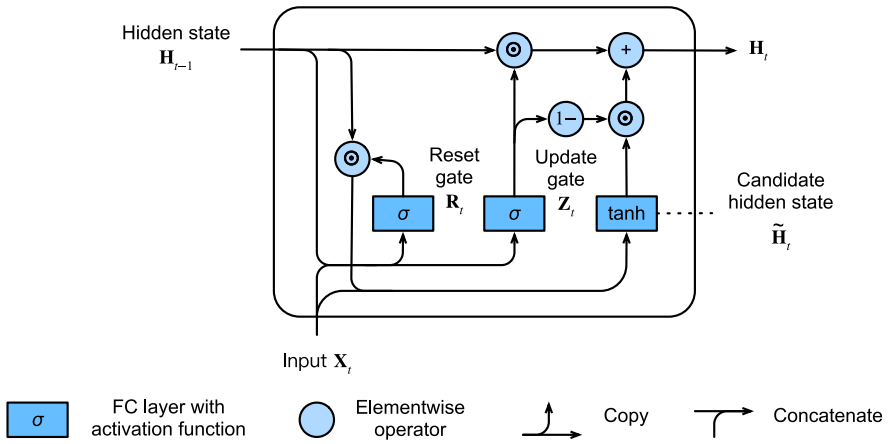


Figure 12: Classical GRU cell for reference, here adapted with diffusion convolutions for spatio-temporal modeling

4.4 Encoder-Decoder Architecture for Spatio-Temporal Prediction

We first employed an encoder-decoder framework (Figure 7) for a self-supervised, next-frame prediction task. This architecture, built from stacked DCGRU cells, was designed to learn the underlying spatio-temporal dynamics of the sEEG signals.
The **encoder** processes a sequence of input frames (e.g., DFTs of short clips, $x_1, \ldots, x_T$) and compresses this information into a set of hidden state representations. The **decoder**'s task is to then generate a sequence of future frames ($x'_{T+1}, \ldots, x'_{2T}$) based on the information provided by the encoder.
In a standard implementation, the final hidden states of the encoder are used as an initial context vector for the decoder. At each decoding step, the attention mechanism allows the decoder to dynamically weigh and access all of the encoder's hidden states, creating a context vector tailored to each output frame.
During training, we utilized **teacher forcing**, where the ground-truth frame from the previous timestep was fed as input to the decoder to ensure stable and efficient learning. The model is then run in a fully autoregressive manner during

inference. Despite successfully training the prediction model, the resulting representations did not provide a clear, discriminative signal for anomaly detection corresponding to the SOZ. This valuable finding suggested that a direct, supervised approach would be more fruitful, leading us to the Transformer-based architecture presented next.

5 Transformer-based SOZ Detection with Contrastive Pre-training

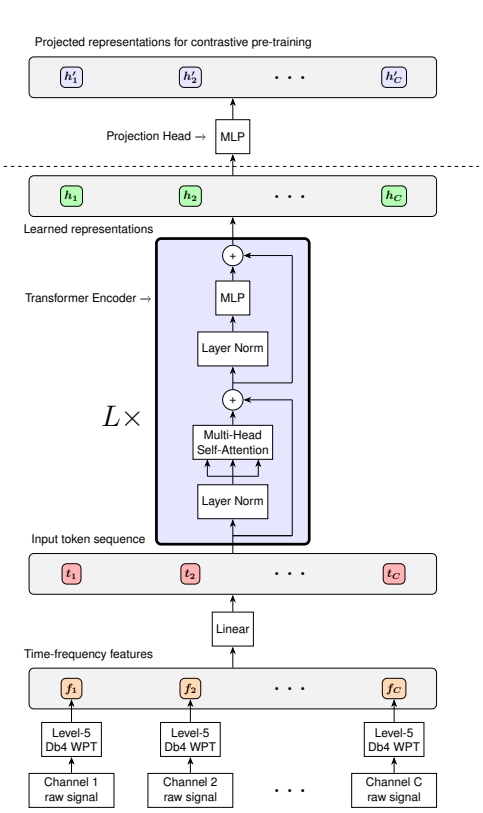
5.1 Model Architecture: A Transformer for Channel-wise SOZ-Detection

To overcome the limitations of the graph-based model, we designed a supervised deep learning pipeline centered on a **Transformer encoder**. This architecture is fundamentally designed to process sEEG channels as an **unordered set**, which makes it inherently flexible and capable of generalizing across patients with different numbers and layouts of implanted electrodes. Instead of predicting future signals, this model directly classifies each channel within a given time window (35 seconds) as either SOZ or non-SOZ.

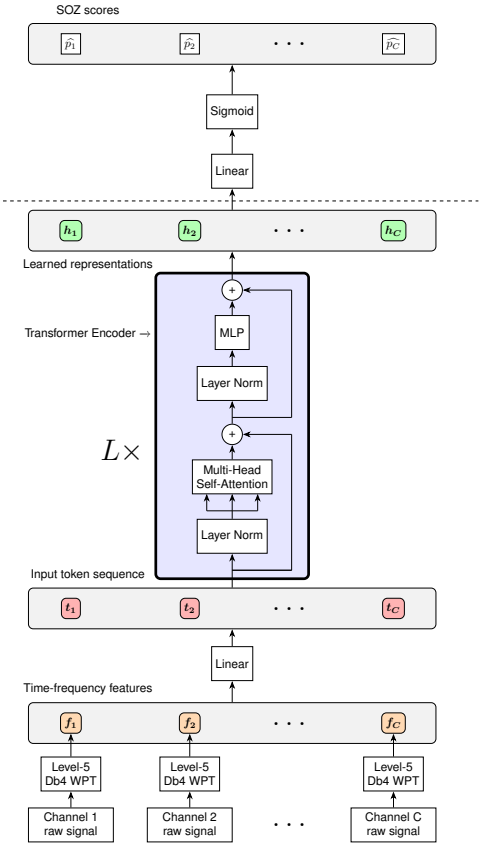
Input Tokenization The pipeline begins by creating a meaningful, fixed-size representation, or "token," for each sEEG channel from its signal. For each channel's 35-second window, we first compute its Daubechies-4 Wavelet Packet Transform (WPT), as detailed in Section 3.4. This high-dimensional WPT output is then flattened and passed through a linear projection layer to produce a **2048-dimensional vector**, which serves as the input token for the Transformer. The input to our model is thus a sequence of these channel tokens. We deliberately omit traditional positional encodings, as there is no single, natural ordering of electrode channels in a typical sEEG implantation. By forgoing a fixed positional prior, we force the model's spatial self-attention mechanism to learn all inter-channel relationships directly from the data, guided purely by the classification task.

Spatial Self-Attention as Learned Connectivity The sequence of channel tokens is fed into a multi-layer Transformer encoder, where the core operation is a **multi-head self-attention mechanism** applied across the channel dimension. This "spatial attention" computes a context-aware representation for each channel by aggregating information from all other channels. For each attention head, the mechanism learns a **dynamic, directed connectivity graph** for every input sample, where attention scores represent the strength of influence between channels. This approach is a fundamental advantage over the DCGRU model, which relied on a static, pre-computed connectivity graph. The Transformer, in contrast, learns a functional connectivity that is both data-driven and optimized specifically for SOZ detection, considering the complete spatial context rather than a fixed K-hop neighborhood.

Two-Stage Training Framework The output of the Transformer encoder is a set of refined embeddings (h_i), one for each channel, that are rich in contextual information. These embeddings form the basis for our two-stage training regimen, as depicted in Figure 13. The process involves an initial **contrastive pre-training** stage—using a dedicated projection head and a focal class-balanced supervised contrastive loss—followed by a **classification fine-tuning** stage that uses a final classification head and a focal class-balanced BCE loss. The specifics of these two stages are elaborated in the subsequent sections.



Encoder with contrastive pre-training head.



Encoder with fine-tuning head.

Figure 13: Architecture of the proposed two-stage training and fine-tuning model. Both stages leverage a shared Transformer Encoder backbone to process channel-wise information. **(a) Stage 1: Contrastive Pre-training.** The model learns to create discriminative representations. Raw sEEG is processed by a WPT and a linear layer to generate input tokens for an L-layer Transformer Encoder. The resulting embeddings (h_i) are passed through an MLP projection head to get final representations (h'_i) used for the contrastive loss calculation. **(b) Stage 2: Classification Fine-tuning.** The model is adapted for SOZ detection. The pre-training head is removed, and the frozen or fine-tuned encoder backbone produces channel embeddings (h_i). These are fed into a classification head (a linear layer and sigmoid activation) to output the final SOZ probability scores ($\hat{p}_i$). This two-stage approach allows the model to first learn robust signal features before specializing on the classification task.

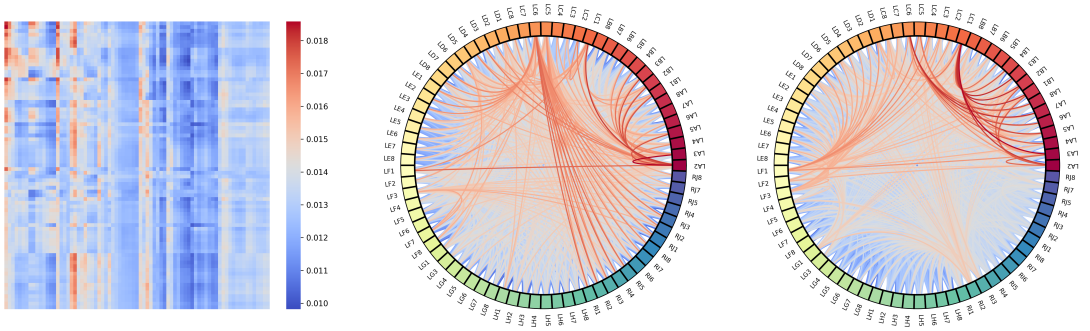


Figure 14: Attention map seen as a directed graph. The center graph shows the upper right triangle of the attention matrix, while the rightmost graph shows the lower triangle. Attention flows from queries (line indices) to keys (column indices). Arrows not shown for clarity.

5.2 Contrastive Pre-training with a Focal Class-Balanced Loss

The first stage of our training pipeline is a pre-training phase designed to learn highly discriminative channel embeddings. The goal is to structure the latent space such that representations of SOZ and non-SOZ channels form distinct, compact clusters. This task faces two significant challenges inherent in sEEG data: **severe class imbalance**, as non-SOZ channels vastly outnumber SOZ channels, and the presence of both **easy and hard examples** within each class. To address these challenges simultaneously, we employ a sophisticated loss function that integrates ideas from Supervised Contrastive learning [18], class-balanced loss [19], and focal loss [20].

The foundation of our approach is the Supervised Contrastive (SupCon) loss, which extends the self-supervised contrastive framework to a fully supervised setting. For a given "anchor" channel embedding, the loss aims to pull embeddings of the same class (the "positives") closer together, while pushing away embeddings from all other classes (the "negatives").

To handle the class imbalance, we incorporate a **class-balancing** weight based on the concept of "effective number of samples." This weighting scheme re-balances the loss by assigning a higher weight to minority classes (i.e., the SOZ channels). The weight for a given anchor i belonging to class y_i is defined as:

$$w_i^{\text{CB}} = \frac{1-\beta}{1-\beta^{N_{y_i}}}$$

where N_{y_i} is the number of training samples in class y_i and $\beta \in [0, 1)$ is a hyper-

parameter that controls the degree of re-weighting. As $\beta \to 1$, this term provides a significant boost to the loss contribution from rare classes.

To focus the training on more difficult examples, we introduce a **focal modulation** term. The intuition is that easy-to-classify positive pairs (those already close in the embedding space) should contribute less to the overall loss, allowing the model to concentrate on harder pairs. This is achieved by modulating the standard cross-entropy loss with a factor of $(1 - p_{ip})^\gamma$, where p_{ip} is the contrastive probability of correctly matching anchor i with its positive partner p. The probability p_{ip} is calculated as:

$$p_{ip} = \frac{\exp(\text{sim}(\boldsymbol{z}_i, \boldsymbol{z}_p)/\tau)}{\sum_{k \in A(i)} \exp(\text{sim}(\boldsymbol{z}_i, \boldsymbol{z}_k)/\tau)}$$

Here, $\boldsymbol{z}$ are the channel embeddings from the projection head, $A(i)$ is the set of all other embeddings in the batch (both positive and negative), and τ is the temperature hyperparameter. When a pair is easy ($p_{ip} \to 1$), the focal term approaches zero, down-weighting its contribution.

By combining these elements, we arrive at the final Focal Class-Balanced Supervised Contrastive Loss, which we aggregate over all anchor channels i in a batch:

$$\mathcal{L}_{\text{F-CB-SC}} = \sum_{i \in I} w_i^{\text{CB}} \frac{-1}{|P(i)|} \sum_{p \in P(i)} (1 - p_{ip})^\gamma \log(p_{ip}) \tag{3}$$

where $P(i)$ is the set of positive examples for anchor i in the batch. This composite loss function effectively guides the pre-training process to learn an embedding space that is robust to class imbalance and focused on the most informative examples, creating an ideal foundation for the downstream classification task.

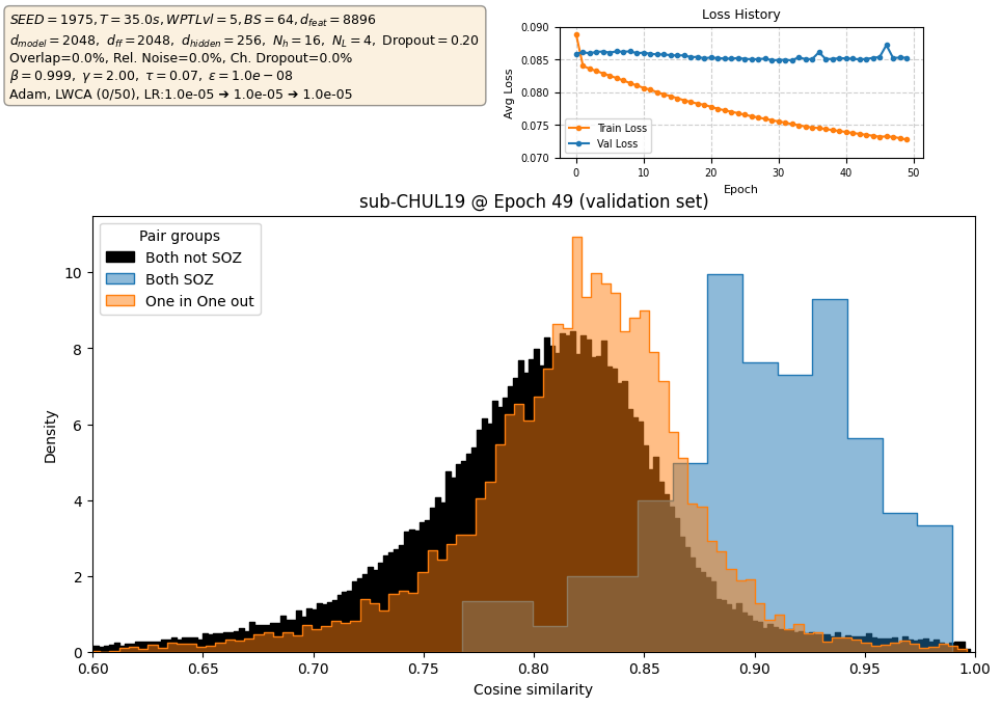


Figure 15: Pairwise cosine similarity distributions for validation patient `CHUL19` after pre-training. The clear separation between same-class pairs (high similarity) and different-class pairs (low similarity) demonstrates that the supervised contrastive loss has successfully structured the embedding space.

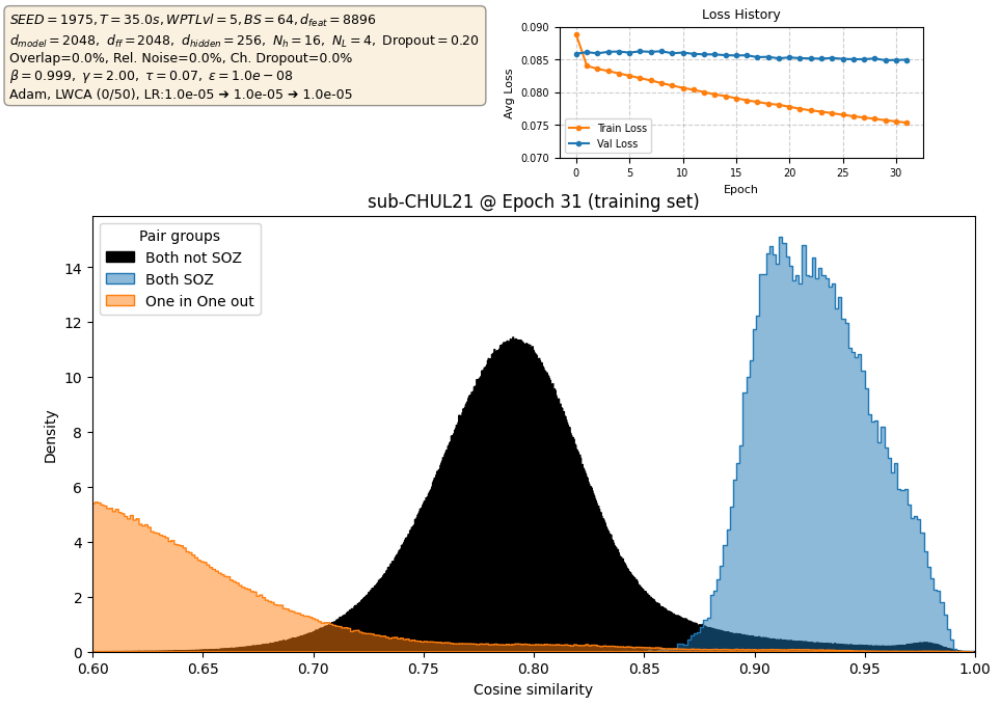


Figure 16: Similarity distributions for training patient `CHUL21`, showing highly effective clustering. The separation is so pronounced that the distribution for different-class (SOZ-non-SOZ) pairs is pushed towards zero similarity, partially outside the plot's visible frame.

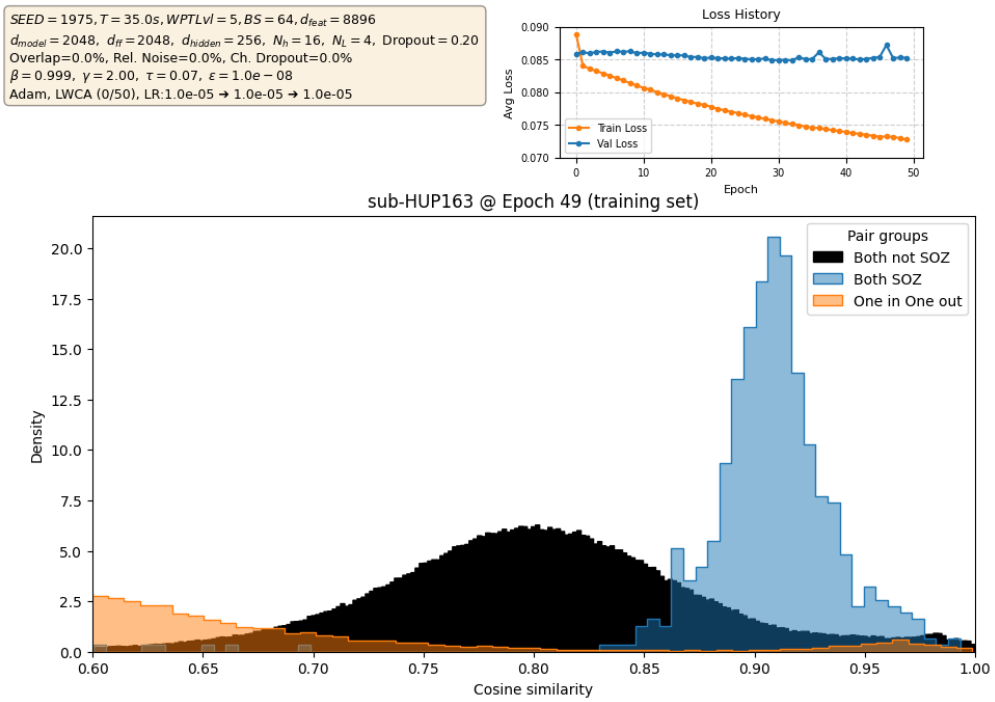


Figure 17: An example of strong class separation on the training set for patient `HUP163`. The distributions for same-class pairs are tightly clustered at high similarity, while the distribution for different-class pairs is distinctly separated at low similarity, confirming the success of the pre-training objective.

5.3 Fine-tuning for SOZ Channel Classification

After the contrastive pre-training stage, the model has learned a powerful and well-structured representation of sEEG channels. The second and final stage of our pipeline is to **fine-tune** this pre-trained model for the specific downstream task of Seizure Onset Zone (SOZ) classification. For this, the contrastive projection head is removed and replaced with a new **classification head**, which consists of a linear layer that maps the learned channel embeddings (h_i) to a single logit, followed by a sigmoid activation function to produce a SOZ probability score between 0 and 1 for each channel.

The critical challenge of **class imbalance** persists during this phase. To prevent the model's predictions from being biased towards the vastly more numerous non-SOZ channels, we adapt the principles from the pre-training loss to the classification setting. We employ a **Focal Class-Balanced Binary Cross-Entropy (BCE) Loss**, which modifies the standard BCE loss to prioritize both the minority class

and hard-to-classify examples.

The standard BCE loss for a single channel prediction is $-\log(p_t)$, where p_t is the predicted probability for the ground truth class. Our loss function enhances this in two ways. First, we apply the same **class-balancing weight** (w_n^{CB}) used during pre-training, which is calculated based on the effective number of samples with hyperparameter β. This weight significantly increases the loss contribution from the minority SOZ class, forcing the model to learn its features effectively.

Second, we incorporate the **focal modulation** factor $(1 - p_t^n)^\gamma$, where γ is the focusing parameter. This term reduces the loss for well-classified examples (where the predicted probability for the true class, p_t^n, is high), thereby focusing the training on "hard" examples where the model is less confident.

Combining these elements, the complete Focal Class-Balanced BCE loss, aggregated over all N channels in a batch, is defined as:

$$\mathcal{L}_{\text{F-CB-BCE}} = -\frac{1}{N} \sum_{n=1}^{N} w_n^{\text{CB}} (1 - p_t^n)^\gamma \log(p_t^n) \tag{4}$$

where p_t^n is defined as p_n if the true label $y_n = 1$, and $1 - p_n$ if $y_n = 0$, with p_n being the sigmoid output for channel n. By starting with the rich, context-aware features from the pre-trained encoder and fine-tuning with this robust, imbalance-aware loss function, the model learns to accurately and reliably identify SOZ channels.

Staged Fine-tuning Strategy The fine-tuning process itself was conducted incrementally to preserve the rich features learned during pre-training. Our initial strategy involved freezing the entire Transformer encoder backbone and training only the parameters of the newly added linear classification head. This approach, however, failed to produce a meaningful classifier, yielding an Area Under the Receiver Operating Characteristic (AUROC) of approximately 0.5, equivalent to random guessing. This result strongly suggests that the representations learned via contrastive pre-training, while well-clustered, were not **linearly separable**. A single linear layer possessed insufficient expressive power to find a decision boundary in the complex embedding space. In retrospect, a more powerful classification head, such as a Multi-Layer Perceptron (MLP), might have been more effective at this stage by introducing non-linearity. To achieve successful classification, it was therefore necessary to adopt a more comprehensive approach: we proceeded to gradually unfreeze the layers of the Transformer encoder, starting from the final

layer and moving downwards, fine-tuning them with a small learning rate. This allowed the model to subtly adapt its core representations to the specific demands of the SOZ classification task, ultimately leading to a high-performing classifier.

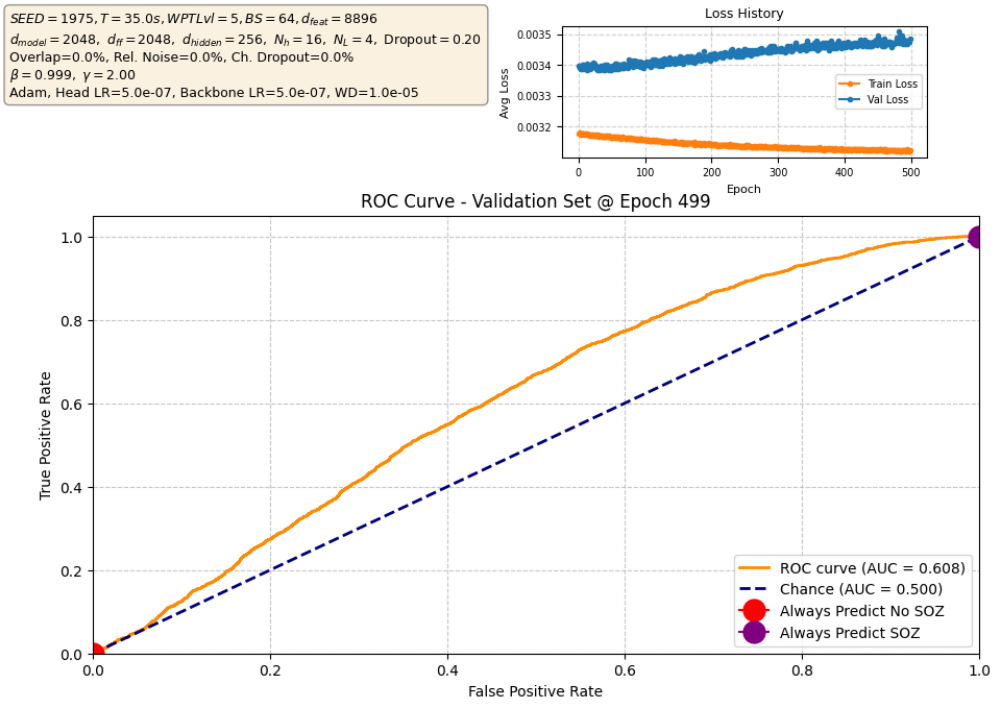


Figure 18: Global Receiver Operating Characteristic (ROC) curve aggregated over all patients in the validation set. The model achieves an AUROC of 0.608 using 35-second signal clips.

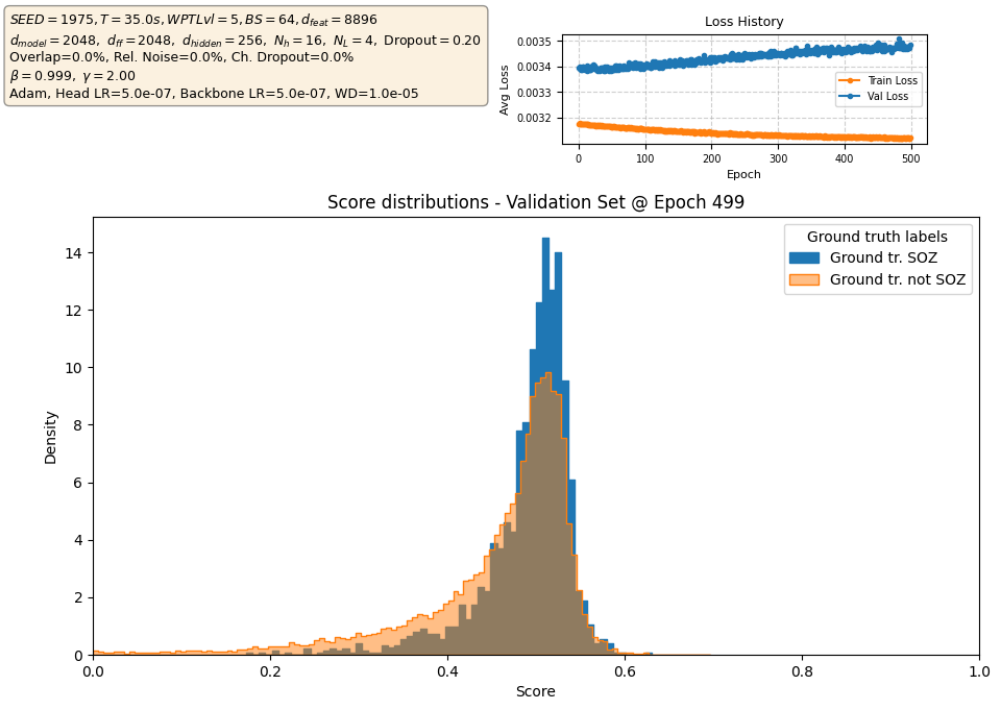


Figure 19: Global distribution of predicted SOZ scores, aggregated from all validation patients. The slight separation, with SOZ channels (blue) shifted towards higher probabilities than non-SOZ channels (orange), visually corresponds to the global AUROC of 0.608.

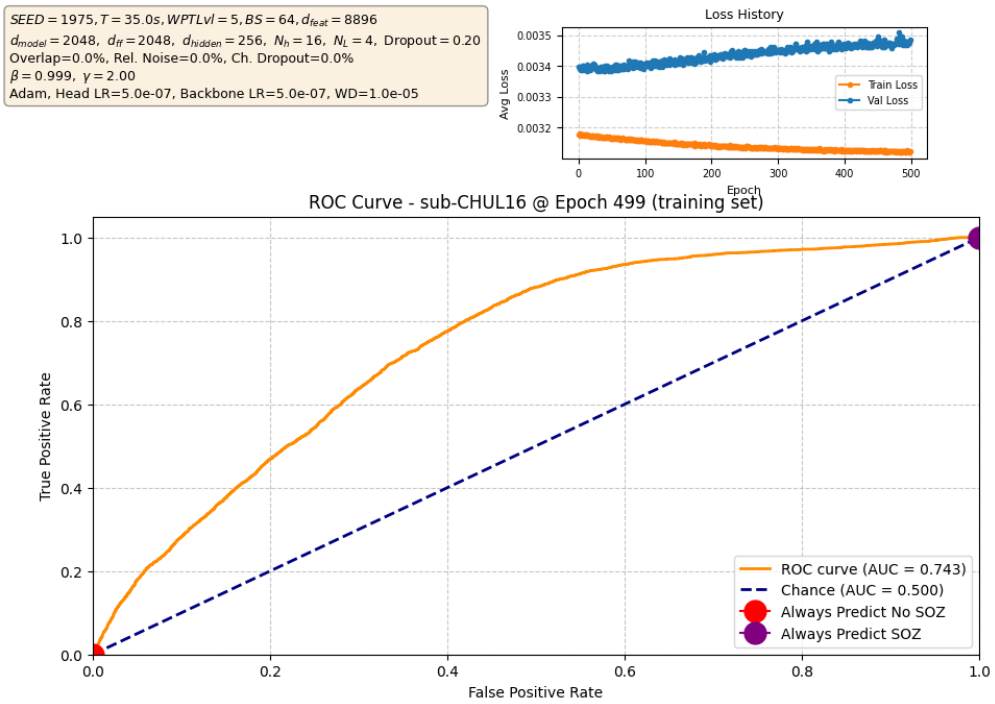


Figure 20: Example of strong classification performance on the training set for patient `CHUL16`, achieving an AUROC of 0.74.

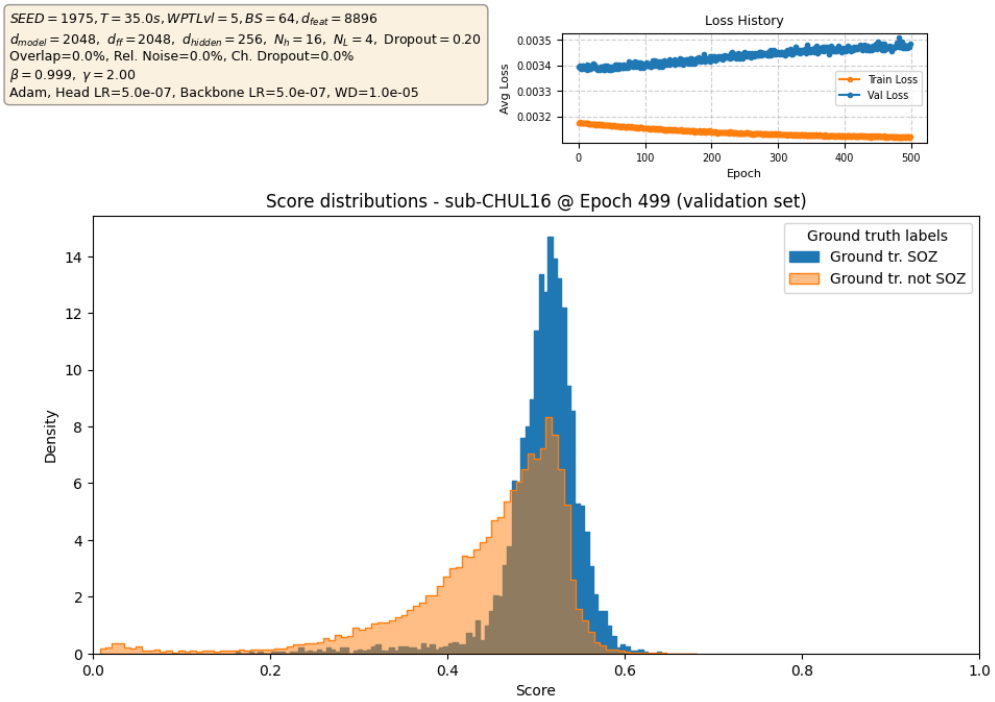


Figure 21: The corresponding score distribution for patient `CHUL16`. The high AUROC is reflected in a clear separation between the SOZ and non-SOZ score distributions.

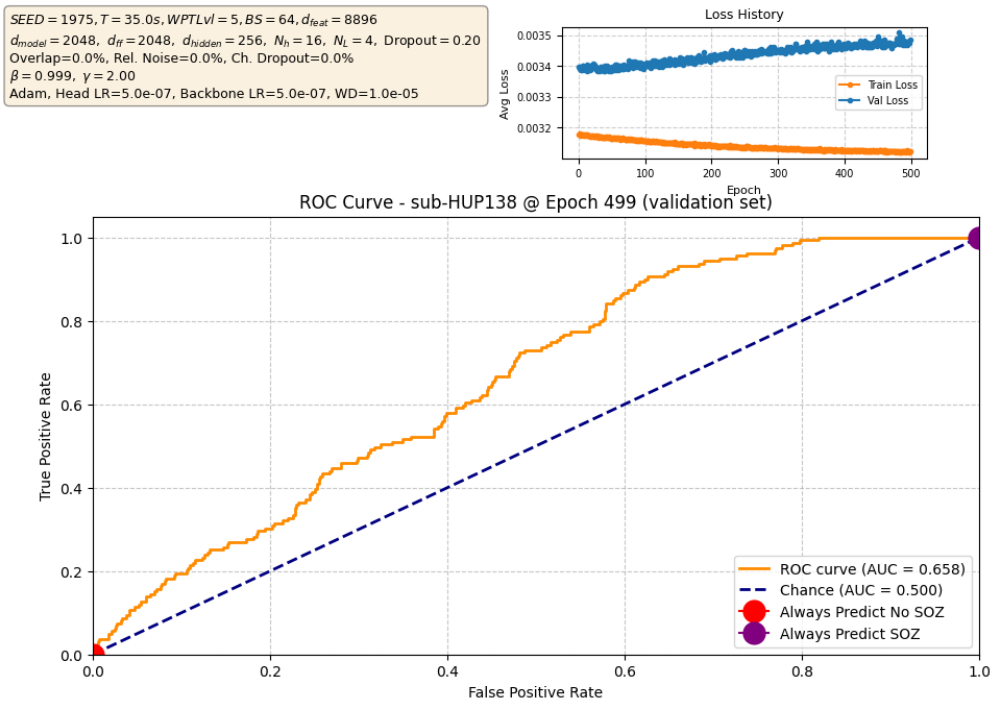


Figure 22: ROC curve for patient `HUP138` from the validation set, demonstrating above-average performance for a single patient with an AUROC of 0.66.

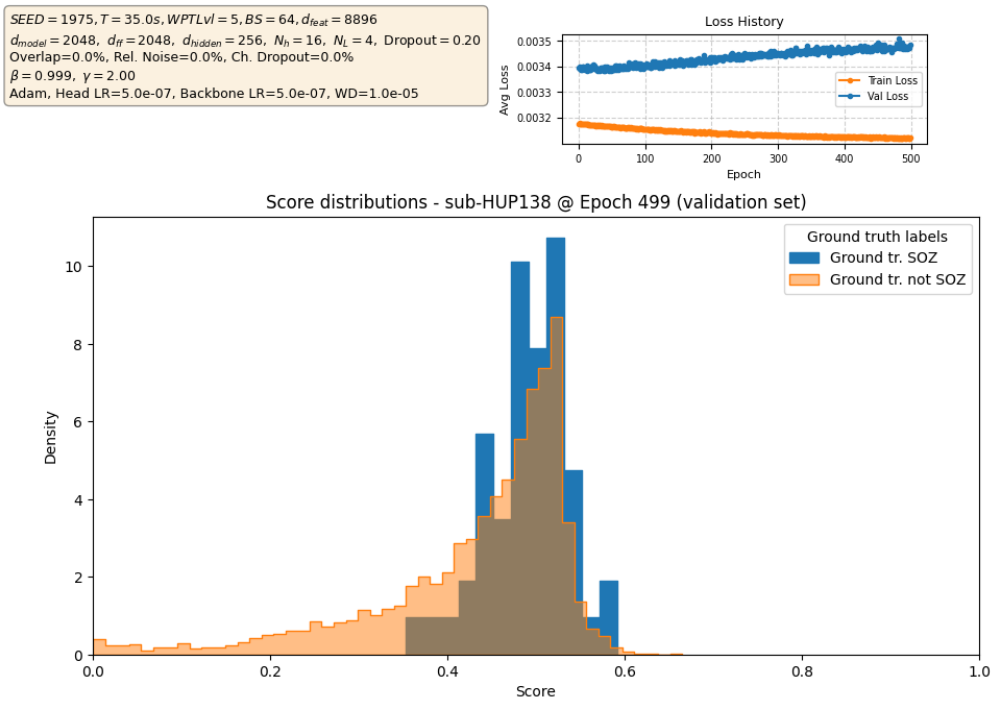


Figure 23: The corresponding score distribution for patient `HUP138`. The higher AUROC is visually confirmed by a more pronounced separation between the SOZ (blue) and non-SOZ (orange) score distributions.

5.4 Preliminary Results and Discussion

Upon completion of the two-stage training process, the model was evaluated on the held-out validation set, achieving an Area Under the Receiver Operating Characteristic (AUROC) of 0.60. While this result is modest and indicates that the classifier is not yet performing at a state-of-the-art level, its significance lies in the challenging conditions under which it was obtained. This performance was achieved using input clips of only **35 seconds** of sEEG signal. The ability of the model to extract any meaningful, discriminative signal for SOZ localization from such a brief temporal window is in itself a remarkable outcome, as it suggests the spatial attention mechanism is effectively identifying complex, latent spatio-temporal biomarkers.

This result should therefore be interpreted as a strong **proof-of-concept** for our proposed methodology. The architecture, pre-training strategy, and loss functions have successfully created a system capable of learning from this challenging data. We anticipate that performance can be substantially improved through two

primary avenues. First, as more training data becomes available through our ongoing collaboration, the model will be exposed to a wider variety of patient cases and seizure patterns, which is critical for generalization and improved accuracy. Second, by implementing the hierarchical architecture discussed as future work—first aggregating features with a temporal attention mechanism before the spatial attention stage—the model will be better equipped to capture the evolving temporal dynamics of seizure precursors. The combination of a larger dataset and a more powerful, temporally-aware architecture provides a clear path toward developing a high-performance model for SOZ detection that could have significant clinical impact.

6 Conclusion and Future Work

This research has successfully demonstrated the potential of a Transformer-based architecture, enhanced by a spatial contrastive pre-training strategy and focal class-balanced loss functions, for the challenging task of Seizure Onset Zone detection from sEEG data. Our approach effectively addresses class imbalance and learns discriminative channel-specific representations, moving beyond fixed graph structures by allowing the model to learn dynamic inter-channel relationships through spatial attention. The promising preliminary results, including the presentation at the GSP Workshop, underscore the viability of this methodology.

Looking ahead, a key avenue for extending this work is to overcome the limitations of the 35-second analysis window, which may not capture longer-term electrophysiological patterns that are crucial for seizure prediction. A **hierarchical Transformer architecture** presents a compelling next step specifically designed to solve this problem. This approach would enable the analysis of much longer recordings (e.g., five minutes or more) in a computationally efficient manner.

The process would involve two stages. First, a long signal would be divided into a sequence of shorter, non-overlapping sub-windows (e.g., 5 or 10-second clips). A **temporal attention mechanism** would then operate along the time axis for each channel independently, learning to aggregate features and create a single, context-aware summary representation for each sub-window. The result would be a much shorter sequence of summary vectors for each channel. These condensed sequences, which now represent long-term dynamics, would then be fed into the **spatial Transformer encoder**, as described in this report. The model could then perform its spatial attention on these temporally-aware summaries to model complex, long-range functional connectivity. Such an architecture would be far more powerful, allowing the model to learn from the slow build-up of seizure activity and significantly improving its potential for clinical application.

Further research could also explore the integration of other modalities, the refinement of wavelet packet parameters, or advanced techniques for interpreting the learned spatial attention maps to provide clinicians with greater insight into the model's decision-making process. With these promising avenues for development, we aim to significantly advance this research and prepare a comprehensive manuscript for submission to the International Conference on Learning Representations (ICLR) in the fall of 2025, for the 2026 conference.

References

[1] L. Qi, X. Fan, X. Tao, *et al.*, "Identifying the Epileptogenic Zone With the Relative Strength of High-Frequency Oscillation: A Stereoelectroencephalography Study," *Frontiers in Human Neuroscience*, vol. 14, Jun. 9, 2020, ISSN: 1662-5161. DOI: 10.3389/fnhum.2020.00186. [Online]. Available: https://www.frontiersin.org/journals/human-neuroscience/articles/10.3389/fnhum.2020.00186/full.

[2] H. E. Goldstein, B. E. Youngerman, B. Shao, *et al.*, "Safety and efficacy of stereoelectroencephalography in pediatric focal epilepsy: A single-center experience," *Journal of Neurosurgery: Pediatrics*, vol. 22, no. 4, pp. 444–452, Jul. 20, 2018, ISSN: 1933-0715, 1933-0707. DOI: 10.3171/2018.5.PEDS1856. [Online]. Available: https://thejns.org/pediatrics/view/journals/j-neurosurg-pediatr/22/4/article-p444.xml.

[3] P. Chauvel, *The History and Principles of Stereo EEG*. Springer Publishing Company, Aug. 20, 2023, ISBN: 978-0-8261-3692-3. [Online]. Available: https://connect.springerpub.com/content/book/978-0-8261-3693-0/part/part01/chapter/ch01.

[4] F. Tadel, S. Baillet, J. C. Mosher, D. Pantazis, and R. M. Leahy, "Brainstorm: A User-Friendly Application for MEG/EEG Analysis," *Computational Intelligence and Neuroscience*, vol. 2011, no. 1, p. 879 716, 2011, ISSN: 1687-5273. DOI: 10.1155/2011/879716. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1155/2011/879716.

[5] C. Holdgraf, S. Appelhoff, S. Bickel, *et al.*, "iEEG-BIDS, extending the Brain Imaging Data Structure specification to human intracranial electrophysiology," *Scientific Data*, vol. 6, no. 1, p. 102, Jun. 25, 2019, ISSN: 2052-4463. DOI: 10.1038/s41597-019-0105-7. [Online]. Available: https://www.nature.com/articles/s41597-019-0105-7.

[6] J. M. Bernabei, A. Li, A. Y. Revell, *et al.*, *HUP iEEG Epilepsy Dataset*, Openneuro, 2023. DOI: 10.18112/OPENNEURO.DS004100.V1.1.3. [Online]. Available: https://openneuro.org/datasets/ds004100/versions/1.1.3.

[7] I. Daubechies, *Ten Lectures on Wavelets* (Regional Conference Series in Applied Mathematics 61), 9. print. Philadelphia, Pa: Society for Industrial and Applied Mathematics, 2006, 357 pp., ISBN: 978-0-89871-274-2.

[8] O. Grinenko, J. Li, J. C. Mosher, *et al.*, "A fingerprint of the epileptogenic zone in human epilepsies," *Brain: A Journal of Neurology*, vol. 141, no. 1, pp. 117–131, Jan. 1, 2018, ISSN: 1460-2156. DOI: 10.1093/brain/awx306. PMID: 29253102.

[9] N. Roehri, J.-M. Lina, J. C. Mosher, F. Bartolomei, and C.-G. Bénar, "Time-Frequency Strategies for Increasing High-Frequency Oscillation Detectability in Intracerebral EEG," *IEEE Transactions on Biomedical Engineering*, vol. 63, no. 12, pp. 2595–2606, Dec. 2016, ISSN: 1558-2531. DOI: 10.1109/TBME.2016.2556425. [Online]. Available: https://ieeexplore.ieee.org/document/7458827.

[10] A. Paszke, S. Gross, F. Massa, *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems*, vol. 32, Curran Associates, Inc., 2019. [Online]. Available: https://papers.nips.cc/paper_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[11] S. Tang, J. Dunnmon, K. K. Saab, *et al.*, "Self-Supervised Graph Neural Networks for Improved Electroencephalographic Seizure Analysis," presented at the International Conference on Learning Representations, Oct. 6, 2021. [Online]. Available: https://openreview.net/forum?id=k9bx1EfHI_-.

[12] Y. Li, R. Yu, C. Shahabi, and Y. Liu, "Diffusion Convolutional Recurrent Neural Network: Data-Driven Traffic Forecasting," presented at the International Conference on Learning Representations, Feb. 15, 2018. [Online]. Available: https://openreview.net/forum?id=SJiHXGWAZ.

[13] L. Dinh, J. Sohl-Dickstein, and S. Bengio, "Density estimation using Real NVP," Aug. 17, 2016. [Online]. Available: https://openreview.net/forum?id=SyPNSAW5.

[14] P. Kirichenko, P. Izmailov, and A. G. Wilson, "Why Normalizing Flows Fail to Detect Out-of-Distribution Data," in *Advances in Neural Information Processing Systems*, vol. 33, Curran Associates, Inc., 2020, pp. 20 578–20 589. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/ecb9fe2fbb99c31f567e9823e884dbec-Abstract.html.

[15] A. Ryzhikov, M. Borisyak, A. Ustyuzhanin, and D. Derkach, "NFAD: Fixing anomaly detection using normalizing flows," *PeerJ Computer Science*, vol. 7, e757, Nov. 18, 2021, ISSN: 2376-5992. DOI: 10.7717/peerj-cs.757. [Online]. Available: https://peerj.com/articles/cs-757.

[16] L.-L. Chiu and S.-H. Lai, "Self-Supervised Normalizing Flows for Image Anomaly Detection and Localization," in *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, Jun. 2023, pp. 2927–2936. DOI: 10.1109/CVPRW59228.2023.00294. [Online]. Available: https://ieeexplore.ieee.org/document/10208652.

[17] D. Wang, H.-W. Wang, K.-F. Lu, Z.-R. Peng, and J. Zhao, "Regional Prediction of Ozone and Fine Particulate Matter Using Diffusion Convolutional Recurrent Neural Network," *International Journal of Environmental Research and Public Health*, vol. 19, no. 7, p. 3988, Mar. 27, 2022, ISSN: 1660-4601. DOI: 10.3390/ijerph19073988. PMID: 35409671.

[18] P. Khosla, P. Teterwak, C. Wang, *et al.*, "Supervised Contrastive Learning," in *Advances in Neural Information Processing Systems*, vol. 33, Curran Associates, Inc., 2020, pp. 18 661–18 673. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/d89a66c7c80a29b1bdbab0f2a1a94af8-Abstract.html.

[19] Y. Cui, M. Jia, T.-Y. Lin, Y. Song, and S. Belongie, "Class-Balanced Loss Based on Effective Number of Samples," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2019, pp. 9260–9269. DOI: 10.1109/CVPR.2019.00949. [Online]. Available: https://ieeexplore.ieee.org/document/8953804.

[20] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal Loss for Dense Object Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 2, pp. 318–327, Feb. 2020, ISSN: 1939-3539. DOI: 10.1109/TPAMI.2018.2858826. [Online]. Available: https://ieeexplore.ieee.org/document/8417976.

A Python Implementation Details

A.1 Data Processing and Feature Extraction

A.1.1 Db-4 Wavelet Packet Transform PyTorch implementation

```
import torch
import numpy as np
import math
from typing import Tuple, Union

def db4_dwt(x: Union[torch.Tensor, np.ndarray]) -> Tuple[torch.Tensor, torch.Tensor]:
    """
    Computes the Daubechies 4 wavelet transform of the input.

    Args:
    x: A PyTorch tensor or NumPy array with shape (num_ch, num_samples)

    Returns:
    A tuple containing the approximation and detail coefficients.
    """
    if x.dim() != 2:
        raise ValueError("Input must have shape (num_ch, num_samples)")

    device = x.device
    sqrt2 = math.sqrt(2)
    sqrt3 = math.sqrt(3)
    h0 = (1 + sqrt3) / (4 * sqrt2)
    h1 = (3 + sqrt3) / (4 * sqrt2)
    h2 = (3 - sqrt3) / (4 * sqrt2)
    h3 = (1 - sqrt3) / (4 * sqrt2)
    low_filter = torch.tensor([h0, h1, h2, h3], dtype=x.dtype, device=device).view(1, 1, -1)
    high_filter = torch.tensor([h3, -h2, h1, -h0], dtype=x.dtype, device=device).view(1, 1, -1)

    unsqueezed = False
    if x.shape[0] == 1:
        x = x.unsqueeze(1)
        unsqueezed = True
    else:
        x = x.unsqueeze(1)

    try:
        approx = torch.nn.functional.conv1d(x, low_filter, stride=2)
        detail = torch.nn.functional.conv1d(x, high_filter, stride=2)
    except Exception as e:
        raise RuntimeError("Error during convolution") from e

    approx = approx.squeeze(1)
```

```
    detail = detail.squeeze(1)

    if unsqueezed:
        approx = approx.squeeze(0)
        detail = detail.squeeze(0)

    return approx, detail

def _wpt_recursive(x: torch.Tensor, level: int) -> list:
    if level <= 0:
        return [x]
    else:
        approx, detail = db4_dwt(x)
        return _wpt_recursive(approx, level - 1) + _wpt_recursive(detail, level - 1)

def db4_wpt(x: Union[torch.Tensor, np.ndarray], level: int) -> torch.Tensor:
    """
    Computes the wavelet packet transform of the input up to the specified level.

    Args:
    x: A PyTorch tensor or NumPy array with shape (num_ch, num_samples)
    level: The level of the wavelet packet transform

    Returns:
    A PyTorch tensor containing the concatenated leaves of the wavelet packet tree.
    """
    x = torch.as_tensor(x)
    if x.dim() != 2:
        raise ValueError("Input must have shape (num_ch, num_samples)")
    if level < 0:
        raise ValueError("Level must be a non-negative integer")

    leaves = _wpt_recursive(x, level)
    output = torch.cat(leaves, dim=-1)
    return output
```

A.1.2 HUP+CHUL Dataset interface

```
from pyedflib import highlevel
from torch.utils.data import Dataset
from torch.nn.utils.rnn import pad_sequence
from scipy import signal
import pickle
import os
import glob
import csv
from itertools import accumulate
from bisect import bisect_right
import numpy as np
import torch
from wavelets import db4_wpt
import math

class HUP_CHUL_Dataset(Dataset):
    def __init__(self, hup_path, chul_path, patient_ids, save_dir, clip_time, wpt_level, overlap_rate=0.0, mode="train",
    ↪  relative_noise_std_factor=0.0, channel_dropout_rate=0.0):
```

```
        """
        Args:
            hup_path (str): Path to HUP dataset.
            chul_path (str): Path to CHUL dataset.
            patient_ids (list): List of patient IDs to include.
            save_dir (str): Directory to save/load preprocessed .pkl files.
            clip_time (float): Duration of each clip in seconds.
            wpt_level (int): Level for Wavelet Packet Transform.
            overlap_rate (float): Fraction of overlap (0.0 <= overlap_rate < 1.0).
            mode (str): Dataset mode ('train', 'eval', 'test'). Augmentations applied only in 'train'.
            relative_noise_std_factor (float): Relative standard deviation of gaussian noise added to raw signal
            channel_dropout_rate (float): Probability of zeroing out a channel (train mode only).
        """
        self.NOTCH_Q = 40.0
        self.FS_NEW = 256.0
        self.CLIP_SIZE = int(clip_time * self.FS_NEW)
        self.WPT_LEVEL = wpt_level
        self.save_dir = save_dir


        # --- Mode and Augmentation Params ---
        if mode not in ['train', 'eval', 'test']:
            raise ValueError("mode must be 'train', 'eval', or 'test'")
        self.mode = mode
        self.relative_noise_std_factor = relative_noise_std_factor if self.mode == 'train' else 0.0
        self.channel_dropout_rate = channel_dropout_rate if self.mode == 'train' else 0.0
        if not (0.0 <= self.channel_dropout_rate < 1.0):
             raise ValueError("channel_dropout_rate must be between 0.0 (inclusive) and 1.0 (exclusive)")
        # --- End Mode/Augmentation ---

        if not (0.0 <= overlap_rate < 1.0):
            raise ValueError("overlap_rate must be between 0.0 (inclusive) and 1.0 (exclusive)")
        self.overlap_rate = overlap_rate
        self.stride = int(self.CLIP_SIZE * (1.0 - self.overlap_rate))

        self.edf_paths = self._get_edf_paths(hup_path, chul_path, patient_ids)
        self.sigs = [self._get_sig(edf_path) for edf_path in self.edf_paths]
        self.num_clips_cumulative = self._get_num_clips_cumulative()

    def _get_edf_paths(self, hup_path, chul_path, patient_ids):
        def get_edf_paths(patient_id):
            base_path = hup_path if "HUP" in patient_id else chul_path
            sub_path = "ses-presurgery" if "HUP" in patient_id else ""
            search_path = os.path.join(base_path, patient_id, sub_path, "ieeg", "*.edf")
            return sorted(glob.glob(search_path))

        return [path for patient_id in patient_ids for path in get_edf_paths(patient_id)]

    def _get_sig(self, edf_path):
        # returns tuple (patient_id,sig,ch,soz) after filtering and resampling
        patient_id = os.path.basename(edf_path).split("_")[0]
        pkl_fn_base = os.path.basename(edf_path).split(".")[0]
        pkl_fn = f"{pkl_fn_base}_fs{int(self.FS_NEW)}_notch{int(self.NOTCH_Q)}.pkl"
        pkl_path = os.path.join(self.save_dir, patient_id, pkl_fn)

        if os.path.exists(pkl_path):
            try:
                with open(pkl_path, "rb") as fh:
                    sig, ch, soz = pickle.load(fh)
            except Exception as e:
                os.remove(pkl_path)
                sig, ch, soz = self._process_edf(edf_path, pkl_path)
        else:
            sig, ch, soz = self._process_edf(edf_path, pkl_path)
        return patient_id, sig, ch, soz

    def _process_edf(self, edf_path, pkl_path):
        """Helper function to process a single EDF file."""
        ch, soz, sig_mask = [], [], []
        try:
            signals, signal_headers, header = highlevel.read_edf(edf_path)
            if signals is None or (isinstance(signals, np.ndarray) and signals.size == 0) or not signal_headers:
                print(f"Warning: No valid signals data or headers read from {edf_path}. Skipping.")
                # Return empty tensors/lists to avoid downstream errors
                return torch.empty((0,0), dtype=torch.float32), [], torch.empty((0,), dtype=torch.bool)

            fs_read = signal_headers[0]["sample_frequency"]
            channels_tsv = "_".join(edf_path.split("_")[:-1]) + "_channels.tsv"
            if not os.path.exists(channels_tsv):
                 print(f"Warning: Channels TSV file not found: {channels_tsv}. Cannot determine SOZ/status for {edf_path}.
                 ↪  Skipping.")
                 return torch.empty((0,0), dtype=torch.float32), [], torch.empty((0,), dtype=torch.bool)
```

```
            with open(channels_tsv) as fh:
                reader = csv.reader(fh, delimiter="\t")
                column_names = next(reader)
                # Check for required column names robustly
                required_cols = ["name", "status", "type"]
                if "HUP" in edf_path: required_cols.append("status_description")
                else: required_cols.append("soz_label")
                if not all(col in column_names for col in required_cols):
                    print(f"Warning: Missing required columns in {channels_tsv} (needs {required_cols}). Skipping {edf_path}.")
                    return torch.empty((0,0), dtype=torch.float32), [], torch.empty((0,), dtype=torch.bool)

                # Get indices after validation
                channel_name_idx = column_names.index("name")
                status_idx = column_names.index("status")
                signal_type_idx = column_names.index("type")
                status_description_idx = column_names.index("status_description") if "HUP" in edf_path else -1
                soz_label_idx = column_names.index("soz_label") if "HUP" not in edf_path else -1


                for row in reader:
                    channel_name = row[channel_name_idx]
                    status = row[status_idx]
                    signal_type = row[signal_type_idx]
                    is_good_seeg = (status == "good" and signal_type == "SEEG")

                    if is_good_seeg:
                        sig_mask.append(True)
                        ch.append(channel_name)
                        is_soz = False
                        if "HUP" in edf_path:
                            status_description = row[status_description_idx]
                            is_soz = "soz" in status_description.lower() # Case-insensitive check
                        else:
                            soz_label = row[soz_label_idx]
                            is_soz = soz_label.lower() == "soz" # Case-insensitive check
                        soz.append(is_soz)
                    else:
                        sig_mask.append(False)

            if not any(sig_mask): # Check if any good SEEG channels were found
                print(f"Warning: No 'good' SEEG channels found in {edf_path} according to {channels_tsv}. Skipping.")
                return torch.empty((0,0), dtype=torch.float32), [], torch.empty((0,), dtype=torch.bool)


            soz = torch.Tensor(soz).to(torch.bool)
            sig = signals[sig_mask]

            # Filter only if signal has non-zero length
            if sig.shape[1] > 0:
                NOTCH_FREQ = 60.0 if "HUP" in edf_path else 50.0
                sos_notch = signal.tf2sos(*signal.iirnotch(NOTCH_FREQ, self.NOTCH_Q, fs_read))
                sig = signal.sosfiltfilt(sos_notch, sig, axis=-1)

                # Resample only if necessary and signal has length
                if not np.isclose(self.FS_NEW, fs_read):
                    num_samples_new = int(np.round(sig.shape[1] * self.FS_NEW / fs_read))
                    if num_samples_new > 0:
                       sig = signal.resample(sig, num_samples_new, axis=-1)
                    else:
                       sig = np.empty((sig.shape[0], 0)) # Handle case where resampling results in 0 samples

            sig=torch.from_numpy(sig.copy()).to(torch.float32)

            # Save the processed data
            os.makedirs(os.path.dirname(pkl_path), exist_ok=True)
            with open(pkl_path, "wb") as fh:
                pickle.dump((sig, ch, soz), fh)

            return sig, ch, soz

        except Exception as e:
            print(f"!!! Critical Error processing {edf_path}: {e}")
            print(f"Skipping this file.")
            # Return empty data to prevent crashes later
            return torch.empty((0,0), dtype=torch.float32), [], torch.empty((0,), dtype=torch.bool)

    def _calculate_num_clips(self, signal_length):
        """Calculates the number of extractable clips from a signal of given length."""
        if signal_length < self.CLIP_SIZE:
            return 0
        else:
            # The index of the last possible clip start position (0-based)
            last_possible_start_index = signal_length - self.CLIP_SIZE
```

```
            # Number of clips = number of valid start positions / stride, rounded down, plus 1
            num_clips = math.floor(last_possible_start_index / self.stride) + 1
            return num_clips

    def _get_num_clips_cumulative(self):
        """Calculates cumulative clip counts using the overlap/stride logic."""
        num_clips_per_file = []
        for _, sig, _, _ in self.sigs:
            if isinstance(sig, torch.Tensor) and sig.ndim >= 2:
                signal_length = sig.shape[1]
                num_clips = self._calculate_num_clips(signal_length)
                num_clips_per_file.append(num_clips)
            else:
                print(f"Warning: Invalid signal data encountered during clip calculation. Shape: {getattr(sig, 'shape', 'N/A')}")
                num_clips_per_file.append(0)
        return list(accumulate(num_clips_per_file))

    def __len__(self):
        """Returns the total number of clips across all files, considering overlap."""
        if not self.num_clips_cumulative: return 0
        return self.num_clips_cumulative[-1]

    def __getitem__(self, global_idx):
        if global_idx >= self.__len__() or global_idx < 0:
             raise IndexError(f"Index {global_idx} out of range for dataset length {self.__len__()}")

        sig_idx = bisect_right(self.num_clips_cumulative, global_idx)
        local_idx = global_idx if sig_idx == 0 else global_idx - self.num_clips_cumulative[sig_idx - 1]
        patient_id, sig, ch, soz = self.sigs[sig_idx]
        start_sample = local_idx * self.stride
        end_sample = start_sample + self.CLIP_SIZE

        if end_sample > sig.shape[1]:
            end_sample = sig.shape[1]
            start_sample = end_sample - self.CLIP_SIZE
            if start_sample < 0: start_sample = 0

        clip = sig[:, start_sample:end_sample]
        if not isinstance(clip, torch.Tensor):
            clip = torch.from_numpy(clip.copy()).to(torch.float32)
        else:
             clip = clip.clone().detach().to(torch.float32)

        # --- Apply Augmentations only in 'train' mode ---
        if self.mode == 'train':
            # 1. Adaptive Gaussian Noise
            if self.relative_noise_std_factor > 0 and clip.numel() > 0: # Check clip not empty
                # Calculate std deviation FOR EACH CHANNEL within the clip
                # Add epsilon to prevent division by zero or issues with flat channels
                clip_std_per_channel = torch.std(clip, dim=1, keepdim=True, unbiased=False) + 1e-6

                # Calculate noise level per channel based on the factor
                noise_level_per_channel = clip_std_per_channel * self.relative_noise_std_factor

                # Generate noise and scale it per channel
                noise = torch.randn_like(clip) * noise_level_per_channel
                clip = clip + noise

            # 2. Channel Dropout
            num_channels = clip.shape[0]
            if self.channel_dropout_rate > 0 and num_channels > 0:
                keep_mask = (torch.rand(num_channels, device=clip.device) > self.channel_dropout_rate).unsqueeze(1)
                clip = clip * keep_mask.float()
        # --- End Augmentations ---

        wpt = db4_wpt(clip, self.WPT_LEVEL)
        return patient_id, wpt, ch, soz

def collate_fn(batch):
    patient_id,wpt,ch,soz = zip(*batch)
    padded_wpt = pad_sequence(wpt, batch_first=True, padding_value=0.)
    padded_soz = pad_sequence(soz, batch_first=True, padding_value=0.)
    key_padding_mask = (padded_wpt == 0.).float().sum(dim=-1) > 0
    max_channels = max(len(channels) for channels in ch)
    padded_ch = [channels+["pad_ch"]*(max_channels-len(channels)) for channels in ch]
    patient_id = list(patient_id)
    return patient_id,padded_wpt,padded_ch,padded_soz,key_padding_mask
```

A.2 Core Model and Training Framework

A.2.1 Model architecture

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class TransformerEncoderLayer(nn.Module):
    """
    A single transformer encoder layer with pre-attention layer normalization.
    """

    def __init__(self, d_model, num_heads, dim_feedforward=2048, dropout=0.1):
        """
        Args:
            d_model (int): The number of expected features in the input (embedding dimension).
            num_heads (int): The number of heads in the multi-head self-attention.
            dim_feedforward (int): The dimension of the feed-forward network model.
            dropout (float): Dropout value.
        """
        super(TransformerEncoderLayer, self).__init__()

        self.self_attn = nn.MultiheadAttention(d_model, num_heads, dropout=dropout, batch_first=False)

        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)

        self.linear1 = nn.Linear(d_model, dim_feedforward)
        self.linear2 = nn.Linear(dim_feedforward, d_model)

    def forward(self, src, src_mask=None, src_key_padding_mask=None):
        """
        Args:
            src (Tensor): Input tensor of shape (sequence_length, batch_size, d_model).
            src_mask (Tensor, optional): Attention mask for sequence masking (size: seq_len x seq_len).
                - Used to prevent positions from attending to future tokens (for causal masking) or to mask specific tokens.
            src_key_padding_mask (Tensor, optional): Padding mask for input tokens (size: batch_size x seq_len).
                - Used to mask out padding tokens so they do not contribute to self-attention.

        Returns:
            Tensor: Output tensor of shape (sequence_length, batch_size, d_model).
        """
        # Pre-attention layer normalization.
        src_norm = self.norm1(src)
        # Apply multi-head self-attention. Note that nn.MultiheadAttention expects input shape:
        # (sequence_length, batch_size, embedding_dim)
        attn_output, _ = self.self_attn(
            src_norm, src_norm, src_norm,
            attn_mask=src_mask, # seq_len x seq_len mask
            key_padding_mask=src_key_padding_mask   # batch_size x seq_len mask
        )

        # Residual connection & dropout
        src = src + self.dropout1(attn_output)

        # Pre-FFN layer normalization
        src_norm = self.norm2(src)

        # Feed-forward network with GELU activation
        ffn_output = self.linear2(nn.functional.gelu(self.linear1(src_norm)))

        # Residual connection & dropout
        src = src + self.dropout2(ffn_output)

        return src

class TransformerEncoder(nn.Module):
    """
    Transformer Encoder consisting of a stack of encoder layers with masking support.
    """

    def __init__(self, d_model, num_heads, num_layers, dim_feedforward=2048, dropout=0.1):
        """
        Args:
            d_model (int): Embedding dimension.
            num_heads (int): Number of attention heads.
            num_layers (int): Number of encoder layers.
            dim_feedforward (int): Dimension of the feed-forward network.
```

```
            dropout (float): Dropout rate.
        """
        super(TransformerEncoder, self).__init__()
        self.layers = nn.ModuleList([
            TransformerEncoderLayer(d_model, num_heads, dim_feedforward, dropout)
            for _ in range(num_layers)
        ])
        self.norm = nn.LayerNorm(d_model)  # Final Layer Normalization

    def forward(self, src, mask=None, src_key_padding_mask=None):
        """
        Args:
            src (Tensor): Input tensor of shape (sequence_length, batch_size, d_model).
            mask (Tensor, optional): Attention mask.
            src_key_padding_mask (Tensor, optional): Padding mask.
        Returns:
            Tensor: Output tensor of shape (sequence_length, batch_size, d_model).
        """
        output = src
        # Iterate through the encoder layers, passing the masks as well
        for layer in self.layers:
            output = layer(output, src_mask=mask, src_key_padding_mask=src_key_padding_mask)
        output = self.norm(output)
        return output

class TransformerEncoderWithProjection(nn.Module):
    """
    Adds a projection layer before the transformer encoder to reduce token size and support masking.
    """

    def __init__(self, d_features, d_model, num_heads, num_layers, dim_feedforward=2048, dropout=0.1):
        """
        Args:
            d_features (int): Input features dimension.
            d_model (int): Embedding dimension.
            num_heads (int): Number of attention heads.
            num_layers (int): Number of encoder layers.
            dim_feedforward (int): Dimension of the feed-forward network.
            dropout (float): Dropout rate.
        """
        super().__init__()
        self.FeatureProjection = nn.Linear(d_features, d_model)  #Projection layer
        self.TransformerEncoder = TransformerEncoder(d_model, num_heads, num_layers, dim_feedforward, dropout)

    def forward(self, input_seq, mask=None, src_key_padding_mask=None):
        """
        Args:
            input_seq (Tensor): Input tensor of shape (batch_size, seq_len, d_features).
            mask (Tensor, optional): Attention mask.
            src_key_padding_mask (Tensor, optional): Padding mask.
        Returns:
            Tensor: Output tensor of shape (sequence_length, batch_size, d_model).
        """

        # Transpose input to match expected Transformer input shape
        # (batch_size, seq_len, d_feature) -> (seq_len, batch_size, d_feature)
        input_seq = input_seq.permute(1, 0, 2)

        # Project input to the model's embedding dimension
        #(seq_len, batch_size, d_model)
        src = self.FeatureProjection(input_seq)

        # Apply Transformer Encoder with proper masking
        #(seq_len, batch_size, d_model)
        output = self.TransformerEncoder(src, mask=mask, src_key_padding_mask=src_key_padding_mask)

        return output

class SOZDetectionPretraining(nn.Module):
    """
    TF Features => Linear => Transformer Encoder => Normalize => MLP Head => output
    <= Contrastive objective on d_hidden dimensional representations from MLP Head
    """
    def __init__(
        self,
        d_features,
        d_model,
        d_hidden, # Dimension of the output from the MLP projection head
        num_heads,
        num_layers,
        dim_feedforward=2048,
        dropout=0.1,
        mlp_intermediate_factor=1 # Factor to determine MLP intermediate dim (d_model * factor)
```

```
                                # Or you could pass a specific dim directly
    ):
        """
        Args:
            d_features (int): Input features dimension.
            d_model (int): Embedding dimension of the Transformer Encoder.
            d_hidden (int): Output dimension of the MLP projection head.
            num_heads (int): Number of attention heads in Transformer.
            num_layers (int): Number of encoder layers in Transformer.
            dim_feedforward (int): Dimension of the feed-forward network in Transformer layers.
            dropout (float): Dropout rate used in Transformer.
            mlp_intermediate_factor (int): Determines intermediate dimension of MLP
                                           (d_model * mlp_intermediate_factor). Common is 1 or 2.
        """
        super().__init__()
        self.FeatureProjection = nn.Linear(d_features, d_model)
        self.TransformerEncoder = TransformerEncoder(
            d_model, num_heads, num_layers, dim_feedforward, dropout
        )

        # --- Define the MLP Projection Head ---
        mlp_intermediate_dim = int(d_model * mlp_intermediate_factor)
        self.projection_head = nn.Sequential(
            nn.Linear(d_model, mlp_intermediate_dim),
            nn.ReLU(),
            nn.Linear(mlp_intermediate_dim, d_hidden)
            # Note: No activation/normalization after the final linear layer is common
            #       in contrastive projection heads (e.g., SimCLR).
        )
        # --- End MLP Definition ---

    def forward(self, input_seq, mask=None, src_key_padding_mask=None):
        """
        Args:
            input_seq (Tensor): Input tensor of shape (batch_size, seq_len, d_features).
            mask (Tensor, optional): Attention mask for transformer.
            src_key_padding_mask (Tensor, optional): Padding mask for transformer.

        Returns:
            Tensor: Output tensor from MLP head of shape (sequence_length, batch_size, d_hidden).
        """
        # Permute to (seq_len, batch_size, d_features)
        input_seq = input_seq.permute(1, 0, 2)

        # Project features to d_model
        src = self.FeatureProjection(input_seq) # [seq_len, batch_size, d_model]

        # Pass through Transformer Encoder (includes final LayerNorm)
        src = self.TransformerEncoder(
            src, mask=mask, src_key_padding_mask=src_key_padding_mask
        ) # [seq_len, batch_size, d_model]

        # Apply the MLP projection head
        projected_output = self.projection_head(src) # [seq_len, batch_size, d_hidden]

        return projected_output

class SOZDetectionFinetuning(nn.Module):
    """
    Fine-tuning model: Takes pre-trained backbone and adds a linear head.
    TF Features => Linear => Transformer Encoder => Linear => **Unbounded scores**
    """
    def __init__(self, pretrained): # Add type hint: e.g., pretrained: SOZDetectionPretraining
        super().__init__()
        self.pretrained = pretrained
        d_model = pretrained.FeatureProjection.out_features # Or however d_model is stored/accessed
        self.classification_head = nn.Linear(d_model, 1) # Renamed for clarity

    def forward(self, input_seq, mask=None, src_key_padding_mask=None):
        input_seq = input_seq.permute(1, 0, 2)
        # Pass through pre-trained backbone
        features = self.pretrained.FeatureProjection(input_seq)
        features = self.pretrained.TransformerEncoder(
            features, mask=mask, src_key_padding_mask=src_key_padding_mask
        ) # Output shape: [seq_len, batch_size, d_model]

        # Apply classification head directly to backbone features
        logits = self.classification_head(features) # Output shape: [seq_len, batch_size, 1]
        return logits

class SOZDetection(nn.Module):
    """
    TF Features => Linear => Transformer Encoder => Normalize => Linear => **Probability estimates**
```

```
    <= Model to use in prediction, that should inherit from the two previous models.
    """
    def __init__(self, model):
        super().__init__()
        self.model = model #model is SOZDetectionFinetuning
        self.sig = nn.Sigmoid()

    def forward(self, input_seq, mask=None, src_key_padding_mask=None):
        """
        Args:
            input_seq (Tensor): Input tensor of shape (batch_size, seq_len, d_features).
            mask (Tensor, optional): Attention mask.
            src_key_padding_mask (Tensor, optional): Padding mask.
        Returns:
            prob_scores: Output tensor of shape (sequence_length, batch_size, 1).
            probability scores between 0 and 1 for the SOZ
        """
        unbounded_scores=self.model(input_seq, mask=mask, src_key_padding_mask=src_key_padding_mask) #(seq_len, batch_size, 1)
        prob_scores=self.sig(unbounded_scores) #(seq_len, batch_size, 1)
        return prob_scores
```

A.2.2 Loss functions

```
import torch
import torch.nn.functional as F
from torch.nn.functional import binary_cross_entropy_with_logits, sigmoid

def contrastive_loss(h, padded_soz, src_key_padding_mask, mplus=0.9, mminus=-0.9):
    """
    A spatial contrastive loss for SOZ detection
    Args:
        h ([max_seq_len, batch_size, d_model]): output representation
        padded_soz ([batch_size, max_seq_len]): Multi-hot encoded epileptogenic zone label
        src_key_padding_mask ([batch_size, max_seq_len]): padding mask (False for valid tokens)
    Returns:
        loss: contrastive loss that will cluster the latent space according to the SOZ
    """
    loss=0.0
    valid_lengths=(~src_key_padding_mask).sum(dim=1)
    batch_size=len(valid_lengths)
    for i,valid_length in enumerate(valid_lengths):
        h_norm = torch.nn.functional.normalize(h[:valid_length,i,:], p=2, dim=1)
        S = h_norm@h_norm.T

        ez_multihot=padded_soz[i,:valid_length]
        remove_diag=~torch.eye(len(ez_multihot), dtype=torch.bool, device=ez_multihot.device)
        #both_not_ez=~ez_multihot.unsqueeze(0) & ~ez_multihot.unsqueeze(1) & remove_diag
        both_ez=ez_multihot.unsqueeze(0) & ez_multihot.unsqueeze(1) & remove_diag
        one_in_one_out=(ez_multihot.unsqueeze(0) & ~ez_multihot.unsqueeze(1)) | (~ez_multihot.unsqueeze(0) &
        ↪  ez_multihot.unsqueeze(1))

        #coords=padded_coords[i,:valid_length]
        #D_sq = ((coords.unsqueeze(0) - coords.unsqueeze(1))**2).sum(dim=2)
        #sigma=torch.std(torch.sqrt(D_sq))
        #D_kernel = torch.exp(-D_sq / (sigma**2))
        both_ez_terms=both_ez*torch.nn.functional.relu(mplus-S)*(1/(both_ez.sum()))
        #both_not_ez_terms=both_not_ez*torch.nn.functional.relu(mplus-S)*(1/(both_not_ez.sum()))
        one_in_one_out_terms=one_in_one_out*torch.nn.functional.relu(S-mminus)*(1/(one_in_one_out.sum()))
        loss_terms=one_in_one_out_terms+both_ez_terms
        loss+=torch.mean(loss_terms)
    return loss/batch_size


def cb_bce_loss_with_logits(
    unbounded_scores,           # Shape: [max_seq_len, batch_size, 1]
    padded_soz,                 # Shape: [batch_size, max_seq_len] (True/False or 1/0)
    src_key_padding_mask,       # Shape: [batch_size, max_seq_len] (True for padding)
    beta=0.999,                 # Class balancing hyperparameter
    epsilon=1e-8                # Numerical stability
    ):
    """
    Class-Balanced Binary Cross Entropy Loss With Logits.

    Applies BCE loss with dynamic weights calculated per sample based on the
    effective number of positive (SOZ) and negative (non-SOZ) examples
    within that sample, using the class-balancing formula.

    Args:
        unbounded_scores ([max_seq_len, batch_size, 1]): Raw output scores from the model (before sigmoid).
        padded_soz ([batch_size, max_seq_len]): Boolean or 0/1 tensor indicating SOZ status.
```

```
        src_key_padding_mask ([batch_size, max_seq_len]): Padding mask (True for padding, False for valid tokens).
        beta (float): Hyperparameter for class balancing weight calculation (e.g., 0.9, 0.99, 0.999).
        epsilon (float): Small value for numerical stability.

    Returns:
        torch.Tensor: The final computed class-balanced loss scalar, averaged over all valid channels in the batch.
    """
    total_loss = 0.0
    total_valid_channels = 0.0 # Normalizing by total valid channels across batch

    valid_lengths = (~src_key_padding_mask).sum(dim=1)
    batch_size = len(valid_lengths)

    # Ensure padded_soz is float for BCE calculation later
    padded_soz_float = padded_soz.float()

    for i, valid_length in enumerate(valid_lengths):
        if valid_length == 0: # Skip samples with no valid channels
            continue

        # Extract valid scores and SOZ labels for this sample
        scores_valid = unbounded_scores[:valid_length, i, :].squeeze(-1) # Shape: [valid_length]
        soz_valid = padded_soz_float[i, :valid_length]                # Shape: [valid_length]
        soz_valid_bool = padded_soz[i, :valid_length].bool()           # Shape: [valid_length], boolean for counting

        # --- Calculate Class Counts for this specific sample ---
        N_soz = soz_valid_bool.sum().float()        # Ensure float for power calculation
        N_non_soz = valid_length.float() - N_soz  # Ensure float

        # --- Calculate Class-Balanced Weights for this sample ---
        if N_soz > 0:
            weight_soz = (1.0 - beta) / (1.0 - torch.pow(beta, N_soz) + epsilon)
        else:
            weight_soz = 0 # Assign 0 weight if no positive samples exist in this instance

        if N_non_soz > 0:
            weight_non_soz = (1.0 - beta) / (1.0 - torch.pow(beta, N_non_soz) + epsilon)
        else:
            weight_non_soz = 0 # Assign 0 weight if no negative samples exist

        # --- Create per-channel weight tensor ---
        # Assign weight_soz to positive samples, weight_non_soz to negative samples
        sample_weights = torch.where(soz_valid_bool, weight_soz, weight_non_soz) # Shape: [valid_length]

        # --- Calculate Weighted BCE Loss for this sample ---
        # Apply weights element-wise, then sum the loss for this sample
        sample_loss = F.binary_cross_entropy_with_logits(
            scores_valid,
            soz_valid,
            weight=sample_weights, # Apply per-channel weights
            reduction='sum'        # Sum the loss across channels for this sample
        )

        total_loss += sample_loss
        total_valid_channels += valid_length.float() # Accumulate total valid channels processed

    # Average the total loss over all valid channels processed in the batch
    final_loss = total_loss / (total_valid_channels + epsilon)

    return final_loss


def focal_cb_bce_loss_with_logits(
    unbounded_scores,           # Shape: [max_seq_len, batch_size, 1]
    padded_soz,                 # Shape: [batch_size, max_seq_len] (True/False or 1/0)
    src_key_padding_mask,       # Shape: [batch_size, max_seq_len] (True for padding)
    beta=0.999,                 # Class balancing hyperparameter
    gamma=2.0,                  # Focal loss focusing parameter (gamma=0 recovers non-focal version)
    epsilon=1e-8                # Numerical stability
    ):
    """
    Focal Class-Balanced Binary Cross Entropy Loss With Logits.

    Combines Focal Loss modulation (down-weights easy examples) with
    Class-Balanced weighting (based on effective number of samples per class
    within each sample).

    Args:
        unbounded_scores ([max_seq_len, batch_size, 1]): Raw output scores from the model (before sigmoid).
        padded_soz ([batch_size, max_seq_len]): Boolean or 0/1 tensor indicating SOZ status.
        src_key_padding_mask ([batch_size, max_seq_len]): Padding mask (True for padding, False for valid tokens).
        beta (float): Hyperparameter for class balancing weight calculation.
        gamma (float): Focusing parameter for Focal loss. gamma=0 disables the focal component.
```

```
        epsilon (float): Small value for numerical stability.

    Returns:
        torch.Tensor: The final computed focal class-balanced loss scalar, averaged over valid channels.
    """
    total_loss = 0.0
    total_valid_channels = 0.0 # Normalizing by total valid channels across batch

    valid_lengths = (~src_key_padding_mask).sum(dim=1)
    batch_size = len(valid_lengths)

    padded_soz_float = padded_soz.float() # Ensure float for BCE targets

    for i, valid_length in enumerate(valid_lengths):
        if valid_length == 0: continue

        scores_valid = unbounded_scores[:valid_length, i, :].squeeze(-1) # [valid_length]
        soz_valid = padded_soz_float[i, :valid_length]                # [valid_length] (float)
        soz_valid_bool = padded_soz[i, :valid_length].bool()           # [valid_length] (bool)

        # --- Calculate Class Counts ---
        N_soz = soz_valid_bool.sum().float()
        N_non_soz = valid_length.float() - N_soz

        # --- Calculate Class-Balanced Weights ---
        if N_soz > 0: weight_soz = (1.0 - beta) / (1.0 - torch.pow(beta, N_soz) + epsilon)
        else: weight_soz = 0
        if N_non_soz > 0: weight_non_soz = (1.0 - beta) / (1.0 - torch.pow(beta, N_non_soz) + epsilon)
        else: weight_non_soz = 0
        cb_weights = torch.where(soz_valid_bool, weight_soz, weight_non_soz) # [valid_length]

        # --- Calculate Standard Weighted BCE (Element-wise) ---
        # Need element-wise loss before applying focal modulation
        bce_loss_elements = F.binary_cross_entropy_with_logits(
            scores_valid,
            soz_valid,
            reduction='none' # Get per-element loss
        )

        # --- Calculate Focal Modulation Factor ---
        if gamma > 0:
            # Calculate probabilities p = sigmoid(scores)
            probs = torch.sigmoid(scores_valid)
            # Calculate p_t (probability of the true class)
            p_t = torch.where(soz_valid_bool, probs, 1.0 - probs)
            # Calculate focal weight: (1 - p_t)^gamma
            # Add epsilon inside pow for stability? Maybe not needed if p_t is bound [0,1]
            focal_modulator = torch.pow(1.0 - p_t, gamma)
        else:
            # If gamma is 0, focal modulator is 1 (no effect)
            focal_modulator = torch.ones_like(bce_loss_elements)

        # --- Combine Losses and Weights ---
        # Final loss per element = focal_modulator * class_balance_weight * bce_loss
        combined_loss_elements = focal_modulator * cb_weights * bce_loss_elements

        # Sum the loss for this sample
        sample_loss = combined_loss_elements.sum()

        total_loss += sample_loss
        total_valid_channels += valid_length.float()

    # Average the total loss over all valid channels processed in the batch
    final_loss = total_loss / (total_valid_channels + epsilon)

    return final_loss

def get_precision_micro(unbounded_scores, padded_soz, src_key_padding_mask, threshold):
    """Calculates precision by aggregating counts across the batch."""
    total_tp = 0.0
    total_predicted_p = 0.0

    valid_lengths = (~src_key_padding_mask).sum(dim=1)
    # batch_size = len(valid_lengths) # Not directly needed for micro average

    for batch_idx, valid_length in enumerate(valid_lengths):
        if valid_length == 0: continue
        unbounded_scores_valid = unbounded_scores[:valid_length, batch_idx, :].squeeze(-1) # Use -1 for robustness
        soz_preds = (F.sigmoid(unbounded_scores_valid) > threshold).float()
        soz_valid = padded_soz[batch_idx, :valid_length].float()

        total_tp += ((soz_preds == 1.) & (soz_valid == 1.)).float().sum().item()
        total_predicted_p += soz_preds.sum().item()
```

```
    if total_predicted_p == 0.:
        # Handle case where no positive predictions were made in the entire batch
        # Can return 0.0, 1.0 (if no true positives either), or NaN depending on preference
        return 0.0
    precision = total_tp / total_predicted_p
    return precision

def get_recall_micro(unbounded_scores, padded_soz, src_key_padding_mask, threshold):
    """Calculates recall by aggregating counts across the batch."""
    total_tp = 0.0
    total_true_p = 0.0

    valid_lengths = (~src_key_padding_mask).sum(dim=1)

    for batch_idx, valid_length in enumerate(valid_lengths):
        if valid_length == 0: continue
        unbounded_scores_valid = unbounded_scores[:valid_length, batch_idx, :].squeeze(-1)
        soz_preds = (F.sigmoid(unbounded_scores_valid) > threshold).float()
        soz_valid = padded_soz[batch_idx, :valid_length].float()

        total_tp += ((soz_preds == 1.) & (soz_valid == 1.)).float().sum().item()
        total_true_p += soz_valid.sum().item()

    if total_true_p == 0.:
        # Handle case where no true positives exist in the entire batch
        # Can return 0.0, 1.0 (if no TPs either), or NaN
        return 1.0 if total_tp == 0 else 0.0 # Common convention: 1.0 if no positives to find and none found
    recall = total_tp / total_true_p
    return recall

def get_fpr_micro(unbounded_scores, padded_soz, src_key_padding_mask, threshold):
    """Calculates FPR by aggregating counts across the batch."""
    total_fp = 0.0
    total_true_n = 0.0

    valid_lengths = (~src_key_padding_mask).sum(dim=1)

    for batch_idx, valid_length in enumerate(valid_lengths):
        if valid_length == 0: continue
        unbounded_scores_valid = unbounded_scores[:valid_length, batch_idx, :].squeeze(-1)
        soz_preds = (F.sigmoid(unbounded_scores_valid) > threshold).float()
        soz_valid = padded_soz[batch_idx, :valid_length].float()

        total_fp += ((soz_preds == 1.) & (soz_valid == 0.)).float().sum().item()
        total_true_n += (soz_valid == 0.).float().sum().item() # Or valid_length - soz_valid.sum()

    if total_true_n == 0.:
        # Handle case where no true negatives exist in the entire batch
        return 0.0 # Common: if no negatives to misclassify, FPR is 0
    fpr = total_fp / total_true_n
    return fpr

def get_accuracy_micro(unbounded_scores, padded_soz, src_key_padding_mask, threshold):
    """Calculates accuracy by aggregating counts across the batch."""
    total_correct_predictions = 0.0
    total_valid_channels = 0.0

    valid_lengths = (~src_key_padding_mask).sum(dim=1)

    for batch_idx, valid_length in enumerate(valid_lengths):
        if valid_length == 0: continue
        unbounded_scores_valid = unbounded_scores[:valid_length, batch_idx, :].squeeze(-1)
        soz_preds = (F.sigmoid(unbounded_scores_valid) > threshold).float()
        soz_valid = padded_soz[batch_idx, :valid_length].float()

        total_correct_predictions += (soz_preds == soz_valid).float().sum().item()
        total_valid_channels += valid_length.item()

    if total_valid_channels == 0.:
        return 0.0
    accuracy = total_correct_predictions / total_valid_channels
    return accuracy

def supcon(h, padded_soz, src_key_padding_mask, tau=0.5, epsilon=1e-8):
    """
    A spatial contrastive loss for SOZ detection based on Supcon (https://arxiv.org/pdf/2004.11362)
    Args:
        h ([max_seq_len, batch_size, d_model]): hidden representation
        padded_soz ([batch_size, max_seq_len]): Multi-hot encoded SOZ label
        src_key_padding_mask ([batch_size, max_seq_len]): padding mask (False for valid tokens)
        tau: temperature parameter
        epsilon: small value to ensure numerical stability
```

```
    Returns:
        loss: contrastive loss that will cluster the latent space according to the SOZ
    """
    batch_loss = 0.0
    valid_lengths = (~src_key_padding_mask).sum(dim=1)
    batch_size = len(valid_lengths)

    for batch_idx, num_chan in enumerate(valid_lengths):
        h_pat = h[:num_chan, batch_idx, :]
        sim_pat = h_pat @ h_pat.T / tau
        soz_pat = padded_soz[batch_idx, :num_chan]

        ignore_current = 1 - torch.eye(num_chan, device=h.device)
        exp_sim = torch.exp(sim_pat)
        denom = (exp_sim * ignore_current).sum(dim=1, keepdim=True) + epsilon
        positives = (soz_pat.unsqueeze(0) == soz_pat.unsqueeze(1)) & ignore_current.bool()
        positives = positives.float()
        log_prob = torch.log(exp_sim / denom)
        loss_terms = -(positives * log_prob).sum(dim=1) / (positives.sum(dim=1) + epsilon)
        batch_loss += loss_terms.sum()

    loss=batch_loss/batch_size
    return loss

def weighted_supcon(h, padded_soz, src_key_padding_mask,
                                w_soz=9.0, w_non_soz=1.0,
                                tau=0.1, epsilon=1e-8):
    """
    Weighted Supervised Contrastive loss based on anchor class frequency.

    Args:
        h ([max_seq_len, batch_size, d_model]): hidden representation
        padded_soz ([batch_size, max_seq_len]): Multi-hot encoded SOZ label (1 for SOZ, 0 for non-SOZ)
        src_key_padding_mask ([batch_size, max_seq_len]): padding mask (False for valid tokens)
        w_soz: Weight for SOZ anchors.
        w_non_soz: Weight for non-SOZ anchors.
        tau: temperature parameter
        epsilon: small value to ensure numerical stability
    Returns:
        loss: weighted contrastive loss
    """
    batch_loss = 0.0
    total_weighted_anchors = 0.0 # Keep track of total weight for normalization

    valid_lengths = (~src_key_padding_mask).sum(dim=1)
    batch_size = len(valid_lengths)

    for batch_idx, num_chan in enumerate(valid_lengths):
        if num_chan <= 1: # Need at least 2 channels
            continue

        h_pat = h[:num_chan, batch_idx, :] # Representations for this patient [num_chan, d_model]
        soz_pat = padded_soz[batch_idx, :num_chan] # Labels for this patient [num_chan]

        # --- Check for edge cases ---
        num_positives = soz_pat.sum()
        if num_positives == 0 or num_positives == num_chan:
             # Skip if all channels are the same class (no contrast possible for SupCon)
             # Or handle differently if needed (e.g., add a small penalty?)
             continue
        # --- End Edge Case Check ---

        # Normalize features (important for cosine similarity interpretation)
        h_pat_norm = F.normalize(h_pat, p=2, dim=1)

        # Similarity matrix [num_chan, num_chan]
        # Using matrix multiplication for cosine similarity on normalized features
        sim_pat_mat = h_pat_norm @ h_pat_norm.T / tau

        # --- Standard SupCon calculations ---
        # Mask for positives (same label, excluding self)
        pos_mask = (soz_pat.unsqueeze(0) == soz_pat.unsqueeze(1)).float()
        diag_mask = 1 - torch.eye(num_chan, device=h.device)
        pos_mask = pos_mask * diag_mask # Exclude self-similarity

        # Numerator calculation (log prob for positive pairs)
        # Use log-sum-exp trick for numerical stability
        # log( exp(sim_ij / tau) / sum_k!=i(exp(sim_ik / tau)) ) = sim_ij/tau - log(sum_k!=i(exp(sim_ik/tau)))
        log_sum_exp_sim = torch.logsumexp(sim_pat_mat * diag_mask, dim=1, keepdim=True)
        log_prob = sim_pat_mat - log_sum_exp_sim

        # SupCon loss per anchor (before weighting)
        # Sum log_prob over positive pairs for each anchor, normalize by num positive pairs
```

```
            loss_terms_unweighted = -(pos_mask * log_prob).sum(dim=1) / (pos_mask.sum(dim=1) + epsilon)

            # --- Anchor Weighting ---
            # Create weights based on anchor label
            anchor_weights = torch.where(soz_pat == 1, w_soz, w_non_soz)

            # Apply weights to the loss terms
            weighted_loss_terms = loss_terms_unweighted * anchor_weights

            # Accumulate weighted loss and the total weight applied in this sample
            batch_loss += weighted_loss_terms.sum()
            total_weighted_anchors += anchor_weights.sum() # Sum of weights for anchors in this sample

        # Normalize by the total weight of anchors processed across the batch
        loss = batch_loss / (total_weighted_anchors + epsilon)

        return loss

    def focal_cb_supcon(h, padded_soz, src_key_padding_mask,
                        beta=0.999,    # Hyperparameter for class balancing weight calculation
                        gamma=2.0,     # Hyperparameter for Focal loss modulation (gamma=0 means no focal loss)
                        tau=0.2,       # Temperature parameter
                        epsilon=1e-8): # Small value for numerical stability
        """
        Focal Class-Balanced Supervised Contrastive loss (using small weights).

        Combines Class-Balanced weighting (based on effective number of samples)
        with a Focal loss adaptation applied to the positive pairs to focus on harder positives.
        Gives a "small weights" formulation for class balancing.

        Args:
            h ([max_seq_len, batch_size, d_model]): Hidden representation output from model.
            padded_soz ([batch_size, max_seq_len]): Multi-hot encoded SOZ label (1 for SOZ, 0 for non-SOZ).
            src_key_padding_mask ([batch_size, max_seq_len]): Padding mask (True for padding, False for valid tokens).
            beta (float): Hyperparameter for class balancing weight calculation (0.9, 0.99, 0.999 are common).
                          Higher beta gives more weight to rarer classes.
            gamma (float): Focusing parameter for Focal loss. gamma=0 recovers standard SupCon NLL term.
                           gamma > 0 increases the relative loss for misclassified/low-probability positives.
            tau (float): Temperature parameter for scaling similarities.
            epsilon (float): Small value added to denominators to prevent division by zero.

        Returns:
            torch.Tensor: The final computed loss scalar.
        """
        batch_loss = 0.0
        total_anchors = 0.0 # Normalizing by anchor count

        valid_lengths = (~src_key_padding_mask).sum(dim=1)
        batch_size = len(valid_lengths)

        for batch_idx, num_chan in enumerate(valid_lengths):
            if num_chan <= 1: # Need at least 2 channels for contrastive loss
                continue

            h_pat = h[:num_chan, batch_idx, :]        # Representations for this patient [num_chan, d_model]
            soz_pat = padded_soz[batch_idx, :num_chan] # Labels for this patient [num_chan]

            # --- Calculate Class Counts ---
            N_soz = soz_pat.sum()
            N_non_soz = num_chan - N_soz

            # Check if all channels are the same class (no contrast possible)
            if N_soz == 0 or N_non_soz == 0:
                continue
            # --- End Class Counts ---

            # Normalize features (important for cosine similarity interpretation)
            h_pat_norm = F.normalize(h_pat, p=2, dim=1)

            # Similarity matrix [num_chan, num_chan]
            sim_pat_mat = h_pat_norm @ h_pat_norm.T / tau

            # --- Identify positive pairs ---
            pos_mask = (soz_pat.unsqueeze(0) == soz_pat.unsqueeze(1)).float()
            diag_mask = 1 - torch.eye(num_chan, device=h.device)
            pos_mask = pos_mask * diag_mask # Exclude self-similarity

            # --- Calculate Softmax Probabilities & Log Probabilities (Core SupCon calculation) ---
            # Use stable log-sum-exp trick for denominator
            log_sum_exp_sim = torch.logsumexp(sim_pat_mat * diag_mask, dim=1, keepdim=True)
            log_prob_all_pairs = sim_pat_mat - log_sum_exp_sim
            nll_all_pairs = -log_prob_all_pairs
```

```
        # --- Focal Loss Adaptation for Positive Pairs ---
        if gamma > 0:
            softmax_probs = torch.exp(log_prob_all_pairs)
            focal_modulator = torch.pow(1. - softmax_probs, gamma)
            modulated_nll_all_pairs = focal_modulator * nll_all_pairs
        else:
            modulated_nll_all_pairs = nll_all_pairs

        # --- Sum Modulated Loss over Positive Pairs for each Anchor ---
        focal_sum_per_anchor = (pos_mask * modulated_nll_all_pairs).sum(dim=1)
        num_positives_per_anchor = pos_mask.sum(dim=1)
        normalized_focal_loss_per_anchor = focal_sum_per_anchor / (num_positives_per_anchor + epsilon)


        # --- Class-Balanced Anchor Weighting (Small Weights Version) ---
        weight_soz = (1.0 - beta) / (1.0 - torch.pow(beta, N_soz) + epsilon)
        weight_non_soz = (1.0 - beta) / (1.0 - torch.pow(beta, N_non_soz) + epsilon)

        # Create weights tensor based on anchor label
        anchor_weights = torch.where(soz_pat == 1, weight_soz, weight_non_soz)

        # --- Calculate Final Weighted Loss Per Anchor ---
        final_weighted_loss_per_anchor = normalized_focal_loss_per_anchor * anchor_weights

        # --- Aggregate Batch Loss ---
        # Summing the weighted loss terms for all anchors in this sample
        batch_loss += final_weighted_loss_per_anchor.sum()
        total_anchors += num_chan # Count total anchors processed


    # Normalize the total loss by the total number of anchors processed across the batch
    loss = batch_loss / (total_anchors + epsilon)

    return loss
```

A.2.3 Pre-training script

```
import glob
import os
import csv
from data_hup_chul import HUP_CHUL_Dataset, collate_fn
import torch
from tqdm import tqdm
import random
from loss import focal_cb_supcon
from viz import plot_simdist
from model import SOZDetectionPretraining
from scheduler import LinearWarmupCosineAnnealingLR
from torch.utils.data import DataLoader
import torch.optim as optim
from torch.utils.tensorboard import SummaryWriter
import json


EXP_DIRNAME = "HUP_CHUL_13"
SEED = 1975

with open("pretrain_config.json") as fh:
    train_config=json.load(fh)
CLIP_TIME = float(train_config["CLIP_TIME"])
WPT_LEVEL = int(train_config["WPT_LEVEL"])
OVERLAP_RATE = float(train_config["OVERLAP_RATE"])
RELATIVE_NOISE_STD_FACTOR = float(train_config["RELATIVE_NOISE_STD_FACTOR"])
CHANNEL_DROPOUT_RATE = float(train_config["CHANNEL_DROPOUT_RATE"])
BATCH_SIZE = int(train_config["BATCH_SIZE"])
FS = 256
BETA = float(train_config["BETA"])
GAMMA = float(train_config["GAMMA"])
TAU = float(train_config["TAU"])
EPSILON = float(train_config["EPSILON"])

D_FEATURES = int(train_config["D_FEATURES"])
D_MODEL = int(train_config["D_MODEL"])
D_FEEDFORWARD = int(train_config["D_FEEDFORWARD"])
D_HIDDEN = int(train_config["D_HIDDEN"])
MLP_INTERMEDIATE_FACTOR = float(train_config["MLP_INTERMEDIATE_FACTOR"])
NUM_HEADS = int(train_config["NUM_HEADS"])
NUM_LAYERS = int(train_config["NUM_LAYERS"])
DROPOUT = float(train_config["DROPOUT"])
WEIGHT_DECAY = float(train_config["WEIGHT_DECAY"])
```

```
model_pretrain = SOZDetectionPretraining(D_FEATURES, D_MODEL, D_HIDDEN, NUM_HEADS, NUM_LAYERS, D_FEEDFORWARD, DROPOUT,
↪  MLP_INTERMEDIATE_FACTOR)

WARMUP_EPOCHS = train_config["WARMUP_EPOCHS"]
TOTAL_EPOCHS = train_config["TOTAL_EPOCHS"]
START_LR = float(train_config["START_LR"])
MAX_LR = float(train_config["MAX_LR"])
FINAL_LR = float(train_config["FINAL_LR"])
optimizer = optim.Adam(model_pretrain.parameters(), lr=START_LR, weight_decay=WEIGHT_DECAY)
scheduler = LinearWarmupCosineAnnealingLR(optimizer, WARMUP_EPOCHS, MAX_LR, FINAL_LR, TOTAL_EPOCHS)
writer = SummaryWriter(f"exps/{EXP_DIRNAME}/tensorboard")

config_title = (
    # Line 1: Basic Data/Input Params
    f"$SEED={SEED}, T={CLIP_TIME}s, WPT Lvl={WPT_LEVEL}, BS={BATCH_SIZE}, d_{{feat}}={D_FEATURES}$ \n" # Single $ block
    # Line 2: Model Architecture Params - Entire line segment in one $ block
    f"$d_{{model}}={D_MODEL},\ d_{{ff}}={D_FEEDFORWARD},\ d_{{hidden}}={D_HIDDEN},\ "
    f"N_{{h}}={NUM_HEADS},\ N_{{L}}={NUM_LAYERS},\ \\text{{Dropout}}={DROPOUT:.2f}$ \n" # Use \text{} or \mathrm{} for Dropout
    # Line 3: NEW - Overlap & Augmentation Params
    f"Overlap={OVERLAP_RATE*100:.1f}%, Rel. Noise={RELATIVE_NOISE_STD_FACTOR*100:.1f}%, Ch. Dropout={CHANNEL_DROPOUT_RATE*100:.1f}%
↪  \n"
    # Line 4: Loss Function Params
    f"$\\beta={BETA:.3f},\ \\gamma={GAMMA:.2f},\ \\tau={TAU:.2f},\ \\epsilon={EPSILON:.1e}$ \n" # Single $ block
    # Line 5: Optimizer & LR Schedule
    f"{optimizer.__class__.__name__}, LWCA ({WARMUP_EPOCHS}/{TOTAL_EPOCHS}), LR:{START_LR:.1e}  {MAX_LR:.1e}  {FINAL_LR:.1e}"
)

random.seed(SEED)
HUP_PATH = "/projects/users/zrodiere/data/HUP"
CHUL_PATH = "/projects/users/zrodiere/data/CHUL"
SAVE_DIR = "/projects/users/zrodiere/data/save_pkl"
TRAIN_SPLIT = 0.8
VAL_SPLIT = 0.1
TEST_SPLIT = 0.1
NUM_TRAIN_PATIENTS_TO_PLOT = 6

CHUL_PATIENT_IDS = [os.path.basename(patient_dir) for patient_dir in sorted(glob.glob(os.path.join(CHUL_PATH, "sub-CHUL*")))]
HUP_PATIENT_IDS_PRESENT = [os.path.basename(patient_dir) for patient_dir in sorted(glob.glob(os.path.join(HUP_PATH, "sub-HUP*")))]
HUP_SEEG_PATIENT_IDS = []
HUP_PARTICIPANTS_TSV = "/projects/users/zrodiere/data/HUP/participants.tsv"
with open(HUP_PARTICIPANTS_TSV) as fh:
    reader = csv.reader(fh, delimiter="\t")
    column_names = next(reader)
    participant_idx = column_names.index("participant_id")
    implant_idx = column_names.index("implant")
    for row in reader:
        implant = row[implant_idx]
        participant_id = row[participant_idx]
        if implant == "SEEG" and participant_id in HUP_PATIENT_IDS_PRESENT:
            HUP_SEEG_PATIENT_IDS.append(participant_id)
random.shuffle(CHUL_PATIENT_IDS)
random.shuffle(HUP_SEEG_PATIENT_IDS)

NUM_PATIENTS_TRAIN_CHUL = int(TRAIN_SPLIT * len(CHUL_PATIENT_IDS))
NUM_PATIENTS_TRAIN_HUP = int(TRAIN_SPLIT * len(HUP_SEEG_PATIENT_IDS))
NUM_PATIENTS_VAL_CHUL = int((len(CHUL_PATIENT_IDS) - NUM_PATIENTS_TRAIN_CHUL) * (VAL_SPLIT / (VAL_SPLIT + TEST_SPLIT)))
NUM_PATIENTS_VAL_HUP = int((len(HUP_SEEG_PATIENT_IDS) - NUM_PATIENTS_TRAIN_HUP) * (VAL_SPLIT / (VAL_SPLIT + TEST_SPLIT)))
NUM_PATIENTS_TEST_CHUL = len(CHUL_PATIENT_IDS) - NUM_PATIENTS_TRAIN_CHUL - NUM_PATIENTS_VAL_CHUL
NUM_PATIENTS_TEST_HUP = len(HUP_SEEG_PATIENT_IDS) - NUM_PATIENTS_TRAIN_HUP - NUM_PATIENTS_VAL_HUP

TRAIN_PATIENT_IDS = CHUL_PATIENT_IDS[:NUM_PATIENTS_TRAIN_CHUL] + HUP_SEEG_PATIENT_IDS[:NUM_PATIENTS_TRAIN_HUP]
VAL_PATIENT_IDS = CHUL_PATIENT_IDS[NUM_PATIENTS_TRAIN_CHUL:NUM_PATIENTS_TRAIN_CHUL+NUM_PATIENTS_VAL_CHUL] +
↪  HUP_SEEG_PATIENT_IDS[NUM_PATIENTS_TRAIN_HUP:NUM_PATIENTS_TRAIN_HUP+NUM_PATIENTS_VAL_HUP]
TEST_PATIENT_IDS = CHUL_PATIENT_IDS[NUM_PATIENTS_TRAIN_CHUL+NUM_PATIENTS_VAL_CHUL:] +
↪  HUP_SEEG_PATIENT_IDS[NUM_PATIENTS_TRAIN_HUP+NUM_PATIENTS_VAL_HUP:]
TRAIN_PLOT_PATIENT_IDS = random.sample(TRAIN_PATIENT_IDS, NUM_TRAIN_PATIENTS_TO_PLOT)

ds_train = HUP_CHUL_Dataset(HUP_PATH, CHUL_PATH, TRAIN_PATIENT_IDS, SAVE_DIR, CLIP_TIME, WPT_LEVEL, OVERLAP_RATE, "train",
↪  RELATIVE_NOISE_STD_FACTOR, CHANNEL_DROPOUT_RATE)
ds_val = HUP_CHUL_Dataset(HUP_PATH, CHUL_PATH, VAL_PATIENT_IDS, SAVE_DIR, CLIP_TIME, WPT_LEVEL, OVERLAP_RATE, "eval",
↪  RELATIVE_NOISE_STD_FACTOR, CHANNEL_DROPOUT_RATE)
ds_test = HUP_CHUL_Dataset(HUP_PATH, CHUL_PATH, TEST_PATIENT_IDS, SAVE_DIR, CLIP_TIME, WPT_LEVEL, OVERLAP_RATE, "test",
↪  RELATIVE_NOISE_STD_FACTOR, CHANNEL_DROPOUT_RATE)
ds_plot_val = [HUP_CHUL_Dataset(HUP_PATH, CHUL_PATH, [patient_id], SAVE_DIR, CLIP_TIME, WPT_LEVEL, OVERLAP_RATE, "eval",
↪  RELATIVE_NOISE_STD_FACTOR, CHANNEL_DROPOUT_RATE) for patient_id in VAL_PATIENT_IDS]
ds_plot_train = [HUP_CHUL_Dataset(HUP_PATH, CHUL_PATH, [patient_id], SAVE_DIR, CLIP_TIME, WPT_LEVEL, OVERLAP_RATE, "train",
↪  RELATIVE_NOISE_STD_FACTOR, CHANNEL_DROPOUT_RATE) for patient_id in TRAIN_PLOT_PATIENT_IDS]
dl_train = DataLoader(ds_train, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_fn)
dl_val = DataLoader(ds_val, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_fn)
dl_test = DataLoader(ds_test, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_fn)
dl_plot_val = [DataLoader(ds, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_fn) for ds in ds_plot_val]
dl_plot_train = [DataLoader(ds, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_fn) for ds in ds_plot_train]
```

```
device="cuda"
model_pth_best=f"exps/{EXP_DIRNAME}/model_pretrain_best.pth"
model_pth_last=f"exps/{EXP_DIRNAME}/model_pretrain_last.pth"
best_epoch_loss = float('inf')
best_epoch = 0
train_loss_history = []
val_loss_history = []
model_pretrain.to(device)
for epoch in range(TOTAL_EPOCHS):
    model_pretrain.train()
    epoch_train_loss_sum = 0.
    num_train_batches = 0
    for (patient_id,padded_wpt,padded_ch,padded_soz,key_padding_mask) in tqdm(dl_train, desc=f"Epoch {epoch} Train"):
        padded_wpt = padded_wpt.to(device, non_blocking=True)
        padded_soz = padded_soz.to(device, non_blocking=True)
        key_padding_mask = key_padding_mask.to(device, non_blocking=True)

        optimizer.zero_grad()
        h = model_pretrain(padded_wpt, mask=None, src_key_padding_mask=key_padding_mask)
        train_loss = focal_cb_supcon(h, padded_soz, key_padding_mask, beta=BETA, gamma=GAMMA, tau=TAU, epsilon=EPSILON)
        train_loss.backward()
        optimizer.step()
        epoch_train_loss_sum += train_loss.item()
        num_train_batches += 1

    avg_epoch_train_loss = epoch_train_loss_sum / num_train_batches if num_train_batches > 0 else 0.0
    scheduler.step()
    writer.add_scalar("epoch_train_loss", avg_epoch_train_loss, epoch)
    train_loss_history.append(avg_epoch_train_loss)

    model_pretrain.eval()
    epoch_val_loss_sum = 0.
    num_val_batches = 0
    with torch.no_grad():
        for (patient_id,padded_wpt,padded_ch,padded_soz, key_padding_mask) in tqdm(dl_val, desc=f"Epoch {epoch} Val"):
            padded_wpt = padded_wpt.to(device, non_blocking=True)
            padded_soz = padded_soz.to(device, non_blocking=True)
            key_padding_mask = key_padding_mask.to(device, non_blocking=True)
            h = model_pretrain(padded_wpt, mask=None, src_key_padding_mask=key_padding_mask)
            val_loss = focal_cb_supcon(h, padded_soz, key_padding_mask, beta=BETA, gamma=GAMMA, tau=TAU, epsilon=EPSILON)
            epoch_val_loss_sum += val_loss.item()
            num_val_batches += 1

        avg_epoch_val_loss = epoch_val_loss_sum / num_val_batches if num_val_batches > 0 else 0.0
        writer.add_scalar("epoch_val_loss", avg_epoch_val_loss, epoch)
        val_loss_history.append(avg_epoch_val_loss)
        for val_patient_id, dl in zip(VAL_PATIENT_IDS, dl_plot_val):
            plot_simdist(dl=dl,
                         model=model_pretrain,
                         plot_title=f"{val_patient_id} @ Epoch {epoch} (validation set)",
                         plot_config=config_title,
                         png_path=f"exps/{EXP_DIRNAME}/plots/simdist_val_"+str(val_patient_id)+"_ep"+str(epoch).zfill(2)+".png",
                         device=device,
                         train_loss_history=train_loss_history,
                         val_loss_history=val_loss_history,
                         current_epoch=epoch)
        for train_patient_id, dl in zip(TRAIN_PLOT_PATIENT_IDS, dl_plot_train):
            plot_simdist(dl=dl,
                         model=model_pretrain,
                         plot_title=f"{train_patient_id} @ Epoch {epoch} (training set)",
                         plot_config=config_title,
                         png_path=f"exps/{EXP_DIRNAME}/plots/simdist_train_"+str(train_patient_id)+"_ep"+str(epoch).zfill(2)+".png",
                         device=device,
                         train_loss_history=train_loss_history,
                         val_loss_history=val_loss_history,
                         current_epoch=epoch)

    if avg_epoch_val_loss < best_epoch_loss:
        best_epoch_loss = avg_epoch_val_loss
        torch.save(model_pretrain.state_dict(), model_pth_best)
        print(f"Model saved at epoch {epoch} with epoch val loss: {avg_epoch_val_loss:.6f}")
    torch.save(model_pretrain.state_dict(), model_pth_last)

    print(f"epoch {epoch} train loss: {avg_epoch_train_loss:.6f}")
    print(f"epoch {epoch} val loss: {avg_epoch_val_loss:.6f}")
```

A.2.4 Fine-tuning script

```
import glob
import os
```

```
import csv
from data_hup_chul import HUP_CHUL_Dataset, collate_fn
import torch
from tqdm import tqdm
import random
from loss import focal_cb_bce_loss_with_logits, get_precision_micro, get_recall_micro
from viz import plot_roc, plot_scoredist
from model import SOZDetectionPretraining, SOZDetectionFinetuning
from scheduler import LinearWarmupCosineAnnealingLR
from torch.utils.data import DataLoader
import torch.optim as optim
from torch.utils.tensorboard import SummaryWriter
import json


EXP_DIRNAME = "HUP_CHUL_13_FINETUNED_5"
PRETRAIN_PATH = "/projects/users/zrodiere/contrastive_transformer/exps/HUP_CHUL_13/model_pretrain_best.pth"
PREV_FINETUNED_PATH = "/projects/users/zrodiere/contrastive_transformer/exps/HUP_CHUL_13_FINETUNED_4/model_finetuned_best.pth"
SEED = 1975
TRAIN_SPLIT = 0.8
VAL_SPLIT = 0.1
TEST_SPLIT = 0.1
NUM_TRAIN_PATIENTS_TO_PLOT = 6
OVERALL_ROC_PLOT_FREQ = 5
PATIENT_ROC_PLOT_FREQ = 20
HEAD_LR = 5e-7
BACKBONE_LR = 5e-7

print("Fine-tuning SOZ classification model")
with open("finetune_config.json") as fh:
    train_config=json.load(fh)
CLIP_TIME = float(train_config["CLIP_TIME"])
WPT_LEVEL = int(train_config["WPT_LEVEL"])
OVERLAP_RATE = float(train_config["OVERLAP_RATE"])
RELATIVE_NOISE_STD_FACTOR = float(train_config["RELATIVE_NOISE_STD_FACTOR"])
CHANNEL_DROPOUT_RATE = float(train_config["CHANNEL_DROPOUT_RATE"])
BATCH_SIZE = int(train_config["BATCH_SIZE"])
FS = 256
BETA = float(train_config["BETA"])
GAMMA = float(train_config["GAMMA"])

D_FEATURES = int(train_config["D_FEATURES"])
D_MODEL = int(train_config["D_MODEL"])
D_FEEDFORWARD = int(train_config["D_FEEDFORWARD"])
D_HIDDEN = int(train_config["D_HIDDEN"])
MLP_INTERMEDIATE_FACTOR = float(train_config["MLP_INTERMEDIATE_FACTOR"])
NUM_HEADS = int(train_config["NUM_HEADS"])
NUM_LAYERS = int(train_config["NUM_LAYERS"])
DROPOUT = float(train_config["DROPOUT"])
WEIGHT_DECAY = float(train_config["WEIGHT_DECAY"])
model_pretrain = SOZDetectionPretraining(D_FEATURES, D_MODEL, D_HIDDEN, NUM_HEADS, NUM_LAYERS, D_FEEDFORWARD, DROPOUT,
↪  MLP_INTERMEDIATE_FACTOR)
model_pretrain.load_state_dict(torch.load(PRETRAIN_PATH, weights_only=True))
model_finetuned = SOZDetectionFinetuning(model_pretrain)
model_finetuned.load_state_dict(torch.load(PREV_FINETUNED_PATH, weights_only=True))

for param in model_finetuned.pretrained.parameters():
    param.requires_grad = False
print("Pre-trained backbone frozen.")

# Unfreeze specific parts of the backbone
print("Unfreezing all Transformer layers and final norm of the pretrained backbone...")

# Unfreeze all TransformerEncoder layers
unfrozen_backbone_params_list = []
for tr_layer in model_finetuned.pretrained.TransformerEncoder.layers:
    for param in tr_layer.parameters():
        param.requires_grad = True
    unfrozen_backbone_params_list.extend(list(tr_layer.parameters())) # Add their parameters

# Unfreeze the final LayerNorm of the TransformerEncoder
norm_to_unfreeze = model_finetuned.pretrained.TransformerEncoder.norm
for param in norm_to_unfreeze.parameters():
    param.requires_grad = True
unfrozen_backbone_params_list.extend(list(norm_to_unfreeze.parameters()))

# Parameters for the classification head (always trainable in this setup)
head_params_list = list(model_finetuned.classification_head.parameters())

trainable_unfrozen_backbone_params = filter(lambda p: p.requires_grad, unfrozen_backbone_params_list)
trainable_head_params = filter(lambda p: p.requires_grad, head_params_list)

TOTAL_EPOCHS = train_config["TOTAL_EPOCHS"]
```

```
optimizer = optim.Adam([
    {'params': trainable_head_params, 'lr': HEAD_LR}, # Group 1: Head
    {'params': trainable_unfrozen_backbone_params, 'lr': BACKBONE_LR} # Group 2: Unfrozen backbone
], weight_decay=WEIGHT_DECAY)

writer = SummaryWriter(f"exps/{EXP_DIRNAME}/tensorboard")

config_title = (
    # Line 1: Basic Data/Input Params
    f"$SEED={SEED}, T={CLIP_TIME}s, WPT Lvl={WPT_LEVEL}, BS={BATCH_SIZE}, d_{{feat}}={D_FEATURES}$ \n" # Single $ block
    # Line 2: Model Architecture Params - Entire line segment in one $ block
    f"$d_{{model}}={D_MODEL},\ d_{{ff}}={D_FEEDFORWARD},\ d_{{hidden}}={D_HIDDEN},\ "
    f"N_{{h}}={NUM_HEADS},\ N_{{L}}={NUM_LAYERS},\ \\text{{Dropout}}={DROPOUT:.2f}$ \n" # Use \text{} or \mathrm{} for Dropout
    # Line 3: NEW - Overlap & Augmentation Params
    f"Overlap={OVERLAP_RATE*100:.1f}%, Rel. Noise={RELATIVE_NOISE_STD_FACTOR*100:.1f}%, Ch. Dropout={CHANNEL_DROPOUT_RATE*100:.1f}%
    ↪  \n"
    # Line 4: Loss Function Params
    f"$\\beta={BETA:.3f},\ \\gamma={GAMMA:.2f}$ \n" # Single $ block
    # Line 5: Optimizer & LR
    f"{optimizer.__class__.__name__}, Head LR={HEAD_LR:.1e}, Backbone LR={BACKBONE_LR:.1e}, WD={WEIGHT_DECAY:.1e}"
)

random.seed(SEED)
HUP_PATH = "/projects/users/zrodiere/data/HUP"
CHUL_PATH = "/projects/users/zrodiere/data/CHUL"
SAVE_DIR = "/projects/users/zrodiere/data/save_pkl"

CHUL_PATIENT_IDS = [os.path.basename(patient_dir) for patient_dir in sorted(glob.glob(os.path.join(CHUL_PATH, "sub-CHUL*")))]
HUP_PATIENT_IDS_PRESENT = [os.path.basename(patient_dir) for patient_dir in sorted(glob.glob(os.path.join(HUP_PATH, "sub-HUP*")))]
HUP_SEEG_PATIENT_IDS = []
HUP_PARTICIPANTS_TSV = "/projects/users/zrodiere/data/HUP/participants.tsv"
with open(HUP_PARTICIPANTS_TSV) as fh:
    reader = csv.reader(fh, delimiter="\t")
    column_names = next(reader)
    participant_idx = column_names.index("participant_id")
    implant_idx = column_names.index("implant")
    for row in reader:
        implant = row[implant_idx]
        participant_id = row[participant_idx]
        if implant == "SEEG" and participant_id in HUP_PATIENT_IDS_PRESENT:
            HUP_SEEG_PATIENT_IDS.append(participant_id)
random.shuffle(CHUL_PATIENT_IDS)
random.shuffle(HUP_SEEG_PATIENT_IDS)

NUM_PATIENTS_TRAIN_CHUL = int(TRAIN_SPLIT * len(CHUL_PATIENT_IDS))
NUM_PATIENTS_TRAIN_HUP = int(TRAIN_SPLIT * len(HUP_SEEG_PATIENT_IDS))
NUM_PATIENTS_VAL_CHUL = int((len(CHUL_PATIENT_IDS) - NUM_PATIENTS_TRAIN_CHUL) * (VAL_SPLIT / (VAL_SPLIT + TEST_SPLIT)))
NUM_PATIENTS_VAL_HUP = int((len(HUP_SEEG_PATIENT_IDS) - NUM_PATIENTS_TRAIN_HUP) * (VAL_SPLIT / (VAL_SPLIT + TEST_SPLIT)))
NUM_PATIENTS_TEST_CHUL = len(CHUL_PATIENT_IDS) - NUM_PATIENTS_TRAIN_CHUL - NUM_PATIENTS_VAL_CHUL
NUM_PATIENTS_TEST_HUP = len(HUP_SEEG_PATIENT_IDS) - NUM_PATIENTS_TRAIN_HUP - NUM_PATIENTS_VAL_HUP

TRAIN_PATIENT_IDS = CHUL_PATIENT_IDS[:NUM_PATIENTS_TRAIN_CHUL] + HUP_SEEG_PATIENT_IDS[:NUM_PATIENTS_TRAIN_HUP]
VAL_PATIENT_IDS = CHUL_PATIENT_IDS[NUM_PATIENTS_TRAIN_CHUL:NUM_PATIENTS_TRAIN_CHUL+NUM_PATIENTS_VAL_CHUL] +
↪  HUP_SEEG_PATIENT_IDS[NUM_PATIENTS_TRAIN_HUP:NUM_PATIENTS_TRAIN_HUP+NUM_PATIENTS_VAL_HUP]
TEST_PATIENT_IDS = CHUL_PATIENT_IDS[NUM_PATIENTS_TRAIN_CHUL+NUM_PATIENTS_VAL_CHUL:] +
↪  HUP_SEEG_PATIENT_IDS[NUM_PATIENTS_TRAIN_HUP+NUM_PATIENTS_VAL_HUP:]
TRAIN_PLOT_PATIENT_IDS = random.sample(TRAIN_PATIENT_IDS, NUM_TRAIN_PATIENTS_TO_PLOT)

ds_train = HUP_CHUL_Dataset(HUP_PATH, CHUL_PATH, TRAIN_PATIENT_IDS, SAVE_DIR, CLIP_TIME, WPT_LEVEL, OVERLAP_RATE, "train",
↪  RELATIVE_NOISE_STD_FACTOR, CHANNEL_DROPOUT_RATE)
ds_val = HUP_CHUL_Dataset(HUP_PATH, CHUL_PATH, VAL_PATIENT_IDS, SAVE_DIR, CLIP_TIME, WPT_LEVEL, OVERLAP_RATE, "eval",
↪  RELATIVE_NOISE_STD_FACTOR, CHANNEL_DROPOUT_RATE)
ds_test = HUP_CHUL_Dataset(HUP_PATH, CHUL_PATH, TEST_PATIENT_IDS, SAVE_DIR, CLIP_TIME, WPT_LEVEL, OVERLAP_RATE, "test",
↪  RELATIVE_NOISE_STD_FACTOR, CHANNEL_DROPOUT_RATE)
ds_plot_val = [HUP_CHUL_Dataset(HUP_PATH, CHUL_PATH, [patient_id], SAVE_DIR, CLIP_TIME, WPT_LEVEL, OVERLAP_RATE, "eval",
↪  RELATIVE_NOISE_STD_FACTOR, CHANNEL_DROPOUT_RATE) for patient_id in VAL_PATIENT_IDS]
ds_plot_train = [HUP_CHUL_Dataset(HUP_PATH, CHUL_PATH, [patient_id], SAVE_DIR, CLIP_TIME, WPT_LEVEL, OVERLAP_RATE, "train",
↪  RELATIVE_NOISE_STD_FACTOR, CHANNEL_DROPOUT_RATE) for patient_id in TRAIN_PLOT_PATIENT_IDS]
dl_train = DataLoader(ds_train, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_fn)
dl_val = DataLoader(ds_val, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_fn)
dl_test = DataLoader(ds_test, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_fn)
dl_plot_val = [DataLoader(ds, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_fn) for ds in ds_plot_val]
dl_plot_train = [DataLoader(ds, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_fn) for ds in ds_plot_train]


device="cuda"
model_pth_best=f"exps/{EXP_DIRNAME}/model_finetuned_best.pth"
model_pth_last=f"exps/{EXP_DIRNAME}/model_finetuned_last.pth"
best_epoch_loss = float('inf')
best_epoch = 0
train_loss_history = []
val_loss_history = []
model_finetuned.to(device)
```

```
for epoch in range(TOTAL_EPOCHS):
    model_finetuned.train()
    epoch_train_loss_sum = 0.
    num_train_batches = 0
    for (patient_id,padded_wpt,padded_ch,padded_soz,key_padding_mask) in tqdm(dl_train, desc=f"Epoch {epoch} Train"):
        padded_wpt = padded_wpt.to(device, non_blocking=True)
        padded_soz = padded_soz.to(device, non_blocking=True)
        key_padding_mask = key_padding_mask.to(device, non_blocking=True)

        optimizer.zero_grad()
        unbounded_scores = model_finetuned(padded_wpt, mask=None, src_key_padding_mask=key_padding_mask)
        train_loss = focal_cb_bce_loss_with_logits(unbounded_scores, padded_soz, key_padding_mask, beta=BETA, gamma=GAMMA)
        train_loss.backward()
        optimizer.step()
        epoch_train_loss_sum += train_loss.item()
        num_train_batches += 1

    avg_epoch_train_loss = epoch_train_loss_sum / num_train_batches if num_train_batches > 0 else 0.0
    writer.add_scalar("epoch_train_loss", avg_epoch_train_loss, epoch)
    train_loss_history.append(avg_epoch_train_loss)

    model_finetuned.eval()
    epoch_val_loss_sum = 0.
    num_val_batches = 0
    with torch.no_grad():
        for (patient_id,padded_wpt,padded_ch,padded_soz, key_padding_mask) in tqdm(dl_val, desc=f"Epoch {epoch} Val"):
            padded_wpt = padded_wpt.to(device, non_blocking=True)
            padded_soz = padded_soz.to(device, non_blocking=True)
            key_padding_mask = key_padding_mask.to(device, non_blocking=True)
            unbounded_scores = model_finetuned(padded_wpt, mask=None, src_key_padding_mask=key_padding_mask)
            val_loss = focal_cb_bce_loss_with_logits(unbounded_scores, padded_soz, key_padding_mask, beta=BETA, gamma=GAMMA)
            epoch_val_loss_sum += val_loss.item()
            num_val_batches += 1

        avg_epoch_val_loss = epoch_val_loss_sum / num_val_batches if num_val_batches > 0 else 0.0
        writer.add_scalar("epoch_val_loss", avg_epoch_val_loss, epoch)
        val_loss_history.append(avg_epoch_val_loss)

        if epoch % PATIENT_ROC_PLOT_FREQ == (PATIENT_ROC_PLOT_FREQ -1) or epoch == 0 or epoch == TOTAL_EPOCHS - 1:
            for val_patient_id, dl in zip(VAL_PATIENT_IDS, dl_plot_val):
                plot_roc(
                    dl=dl,
                    model=model_finetuned,
                    plot_title=f"ROC Curve - {val_patient_id} @ Epoch {epoch} (validation set)",
                    plot_config=config_title,
                    png_path=f"exps/{EXP_DIRNAME}/plots/roc_val_{val_patient_id}_ep{str(epoch).zfill(2)}.png",
                    device=device,
                    train_loss_history=train_loss_history,
                    val_loss_history=val_loss_history,
                    current_epoch=epoch
                )
                plot_scoredist(
                    dl=dl,
                    model=model_finetuned,
                    plot_title=f"Score distributions - {val_patient_id} @ Epoch {epoch} (validation set)",
                    plot_config=config_title,
                    png_path=f"exps/{EXP_DIRNAME}/plots/scoredist_val_{val_patient_id}_ep{str(epoch).zfill(2)}.png",
                    device=device,
                    train_loss_history=train_loss_history,
                    val_loss_history=val_loss_history,
                    current_epoch=epoch
                )
            for train_patient_id, dl in zip(TRAIN_PATIENT_IDS, dl_plot_train):
                plot_roc(
                    dl=dl,
                    model=model_finetuned,
                    plot_title=f"ROC Curve - {train_patient_id} @ Epoch {epoch} (training set)",
                    plot_config=config_title,
                    png_path=f"exps/{EXP_DIRNAME}/plots/roc_train_{train_patient_id}_ep{str(epoch).zfill(2)}.png",
                    device=device,
                    train_loss_history=train_loss_history,
                    val_loss_history=val_loss_history,
                    current_epoch=epoch
                )
                plot_scoredist(
                    dl=dl,
                    model=model_finetuned,
                    plot_title=f"Score distributions - {train_patient_id} @ Epoch {epoch} (training set)",
                    plot_config=config_title,
                    png_path=f"exps/{EXP_DIRNAME}/plots/scoredist_train_{train_patient_id}_ep{str(epoch).zfill(2)}.png",
                    device=device,
                    train_loss_history=train_loss_history,
                    val_loss_history=val_loss_history,
```

```
                current_epoch=epoch
            )

        if epoch % OVERALL_ROC_PLOT_FREQ == (OVERALL_ROC_PLOT_FREQ -1) or epoch == 0 or epoch == TOTAL_EPOCHS - 1:
            plot_roc(
                dl=dl_val,
                model=model_finetuned,
                plot_title=f"ROC Curve - Validation Set @ Epoch {epoch}",
                plot_config=config_title,
                png_path=f"exps/{EXP_DIRNAME}/plots/roc_val_global_ep{str(epoch).zfill(2)}.png",
                device=device,
                train_loss_history=train_loss_history,
                val_loss_history=val_loss_history,
                current_epoch=epoch
            )
            plot_scoredist(
                dl=dl_val,
                model=model_finetuned,
                plot_title=f"Score distributions - Validation Set @ Epoch {epoch}",
                plot_config=config_title,
                png_path=f"exps/{EXP_DIRNAME}/plots/scoredist_val_global_ep{str(epoch).zfill(2)}.png",
                device=device,
                train_loss_history=train_loss_history,
                val_loss_history=val_loss_history,
                current_epoch=epoch
            )

    if avg_epoch_val_loss < best_epoch_loss:
        best_epoch_loss = avg_epoch_val_loss
        torch.save(model_finetuned.state_dict(), model_pth_best)
        print(f"Model saved at epoch {epoch} with epoch val loss: {avg_epoch_val_loss:.6f}")
    torch.save(model_finetuned.state_dict(), model_pth_last)

    print(f"Epoch {epoch} train loss: {avg_epoch_train_loss:.6f}")
    print(f"Epoch {epoch} val loss: {avg_epoch_val_loss:.6f}")
```

A.3 Visualization

```
import torch
import seaborn as sns
import matplotlib.pyplot as plt
import os
import numpy as np
from sklearn.metrics import roc_curve, auc # For efficient ROC calculation

def plot_simdist(dl, model, plot_title, plot_config, png_path, device, train_loss_history: list, val_loss_history: list,
↪   current_epoch: int):
    os.makedirs(os.path.dirname(png_path), exist_ok=True)
    both_not_soz_sims_all = []
    both_soz_sims_all = []
    one_in_one_out_sims_all = []

    for (patient_id,padded_wpt,padded_ch,padded_soz, key_padding_mask) in dl:
        padded_wpt = padded_wpt.to(device, non_blocking=True)
        padded_soz = padded_soz.to(device, non_blocking=True)
        key_padding_mask = key_padding_mask.to(device, non_blocking=True)
        h = model(padded_wpt, mask=None, src_key_padding_mask=key_padding_mask)
        valid_lengths = (~key_padding_mask).sum(dim=1)
        for i, valid_length in enumerate(valid_lengths):
            h_norm = torch.nn.functional.normalize(h[:valid_length, i, :], p=2, dim=1)
            S = h_norm @ h_norm.T
            soz = padded_soz[i, :valid_length]
            both_not_soz = ~soz.unsqueeze(0) & ~soz.unsqueeze(1)
            both_soz = soz.unsqueeze(0) & soz.unsqueeze(1)
            one_in_one_out = (soz.unsqueeze(0) & ~soz.unsqueeze(1)) | (~soz.unsqueeze(0) & soz.unsqueeze(1))
            remove_diag = ~torch.eye(len(soz), dtype=torch.bool, device=both_not_soz.device)
            both_not_soz_sims_all.append(S[both_not_soz & remove_diag].flatten().detach().cpu())
            both_soz_sims_all.append(S[both_soz & remove_diag].flatten().detach().cpu())
            one_in_one_out_sims_all.append(S[one_in_one_out].flatten().detach().cpu())
        del padded_wpt, padded_soz, key_padding_mask, h
        #torch.cuda.empty_cache()

    both_not_soz_sims_all = torch.cat(both_not_soz_sims_all)
    both_soz_sims_all = torch.cat(both_soz_sims_all)
    one_in_one_out_sims_all = torch.cat(one_in_one_out_sims_all)

    fig, ax = plt.subplots(figsize=(10, 7))
    sns.histplot(both_not_soz_sims_all, stat="density", label="Both not SOZ", alpha=1, color="k", ax=ax, element="step", fill=True)
    sns.histplot(both_soz_sims_all, stat="density", label="Both SOZ", alpha=0.5, color="tab:blue", ax=ax, element="step",
↪   fill=True)
```

```
    sns.histplot(one_in_one_out_sims_all, stat="density", label="One in One out", alpha=0.5, color="tab:orange", ax=ax,
    ↪  element="step", fill=True)
    ax.set_xlabel("Cosine similarity")
    ax.set_ylabel("Density")
    ax.legend(title="Pair groups")
    ax.set_title(plot_title)
    ax.set_xlim(0.6, 1.0)

    info_box = fig.text(
        0.02, 0.98, plot_config,
        transform=fig.transFigure, fontsize=9,
        verticalalignment='top', horizontalalignment='left',
        bbox=dict(boxstyle='round,pad=0.4', fc='wheat', alpha=0.4)
    )

    # --- Add Inset Axes for Loss History ---
    # Define position: [left, bottom, width, height] in figure coordinates (0-1)
    # Adjust these values to place it correctly, avoiding overlap
    inset_left = 0.55
    inset_bottom = 0.8
    inset_width = 0.35
    inset_height = 0.18
    ax_inset = fig.add_axes([inset_left, inset_bottom, inset_width, inset_height])

    # Prepare data for inset plot
    epochs_plotted = np.arange(current_epoch + 1)
    train_losses_plotted = train_loss_history[:current_epoch+1]
    val_losses_plotted = val_loss_history[:current_epoch+1]

    # Plot on inset axes
    ax_inset.plot(epochs_plotted, train_losses_plotted, label='Train Loss', color='tab:orange', marker='.', linestyle='-')
    ax_inset.plot(epochs_plotted, val_losses_plotted, label='Val Loss', color='tab:blue', marker='.', linestyle='-')

    # Customize inset axes
    ax_inset.set_title('Loss History', fontsize=9)
    ax_inset.set_xlabel('Epoch', fontsize=8)
    ax_inset.set_ylabel('Avg Loss', fontsize=8)
    ax_inset.tick_params(axis='both', which='major', labelsize=7)
    ax_inset.grid(True, linestyle='--', alpha=0.6)
    # Optionally set y-limits if loss varies a lot, e.g., ax_inset.set_ylim(bottom=0)
    # Add a legend to the inset if space allows
    ax_inset.set_ylim(0.07, 0.09)
    ax_inset.legend(fontsize=7, loc='best') # Might be cluttered

    # Adjust main plot layout to prevent title overlap etc.
    # Increase top margin if needed
    fig.subplots_adjust(left=0.1, bottom=0.1, right=0.95, top=0.7) # Increased top margin
    plt.savefig(png_path, bbox_inches='tight') # Use bbox_inches='tight'
    plt.close(fig)


def get_all_scores_and_labels(dl, model, device):
    """
    Iterates through the DataLoader to get all unbounded scores and true SOZ labels
    for all valid channels.
    """
    all_scores_list = []
    all_true_labels_list = []
    for (patient_id, padded_wpt, padded_ch, padded_soz, key_padding_mask) in dl:
        padded_wpt = padded_wpt.to(device, non_blocking=True)
        padded_soz_labels = padded_soz.to(device, non_blocking=True) # Keep as original type (bool or int)
        key_padding_mask = key_padding_mask.to(device, non_blocking=True)

        # Model forward pass
        # Assuming model_finetuned structure:
        # unbounded_scores shape (max_seq_len, batch_size, 1)
        unbounded_scores_batch = model(padded_wpt, mask=None, src_key_padding_mask=key_padding_mask)

        valid_lengths = (~key_padding_mask).sum(dim=1)
        for i, valid_length in enumerate(valid_lengths):
            if valid_length == 0: continue

            # Get scores for valid channels for this sample
            # unbounded_scores_batch is (max_channels, batch_size, 1)
            # We need (valid_length,)
            scores_sample_valid = unbounded_scores_batch[:valid_length, i, :].squeeze(-1) # Squeeze the last dim

            # Get true labels for valid channels for this sample
            true_labels_sample_valid = padded_soz_labels[i, :valid_length]

            all_scores_list.append(scores_sample_valid.detach().cpu())
            all_true_labels_list.append(true_labels_sample_valid.detach().cpu())
```

```
    if not all_scores_list: # Handle empty dataloader case
        return torch.empty(0), torch.empty(0)

    # Concatenate all scores and labels from the entire dataset
    all_scores_flat = torch.cat(all_scores_list)
    all_true_labels_flat = torch.cat(all_true_labels_list)

    return all_scores_flat, all_true_labels_flat


def plot_roc(dl, model, plot_title, plot_config, png_path, device, train_loss_history: list, val_loss_history: list, current_epoch:
↪ int):
    os.makedirs(os.path.dirname(png_path), exist_ok=True)

    # 1. Get all scores and true labels
    all_scores, all_true_labels = get_all_scores_and_labels(dl, model, device)

    # Ensure we have data to plot
    if all_scores.numel() == 0 or all_true_labels.numel() == 0:
        print(f"Warning: No data to plot ROC for '{plot_title}'. Skipping.")
        return

    # Convert to NumPy for scikit-learn
    y_true_np = all_true_labels.numpy().astype(int)
    y_scores_np = torch.sigmoid(all_scores).numpy()

    # 2. Calculate ROC curve points and AUC using scikit-learn
    # (handles multiple thresholds automatically and efficiently)
    fpr, tpr, thresholds = roc_curve(y_true_np, y_scores_np)
    roc_auc = auc(fpr, tpr)

    # --- Plotting ---
    fig, ax = plt.subplots(figsize=(10, 7)) # Main plot axes

    # Plot ROC curve
    ax.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC = {roc_auc:.3f})')
    ax.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', label='Chance (AUC = 0.500)') # Diagonal line
    ax.plot(0, 0, marker='o', markersize=15, color='red', label='Always Predict No SOZ')
    ax.plot(1, 1, marker='o', markersize=15, color='purple', label='Always Predict SOZ')

    # Customize main ROC plot
    ax.set_xlim([0.0, 1.0])
    ax.set_ylim([0.0, 1.05]) # Slightly more than 1.0 for better visibility of top curve
    ax.set_xlabel('False Positive Rate')
    ax.set_ylabel('True Positive Rate')
    ax.set_title(plot_title)
    ax.legend(loc="lower right")
    ax.grid(True, linestyle='--', alpha=0.7)

    # Configuration Info Box (same as plot_simdist)
    info_box = fig.text(
        0.02, 0.98, plot_config,
        transform=fig.transFigure, fontsize=9,
        verticalalignment='top', horizontalalignment='left',
        bbox=dict(boxstyle='round,pad=0.4', fc='wheat', alpha=0.4)
    )

    # --- Add Inset Axes for Loss History (same as plot_simdist) ---
    inset_left = 0.55 #0.55  # Adjusted for potentially wider ROC plot
    inset_bottom = 0.78 #0.78 # Adjusted to be lower, ROC curve is main focus
    inset_width = 0.35
    inset_height = 0.18 #0.18
    ax_inset = fig.add_axes([inset_left, inset_bottom, inset_width, inset_height])

    epochs_plotted = np.arange(current_epoch + 1)
    train_losses_plotted = train_loss_history[:current_epoch+1]
    val_losses_plotted = val_loss_history[:current_epoch+1]

    ax_inset.plot(epochs_plotted, train_losses_plotted, label='Train Loss', color='tab:orange', marker='.', linestyle='-')
    ax_inset.plot(epochs_plotted, val_losses_plotted, label='Val Loss', color='tab:blue', marker='.', linestyle='-')

    ax_inset.set_title('Loss History', fontsize=9)
    ax_inset.set_xlabel('Epoch', fontsize=8)
    ax_inset.set_ylabel('Avg Loss', fontsize=8)
    ax_inset.tick_params(axis='both', which='major', labelsize=7)
    ax_inset.grid(True, linestyle='--', alpha=0.6)
    # ax_inset.set_ylim(0.07, 0.09) # Adjust based on your typical loss range
    ax_inset.legend(fontsize=7, loc='best')

    # Adjust main plot layout
    fig.subplots_adjust(left=0.1, bottom=0.1, right=0.95, top=0.7)

    plt.savefig(png_path, bbox_inches='tight')
```

```
    plt.close(fig)

def plot_scoredist(dl, model, plot_title, plot_config, png_path, device, train_loss_history: list, val_loss_history: list,
↪   current_epoch: int):
    os.makedirs(os.path.dirname(png_path), exist_ok=True)

    # 1. Get all scores and true labels
    all_scores, all_true_labels = get_all_scores_and_labels(dl, model, device)

    # Ensure we have data to plot
    if all_scores.numel() == 0 or all_true_labels.numel() == 0:
        print(f"Warning: No data to plot score distribution for '{plot_title}'. Skipping.")
        return

    # Convert to NumPy for scikit-learn
    y_true_np = all_true_labels.numpy().astype(int)
    y_scores_np = torch.sigmoid(all_scores).numpy()

    scores_from_soz = np.array([score for score,label in zip(y_scores_np, y_true_np) if label==1])
    scores_from_non_soz = np.array([score for score,label in zip(y_scores_np, y_true_np) if label==0])

    # --- Plotting ---
    fig, ax = plt.subplots(figsize=(10, 7))
    if scores_from_soz.size > 0:
        sns.histplot(scores_from_soz, stat="density", label="Ground tr. SOZ", alpha=1, color="tab:blue", ax=ax, element="step",
        ↪   fill=True)
    else:
        print(f"Warning: No SOZ samples to plot scores for in '{plot_title}'.")

    if scores_from_non_soz.size > 0:
        sns.histplot(scores_from_non_soz, stat="density", label="Ground tr. not SOZ", alpha=0.5, color="tab:orange", ax=ax,
        ↪   element="step", fill=True)
    else:
        print(f"Warning: No non-SOZ samples to plot scores for in '{plot_title}'.")
    ax.set_xlabel("Score")
    ax.set_ylabel("Density")
    ax.legend(title="Ground truth labels")
    ax.set_title(plot_title)
    ax.set_xlim(0.0, 1.0)

    info_box = fig.text(
        0.02, 0.98, plot_config,
        transform=fig.transFigure, fontsize=9,
        verticalalignment='top', horizontalalignment='left',
        bbox=dict(boxstyle='round,pad=0.4', fc='wheat', alpha=0.4)
    )

    inset_left = 0.55
    inset_bottom = 0.80
    inset_width = 0.35
    inset_height = 0.15
    ax_inset = fig.add_axes([inset_left, inset_bottom, inset_width, inset_height])

    epochs_plotted = np.arange(current_epoch + 1)
    train_losses_plotted = train_loss_history[:current_epoch+1]
    val_losses_plotted = val_loss_history[:current_epoch+1]

    ax_inset.plot(epochs_plotted, train_losses_plotted, label='Train Loss', color='tab:orange', marker='.', linestyle='-')
    ax_inset.plot(epochs_plotted, val_losses_plotted, label='Val Loss', color='tab:blue', marker='.', linestyle='-')

    ax_inset.set_title('Loss History', fontsize=9)
    ax_inset.set_xlabel('Epoch', fontsize=8)
    ax_inset.set_ylabel('Avg Loss', fontsize=8)
    ax_inset.tick_params(axis='both', which='major', labelsize=7)
    ax_inset.grid(True, linestyle='--', alpha=0.6)
    ax_inset.legend(fontsize=7, loc='best')

    # Adjust main plot layout
    fig.subplots_adjust(left=0.1, bottom=0.1, right=0.95, top=0.7) # Ensure top is adjusted for info box
    plt.savefig(png_path, bbox_inches='tight')
    plt.close(fig)
```