System Description for MTC AIC-3 EEG Challenge

Mohammed Ahmed Metwally
CodeCraft Team Leader

June 30, 2025

Contents

1 Introduction

1.1 Meet the Team

Hi! I'm Mohammed Ahmed Abd-al-Azim Metwally — a passionate AI enthusiast and currently a second-year student at the Faculty of Engineering, Mansoura University, majoring in Artificial Intelligence Engineering.

Over the last year, I've been actively diving into machine learning competitions, especially on Kaggle. It's through these competitions that I first encountered the world of brain-computer interfaces (BCI) — something I knew almost nothing about before. This particular challenge has been one of the most eye-opening learning experiences I've had, pushing me to explore neuroscience, signal processing, and robust ML design all at once.

Joining me on this journey are my two amazing teammates: Ahmed Nabil and Mohammed Tareq. I'm deeply grateful for their support and presence throughout the competition.

While the effort was collaborative, most of the technical design, experimentation, and core contributions came from my side — and I'm incredibly excited to share the full system, theory, and motivation behind our solution with you in this paper.

Rather than just showing performance numbers, this paper walks you through the reasoning behind each decision, the challenges we faced, and the techniques we discovered along the way. I hope it gives you both a clear understanding and a genuine glimpse into the thought process behind our submission.

1.2 Paper Overview

Here's what the paper covers:

- **System Overview**
  A high-level view of the complete pipeline and how components interact.
- **MTCFormer Architecture**
  Explanation of our custom model design and its inspiration.
- **Motor Imagery Pipeline**
  Details of the three MI models, including preprocessing, architecture, and training.
- **SSVEP Pipeline**
  Overview of the SSVEP model and the frequency-aware preprocessing used.
- **Inference Logic**
  How predictions are made and combined, with an emphasis on ensembling for MI.
- **Unexplored Ideas and Inspirations**
  We include some underexplored ideas.
- **Challanges we faced along the way**
  We highlight the key obstacles we encountered during development. We also reflect on how these challenges shaped the final design choices and what we would improve with more time.

- **Implementation References** For most sections in this paper, we include a dedicated *Implementation* paragraph that references the exact file paths where the corresponding functionality is implemented in the codebase.

2 System Overview

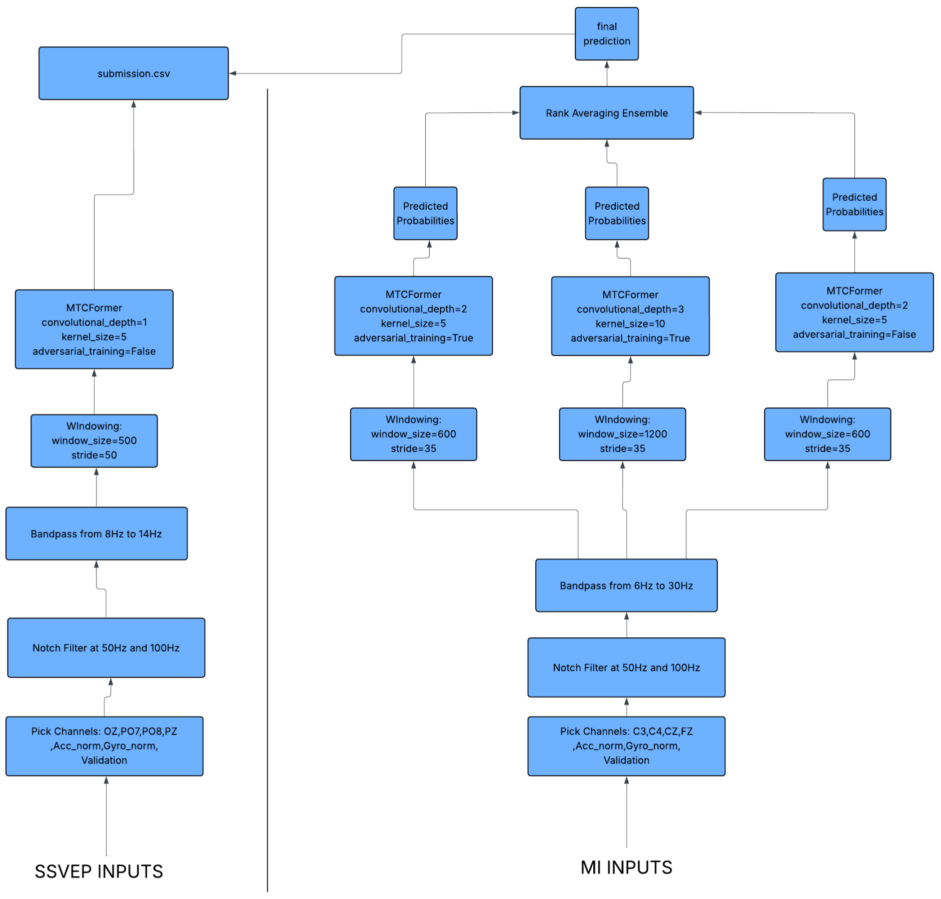
final prediction
submission.csv
Rank Averaging Ensemble
Predicted Probabilities
Predicted Probabilities
Predicted Probabilities
MTCFormer convolutional_depth=1 kernel_size=5 adversarial_training=False
MTCFormer convolutional_depth=2 kernel_size=5 adversarial_training=True
MTCFormer convolutional_depth=3 kernel_size=10 adversarial_training=True
MTCFormer convolutional_depth=2 kernel_size=5 adversarial_training=False
WIndowing: window_size=500 stride=50
WIndowing: window_size=600 stride=35
WIndowing: window_size=1200 stride=35
WIndowing: window_size=600 stride=35
Bandpass from 8Hz to 14Hz
Bandpass from 6Hz to 30Hz
Notch Filter at 50Hz and 100Hz
Notch Filter at 50Hz and 100Hz
Pick Channels: OZ,PO7,PO8,PZ ,Acc_norm,Gyro_norm, Validation
Pick Channels: C3,C4,CZ,FZ ,Acc_norm,Gyro_norm, Validation
SSVEP INPUTS
MI INPUTS

Figure 1: High level overview of the system architecture.

Our solution targets both Motor Imagery (MI) and Steady-State Visual Evoked Potential (SSVEP) classification tasks. Both pipelines are built around variants of a shared architecture — **MTCFormer** — which is a custom-designed architecture that combines convolutional attention and subject conditioning with Temporal modulation. Below, we outline the high-level design of both pipelines.

2.1 Motor Imagery (MI) Pipeline: High-Level Overview

Our MI system is built as an ensemble of three MTCFormer models, each trained with a distinct preprocessing configuration. All three share the same initial preprocessing pipeline:

- Channel selection: **C3, C4, Cz, Fz**, accelerometer norm, gyroscope norm, and the validation signal
- Notch filtering at 50 Hz and 100 Hz (width = 1 Hz)

- Bandpass filtering from 6–30 Hz to focus on mu and beta rhythms

After this shared preprocessing, the signal branches into three separate paths:

Model 1: Short-Window MTCFormer + Adversarial Training

This model applies **windowing**, a sliding window technique that slices the EEG signal into overlapping segments to increase data diversity. Specifically, we use: - Window size: 600 samples - Stride: 35 samples

This setup allows us to generate approximately $\lceil \frac{\text{trial\_length}-600}{35} + 1 \rceil$ training examples from each trial.

The resulting segments are fed into an MTCFormer with: - Convolutional attention depth: 2 - Kernel size: 5 - Trained with **adversarial sample augmentation** to enhance robustness

Model 2: Long-Window MTCFormer + Adversarial Training

This model uses: - Window size: 1200 - Stride: 35

Fewer but longer windows allow it to focus on extended temporal patterns. It uses a deeper MTCFormer (depth = 3), and is also trained adversarially.

Model 3: Short-Window MTCFormer (No Adversarial)

This model mirrors the first one with: - Window size: 600 - Stride: 35 - Depth: 2, kernel size: 5 but is trained **without adversarial augmentation**, providing diversity to the ensemble via a more conventional training setup.

Ensembling Strategy The three models output class probabilities which are then aggregated using a **ranking-average ensembling strategy**. This method preserves prediction order across models and is especially useful when probability calibration differs between learners. It boosts stability and leverages complementary insights from models trained at different window scales and robustness levels.

2.2 SSVEP Pipeline: High-Level Overview

For the SSVEP task, we use a single MTCFormer model trained with a frequency-targeted preprocessing pipeline. The preprocessing consists of:

- Notch filtering at 50 Hz and 100 Hz (width = 1 Hz)
- Bandpass filtering from 8–14 Hz to isolate stimulus-locked frequency responses
- Windowing with a window size of 500 samples and stride of 50

This input is passed to a lightweight MTCFormer with: - Convolutional attention depth: 1 - No adversarial training

The model is tuned to detect steady-state visual responses efficiently and serves as the dedicated branch for the SSVEP classification task.

3 MTCFormer Architecture

3.1 Overview

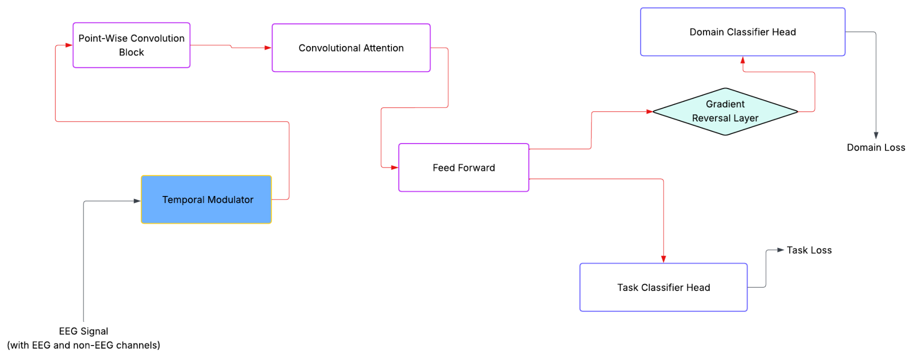

Figure 2: MTCFormer architecture.

As shown in Figure 2, MTCFormer is composed of seven key components, each designed for a different purpose:

- **Temporal Modulator** – Integrates non-EEG signals (e.g., accelerometer, gyroscope) to modulate the raw EEG input dynamically.
- **Pointwise Convolution** – A 1D convolution with kernel size 1, used to project the input into a higher-dimensional feature space while preserving temporal resolution.
- **Convolutional Attention Block** – Inspired by SSVEPFormer, this replaces traditional attention with local convolutions that model temporal dependencies efficiently.
- **Feed-Forward Network** – A small MLP that further transforms the learned representations.
- **Task Classifier Head** – Predicts the output class (e.g., left hand, right hand, etc.) based on extracted features.
- **Optional Domain Classifier Head** – Used for adversarial training to promote subject-invariant features. It includes a Gradient Reversal Layer (GRL) where setting $\lambda = 0$ disables its effect, and $\lambda > 0$ reverses gradients to confuse the domain classifier.

Each of these blocks, along with the design choices behind them, will be described in more detail in the following subsections.

3.2 Temporal Modulator

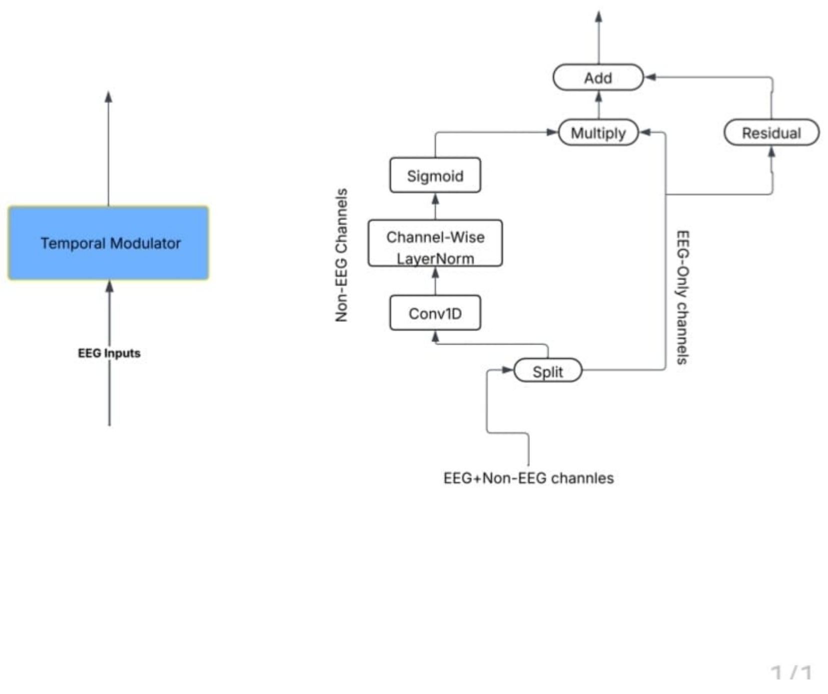

Figure 3: Temporal modulator design.

The Temporal Modulator is the first block in MTCFormer. Its goal is to let the model use non-EEG signals — like the accelerometer norm, gyroscope norm, and validation channel — to help interpret and possibly clean the EEG data early in the network.

We start by splitting the input into two separate branches:

- $\mathbf{X}_{\text{EEG}} \in \mathbb{R}^{C_e \times T}$: the EEG channels
- $\mathbf{X}_{\text{aux}} \in \mathbb{R}^{C_a \times T}$: the auxiliary (non-EEG) features

where C_e is the number of EEG channels, C_a is the number of auxiliary channels, and T is the number of time steps (samples).

The auxiliary branch is passed through a 1D convolution, followed by channel-wise layer normalization, and finally a sigmoid activation:

$$\mathbf{M} = \sigma(\text{LayerNorm}(\text{Conv1D}(\mathbf{X}_{\text{aux}}))) \in \mathbb{R}^{C_e \times T}$$

Here, $\mathbf{M}$ is the modulation signal, and the sigmoid function $\sigma(\cdot)$ ensures that each modulation weight lies in the range $[0, 1]$. We match its shape to the EEG stream via appropriate projection.

These weights are then used to scale the EEG signal elementwise:

$$\mathbf{X}_{\text{mod}} = \mathbf{X}_{\text{EEG}} \odot \mathbf{M}$$

To preserve the ability to fall back to the unmodulated EEG and improve training stability, we add a residual connection:

$$\mathbf{Y} = \mathbf{X}_{\text{EEG}} + \mathbf{X}_{\text{mod}} \in \mathbb{R}^{C_e \times T}$$

This way, the model learns how to enhance or suppress different parts of the EEG based on the auxiliary signals — but it can still rely on the raw EEG if modulation isn't useful.

In short, the Temporal Modulator gives the network flexibility: it can use auxiliary context to filter or amplify signals, but the residual path ensures the core EEG information is always preserved.

3.3 Pointwise Convolution

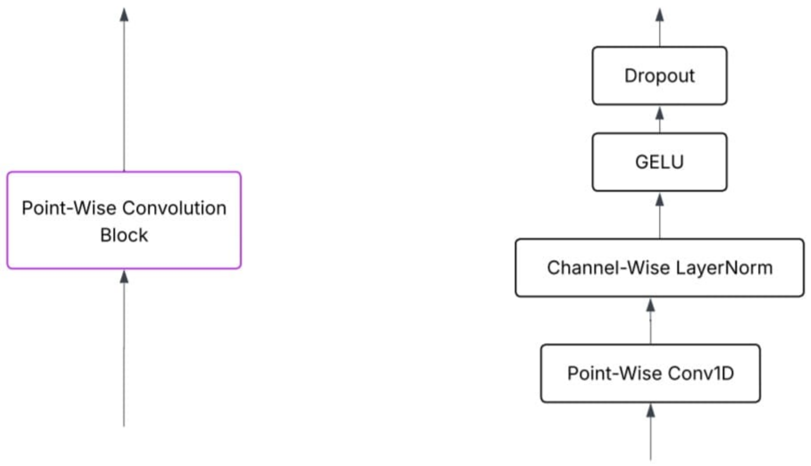

Figure 4: Pointwise Convolution Block design.

The Pointwise Convolution block takes the output of the Temporal Modulator and projects it into a higher-dimensional space. This helps the model expand its feature capacity before applying Convolutional Attention.

We use a 1D convolution with kernel size 1 — commonly known as a pointwise convolution

— which applies a separate linear transformation at each time step, independently across time. This preserves the temporal resolution while transforming each feature vector across channels.

Let the input to this block be:

$$\mathbf{X} \in \mathbb{R}^{C \times T}$$

where C is the number of EEG channels and T is the number of time steps.

The block performs the following operations in sequence:

- **Conv1D (kernel size = 1)** to increase channel dimension:

$$\mathbf{X}' = \text{Conv1D}_{1\times1}(\mathbf{X}) \in \mathbb{R}^{2C \times T}$$

- **Channel-wise Layer Normalization** to stabilize the activation distribution
- **GELU (Gaussian Error Linear Unit)** activation for smoother, noise-tolerant nonlinearity
- **Dropout** to regularize and prevent overfitting

The final output shape of this block is:

$$\mathbf{Y} \in \mathbb{R}^{2C \times T}$$

Using a pointwise convolution here is intentional — it enables the model to increase feature richness without mixing temporal information, preserving the alignment across time steps while expanding the channel dimension for deeper processing in the next stage.

3.4 Convolutional Attention and Feed-Forward Network

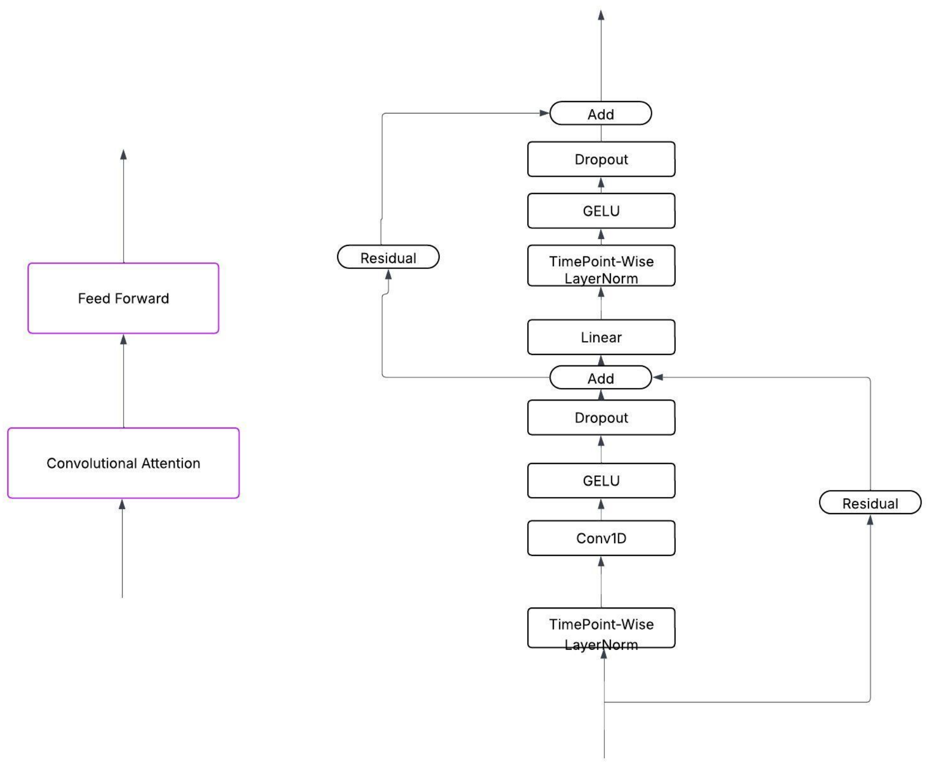

Figure 5: Convolutional Attention Block design.

This block forms the core feature extractor in MTCFormer. It's designed to replace traditional self-attention with a lightweight, local alternative — one that can still capture temporal structure efficiently but with fewer parameters and better inductive bias for EEG signals.

Convolutional Attention Block This part begins with a time-wise layer normalization, followed by a regular 1D convolution. This helps the model capture local temporal patterns:

- **Time-wise LayerNorm**
- **Conv1D**
- **GELU** activation
- **Dropout**
- **Residual Add**

This residual connection is critical — it allows deeper models to be trained stably, especially when stacking multiple blocks. Without it, gradients may vanish or explode, and the model would struggle to refine features over depth.

Feed-Forward Block (Over Time) Immediately after the convolutional attention, we add a feed-forward network, but unlike traditional MLPs applied across channels, ours is applied *through* *time.* That means each time step is processed independently with a small MLP, simulating the behavior of token-wise attention layers.

This design helps the model refine or reweight each temporal feature (using all other temporal features) independently after the temporal convolution. Although, We were concerned about the extent to which this transformation might **distort the temporal structure** of the signal, particularly in ways that could degrade the effectiveness of later **convolutional blocks**. As a result, the maximum depth we used for this **convolution-attention block** was limited to **3**. The structure is:

- **Time-wise LayerNorm**
- **GELU** activation
- **Dropout**
- **Residual Add**

Summary Together, this Convolutional Attention + Feed-Forward block acts as the primary feature extractor in MTCFormer. It captures both short-term patterns and allows nonlinear transformation at each time step. Residuals between each sub-block are key to enabling deeper stacking and stable training.

This overall block design is primarily inspired by the SSVEPFormer architecture, which demonstrated the effectiveness of using local convolutions and token-wise processing as a lightweight alternative to traditional self-attention.

3.5 Task and Domain Classifier Heads

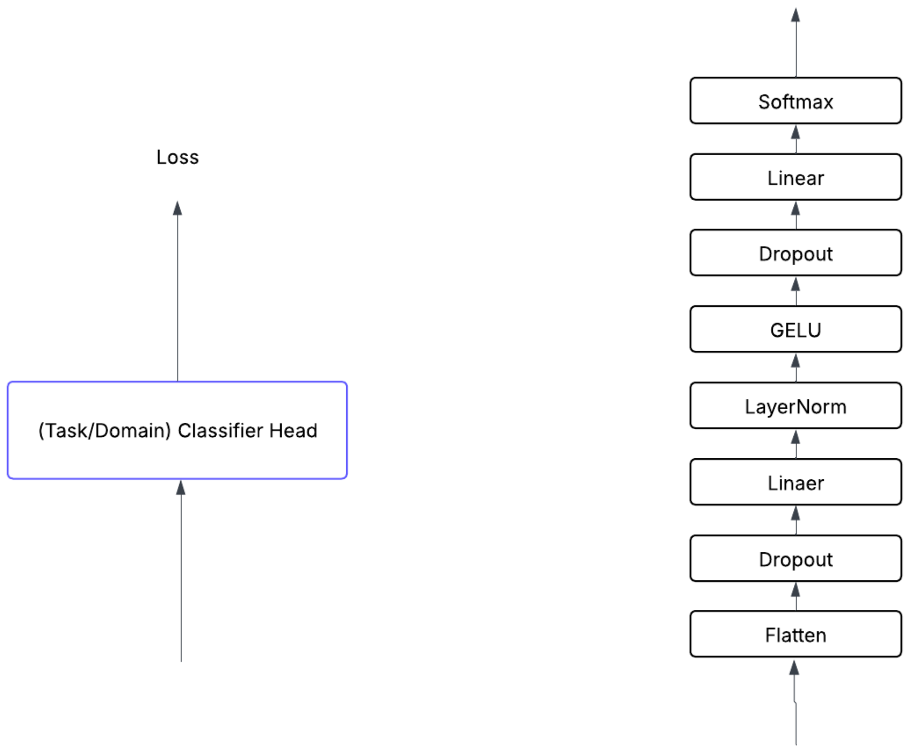

Figure 6: Classification head design.

The MTCFormer ends with two parallel classifier heads: one for the main task and one for domain (subject) prediction. Both heads share the same architecture but serve different objectives.

Task Classifier The task classifier is used to predict the label of the EEG task (e.g., motor imagery class) based on the learned features. It is trained using standard cross-entropy loss against the ground-truth task labels.

Domain Classifier with Gradient Reversal The domain classifier is trained to predict the subject ID of the input sample. Its purpose, however, is not to improve subject prediction accuracy. Instead, it's used adversarially to encourage the shared feature extractor to learn representations that are invariant to subject identity.

To achieve this, we insert a *Gradient Reversal Layer (GRL)* between the feature extractor (earlier layers in the network) and the domain classifier. During the forward pass, GRL does nothing — it passes its input unchanged:

$$\mathrm{GRL}(x) = x$$

But during the backward pass, it multiplies the gradients flowing through it by a negative constant $-\lambda$:

$$\frac{\partial \mathcal{L}_{\text{domain}}}{\partial x} \rightarrow -\lambda \cdot \frac{\partial \mathcal{L}_{\text{domain}}}{\partial x}$$

This has the effect of *fooling* the domain classifier — the feature extractor (earlier layers in the network) is/are updated in a direction that minimizes the task loss but maximally confuses the domain classifier. This encourages the learned features to be subject-invariant (ignoring subject specific noise and patterns).

Controlling λ The value of λ controls the strength of the adversarial signal:

- If $\lambda = 0$, the GRL has no effect and the domain classifier is effectively detached.
- If $\lambda > 0$, the gradient is flipped and scaled, injecting adversarial feedback into the shared layers.

In practice, we choose λ based on validation performance — often increasing it gradually over training (a strategy explored in prior domain-adversarial work).

Head Architecture Both classifier heads (task and domain) use the same architecture:

- **Flatten**
- **Dropout**
- **Linear layer**
- **LayerNorm**
- **GELU activation**
- **Dropout**
- **Final Linear layer**
- **Softmax**

Summary By combining the task classifier with an adversarial domain classifier, we train the model to focus on features relevant to the EEG task while discarding subject-specific variations — helping generalize across unseen subjects.

Implementation The complete model architecture—including the domain classifier, Gradient Reversal Layer, and other components—is primarily implemented in `model/MTCformerV3.py`.

4 Motor Imagery (MI) Pipeline

4.1 Overview

Before diving into the details of the models used for the Motor Imagery task, it's important to first explain a key training technique that played a major role in improving performance: adversarial training using gradient-based attacks. This method, which we applied during training, helped the models generalize better across subjects — and we'll break it down first.

After that, we'll walk through each of the three models we used in our MI ensemble. For every model, we'll cover the specific preprocessing steps, architecture choices, and training strategies.

Finally, we'll explain how we combined predictions from these models using an ensembling strategy designed to balance diversity and robustness.

4.2 Adversarial Training

4.2.1 Adversarial Training Overview

Adversarial training is a technique where a model is trained not only on clean data but also on purposefully perturbed inputs (adversarial examples) designed to fool it. This forces the model to learn more robust and generalizable representations.

While adversarial training is common in image and NLP tasks, it is still relatively uncommon in EEG pipelines, especially outside of domain adaptation settings. In our work, we explore its use in Motor Imagery to improve subject generalization and robustness against noise.

4.2.2 Adversarial Training with Gradient-Based Attack

To improve the model's robustness, we apply adversarial training — a technique where we generate slightly "harder" versions of the input data to help the model learn more general features. These adversarial examples are created using gradient-based perturbations, inspired by the Projected Gradient Descent (PGD) method.

Why Adversarial Training? Instead of adding random noise as in traditional data augmentation pipelines, we use the model itself to find the most confusing directions in the input space. By training on these difficult examples, the model becomes less sensitive to small shifts or noise — which is important in EEG signals, where variation across subjects is a major challenge.

How the Attack Works Let: - x be a single EEG input (shape: channels $\times$ time) - y be its label - $f(x)$ be the model's prediction - $\mathcal{L}_{\text{task}}$ be the loss (cross-entropy)

We start with the clean input:

$$x_{\text{adv}}^{(0)} = x$$

Then, for $k = 1$ to K (number of attack steps), we repeat:

$$g^{(k)} = \nabla_x \mathcal{L}_{\text{task}}(f(x_{\text{adv}}^{(k-1)}), y)$$

$$x_{\text{adv}}^{(k)} = x_{\text{adv}}^{(k-1)} + \alpha \cdot \text{sign}(g^{(k)})$$

We clip the adversarial sample to stay close to the original input (within an ϵ limit):

$$x_{\text{adv}}^{(k)} = \text{clip}(x_{\text{adv}}^{(k)}, x - \epsilon, x + \epsilon)$$

Finally, we clamp the values to keep them inside a valid EEG range (e.g., $[-16.8, 16.8]$):

$$x_{\text{adv}}^{(k)} = \text{clamp}(x_{\text{adv}}^{(k)}, \text{min\_val}, \text{max\_val})$$

Loss During Training During training, we pass both the original and the adversarial examples through the model. The final loss is:

$$\text{Total Loss} = \text{TaskLoss}(x) + \lambda_{\text{adv}} \cdot \text{TaskLoss}(x_{\text{adv}}) + \text{DomainLoss}$$

Here: - λ_{adv} controls how much the adversarial examples influence training - 'TaskLoss' is the cross-entropy loss for the task classifier - 'DomainLoss' comes from the domain (subject) classifier

Parameters Used - α: size of each attack step - ϵ: maximum allowed perturbation - K: number of steps - λ_{adv}: weight for the adversarial loss - min/max: input range clamp limits (e.g., -16.8 to $+16.8$.)

Implementation The adversarial gradient attack method is implemented in `utils/gradient_attack.py`. It is imported and used directly within the training loop defined in `utils/training.py`, allowing seamless integration of adversarial training during model optimization.

Summary This form of adversarial training acts as a targeted data augmentation method. It forces the model to be stable under worst-case noise directions — leading to better generalization across subjects and cleaner decision boundaries

4.2.3 Challenges We Faced

If we were to list every challenge we faced while tuning adversarial training, this section alone could take the whole paper — and possibly the rest of the day too. That said, here are some of the core difficulties we encountered, especially when applying this technique to EEG:

- **Extreme sensitivity to hyperparameters:** Stabilizing adversarial training in our architecture proved to be very delicate. Small changes to α, ϵ, or the number of steps could lead to drastically different behaviors — often breaking training entirely.
- **Step size vs. number of steps:** We noticed that using more attack steps (K) with a smaller step size (α) made the adversarial examples more subtle and harder to classify — which was ideal in theory, but often too difficult for the model to learn from in practice. This caused training to stall and convergence to drop significantly.
- **Clipping and clamping range:** Choosing the clipping range (around the original signal) required careful tuning. If the allowed perturbation (ϵ) was too small, the adversarial example became almost indistinguishable from the original — essentially training the model on the same input twice. On the other hand, too large a range would produce overly distorted inputs that the model could easily separate from the real data — leading to overfitting to artificial signals.
- **Longer attacks slowed training:** Using a higher number of steps did improve robustness, but it came at the cost of significantly slower training time. Since adversarial samples are computed on-the-fly for each batch, this overhead became a bottleneck, especially for larger models.

In the end, finding the right balance for all of these parameters required a lot of trial and error — and many training runs that didn't go quite as planned. But once tuned properly, the adversarial training did offer noticeable benefits in generalization and robustness.

4.3 Preprocessing (common to all MI models)

Each Motor Imagery (MI) model begins with the same core preprocessing steps:

- **Channel Selection:** We use the four most informative EEG channels for motor tasks: C3, C4, Cz, and Fz.
- **Auxiliary Sensor Norms:** In addition to EEG, we include the norm of the accelerometer and gyroscope signals. Given raw axis readings Acc_x, Acc_y, and Acc_z, we compute the L2 norm as:

  $$\text{Acc}_{\text{norm}} = \sqrt{\text{Acc}_x^2 + \text{Acc}_y^2 + \text{Acc}_z^2}$$

  and similarly for the gyroscope:

  $$\text{Gyro}_{\text{norm}} = \sqrt{\text{Gyro}_x^2 + \text{Gyro}_y^2 + \text{Gyro}_z^2}$$

  The individual x, y, and z channels are then discarded, and only the norm is retained.
- **Validation Signal:** We also include the validation flag signal as an additional input feature.

  We apply IIR notch filters at 50 and 100. to suppress powerline interference and its first harmonic.
- **Bandpass Filtering:** A zero-phase FIR bandpass filter is applied from 6 to 30. Using a linear-phase FIR filter avoids any phase distortion and preserves waveform shape .

Implementation All preprocessing-related utilities are defined in `utils/preprocessing.py`, although some parts—such as computing the L2 norms of the accelerometer and gyroscope signals—are handled directly in `convert_csv_to_fif.py`.

4.4 MI Model 1

4.4.1 Data Segmentation and normalization

After applying the shared preprocessing steps, we perform data segmentation using a sliding-window approach. This allows us to extract multiple overlapping snapshots from each MI trial, effectively increasing the number of training examples.

For Model 1, we use:

- **Window size:** 600 samples
- **Stride:** 35 samples

Given that each trial is 2250 samples long, the number of extracted windows per trial is:

$$\left\lfloor \frac{2250 - 600}{35} \right\rfloor + 1 = 48$$

This results in 48 segments per trial — providing significantly more data to train on, and allowing the model to learn temporal patterns from multiple perspectives within each trial.

Each segment is then **normalized** to have zero mean and unit variance using **Z-score normalization**.

4.4.2 Data Augmentation and Training

To improve generalization, we applied two types of data augmentation:

- **Heuristic augmentation — Additive Gaussian noise:** Each input segment is perturbed by zero-mean Gaussian noise:

  $$\tilde{x} = x + \mathcal{N}(0, \sigma^2)$$

  where $\sigma = 0.1$. This acts as a simple form of regularization and encourages robustness to natural signal variation.

- **Model-aware augmentation — Adversarial attack:** We also apply a single-step adversarial perturbation using Projected Gradient Descent (PGD), as detailed in Section 4.2. The following configurations were used:

  - `adversarial_steps` = 1: Number of PGD iterations (single-step FGSM-style attack)
  - `adversarial_alpha` = 0.005: Step size for each attack step
  - `adversarial_epsilon` = 0.05: Maximum allowed perturbation (clipping range)

  This augmentation forces the model to become resilient to worst-case perturbations in the input space.

- **Domain Adversarial Training:** We also include domain-invariant learning using a Gradient Reversal Layer (GRL). The strength of this domain regularization is controlled by:

  `domain_lambda` = 0.01

  This coefficient scales the reversed gradient during backpropagation to suppress subject-specific features, as described in Section 3.5. We kept `lambda_scheduler_fn = None`, meaning λ remains constant throughout training.

- **Architecture configuration:** `depth = 2`, `kernel_size = 5`, `n_times = 600`, `chs_num = 7` (4 EEG + 3 auxiliary), `class_num = 2`, `class_num_domain = 30`, `modulator_dropout = 0.3`, `mid_dropout = 0.5`, `output_dropout = 0.5`, `weight_init_mean = 0`, `weight_init_std = 0.5`

- **Optimizer:** Adam optimizer with learning rate = 0.002

- **Loss Calculation:**

- **Loss Calculation:** Since each trial is split into multiple overlapping windows, we compute a loss for each window, then aggregate them using a reliability-based weighting scheme.

  For each window i with input x_i and label y, we compute a weighted classification loss:

  $$\mathcal{L}_i^{\text{cls}} = w_i \cdot \text{CE}(f(x_i), y)$$

where: - $f(x_i)$ is the model's predicted logit for window i - $\mathrm{CE}(\cdot,\cdot)$ is the standard cross-entropy loss - $w_i \in [0,1]$ is a reliability weight derived from the `validation` signal within that window

We also include:
- A **domain loss** $\mathcal{L}^{\mathrm{dom}}$, computed across all $\sim$35 subjects (domains). A domain classifier is trained to predict the subject ID from extracted features, while the feature extractor is trained adversarially to prevent it. The loss is computed using multi-class cross-entropy over domain labels.
- An **adversarial sample loss** $\mathcal{L}^{\mathrm{adv}}$, computed on perturbed inputs $\tilde{x}_j$:

$$\mathcal{L}^{\mathrm{adv}} = \frac{1}{N'} \sum_{j=1}^{N'} \mathrm{CE}(f(\tilde{x}_j), y)$$

and weighted by a hyperparameter λ_{adv}

Finally, the total loss for a batch of N windows is:

$$\mathcal{L}_{\mathrm{total}} = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_i^{\mathrm{cls}} + \mathcal{L}^{\mathrm{dom}} + \lambda_{\mathrm{adv}} \cdot \mathcal{L}^{\mathrm{adv}}$$

In our implementation, the adversarial loss weight was set to $\lambda_{\mathrm{adv}} = 0.5$.

- **Learning Rate Scheduler:** MultiStepLR with a decay factor $\gamma = 0.1$ applied at epoch 70
- **Implementation** This model is fully implemented in `train/train_mi_model1.py`, where all preprocessing and training steps are handled end-to-end.

4.5 MI Model 2

4.5.1 Data Segmentation and normalization

As with Model 1, we begin by applying the shared preprocessing steps described earlier. We then use a sliding-window strategy to extract longer snapshots from each MI trial, allowing the model to focus on broader temporal context.

For Model 2, we use:

- **Window size:** 1200 samples
- **Stride:** 35 samples

Given that each trial is 2250 samples long, the number of extracted windows per trial is:

$$\left\lfloor \frac{2250 - 1200}{35} \right\rfloor + 1 = 31$$

This results in 31 segments per trial — fewer than Model 1, but each segment captures more of the trial's temporal structure.

Each segment is then **normalized** to have zero mean and unit variance using **Z-score normalization**.

4.5.2 Data Augmentation and Training

We apply the same augmentation and loss strategy as Model 1:

- **Heuristic augmentation — Additive Gaussian noise:** Each input segment is perturbed by zero-mean Gaussian noise:

$$\tilde{x} = x + \mathcal{N}(0, \sigma^2), \quad \sigma = 0.1$$

- **Model-aware augmentation — Adversarial attack:** We use single-step PGD with the following configuration:
  - `adversarial_steps` = 1
  - `adversarial_alpha` = 0.01
  - `adversarial_epsilon` = 0.01

  This matches the technique and intent used in Model 1 (see Section 4.2).

- **Domain Adversarial Training:** Also identical to Model 1, using a constant domain adversarial coefficient:

$$\texttt{domain\_lambda} = 0.01$$

  to encourage subject-invariant representations (see Section 3.5).

- **Architecture configuration:** `depth = 3`, `kernel_size = 10`, `n_times = 1200`, `chs_num = 7` (4 EEG + 3 auxiliary), `class_num = 2`, `class_num_domain = 30`, `modulator_dropout = 0.3`, `mid_dropout = 0.5`, `output_dropout = 0.5`, `weight_init_mean = 0`, `weight_init_std = 0.5`

- **Optimizer:** Adam optimizer with learning rate = 0.002

- **Loss Calculation:** Since each trial is split into multiple overlapping windows, we compute a loss for each window, then aggregate them using a reliability-based weighting scheme.

  For each window i with input x_i and label y, we compute a weighted classification loss:

$$\mathcal{L}_i^{\text{cls}} = w_i \cdot \text{CE}(f(x_i), y)$$

  where: - $f(x_i)$ is the model's predicted logit for window i - $\text{CE}(\cdot, \cdot)$ is the standard cross-entropy loss - $w_i \in [0, 1]$ is a reliability weight derived from the `validation` signal within that window

  We also include:
  - A **domain loss** $\mathcal{L}^{\text{dom}}$, computed across all ∼35 subjects (domains). A domain classifier is trained to predict the subject ID from extracted features, while the feature extractor is trained adversarially to prevent it. The loss is computed using multi-class cross-entropy over domain labels.
  - An **adversarial sample loss** $\mathcal{L}^{\text{adv}}$, computed on perturbed inputs $\tilde{x}_j$:

$$\mathcal{L}^{\text{adv}} = \frac{1}{N'} \sum_{j=1}^{N'} \text{CE}(f(\tilde{x}_j), y)$$

and weighted by a hyperparameter λ_{adv}

Finally, the total loss for a batch of N windows is:

$$\mathcal{L}_{\text{total}} = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}_i^{\text{cls}} + \mathcal{L}^{\text{dom}} + \lambda_{\text{adv}} \cdot \mathcal{L}^{\text{adv}}$$

In our implementation, the adversarial loss weight was set to $\lambda_{\text{adv}} = 0.5$.

- **Implementation** This model is fully implemented in `train/train_mi_model2.py`, where all preprocessing and training steps are handled end-to-end.

4.6 MI Model 3

4.6.1 Data Segmentation and normalization

This model uses the exact same preprocessing pipeline and windowing configuration as Model 1. We apply a sliding window across each trial to extract short, overlapping segments that enrich the training data.

- **Window size:** 600 samples
- **Stride:** 35 samples

The number of windows per trial is:

$$\left\lfloor \frac{2250 - 600}{35} \right\rfloor + 1 = 48$$

which yields 48 overlapping training examples (segments) per trial.

Each segment is then **normalized** to have zero mean and unit variance using **Z-score normalization**.

4.6.2 Data Augmentation and Training

We reuse the same general augmentation and loss strategy used in the previous models, with one key distinction — adversarial training is disabled in this model.

- **Heuristic augmentation — Additive Gaussian noise:** We apply the same zero-mean Gaussian noise to each segment:

  $$\tilde{x} = x + \mathcal{N}(0, \sigma^2), \quad \sigma = 0.1$$

  to improve robustness to random fluctuations in the signal.
- **Model-aware augmentation — Not used:** Unlike Models 1 and 2, we do not apply adversarial training in this model (`adversarial_training = False`). This gives the model a cleaner training signal and provides diversity to the ensemble.

- **Domain Adversarial Training:** Still applied in the same way as previous models using:

  $$\texttt{domain\_lambda} = 0.01$$

  and no scheduling function, ensuring constant adversarial strength from the domain classifier (see Section 3.5).
- **Architecture configuration:** `depth = 2`, `kernel_size = 5`, `n_times = 600`, `chs_num = 7` (4 EEG + 3 auxiliary), `class_num = 2`, `class_num_domain = 30`, `modulator_dropout = 0.3`, `mid_dropout = 0.5`, `output_dropout = 0.5`, `weight_init_mean = 0`, `weight_init_std = 0.5`
- **Optimizer:** Adam optimizer with learning rate = 0.002
- **Loss Calculation:** Same as in Models 1 and 2, we use a reliability-weighted cross-entropy task loss:

  $$\mathcal{L}_i^{\text{task}} = w_i \cdot \text{CE}(f(x_i), y) \quad \Rightarrow \quad \mathcal{L}^{\text{task}} = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}_i^{\text{task}}$$

  where w_i is derived from the average `validation` signal in each window.

  Unlike Models 1 and 2, this model does not include an adversarial training loss. However, it incorporates a domain loss term $\mathcal{L}^{\text{dom}}$, computed over all 35 subjects to encourage domain-invariant representations.

  The final total loss is:

  $$\mathcal{L}_{\text{total}} = \mathcal{L}^{\text{task}} + \mathcal{L}^{\text{dom}}$$
- **Learning Rate Scheduler:** MultiStepLR with decay factor $\gamma = 0.1$ at epoch 70
- **Implementation** This model is fully implemented in `train/train_mi_model3.py`, where all preprocessing and training steps are handled end-to-end.

4.7 Inference Technique

Since each MI trial is segmented into overlapping windows, the model produces a separate prediction for each window. To derive a single trial-level prediction, we perform a weighted averaging of the window-level predictions based on their signal quality.

- Each window is passed independently through the model, producing a probability distribution over task classes.
- We assign a weight w_i to each window based on the mean of its associated `validation` signal, reflecting signal reliability. Clean windows (higher `validation`) receive larger weights, while noisier ones receive smaller weights.
- The final class probability vector for the trial is computed as a weighted mean:

  $$p_{\text{trial}} = \frac{\sum_i w_i \cdot p_i}{\sum_i w_i}$$

  where p_i is the predicted probability vector for window i.

  - In rare cases where all $w_i \approx 0$ (i.e., all windows are severely corrupted), we fall back to a simple unweighted average over all window predictions:

$$p_{\text{trial}} = \frac{1}{N} \sum_{i=1}^{N} p_i$$

This strategy ensures that the model prioritizes cleaner parts of the trial during inference while still remaining robust to signal dropout or noise.

One under-explored technique we briefly investigated was using a **weighted geometric mean** instead of the standard weighted arithmetic mean during inference. The idea is to mitigate the impact of skewed or overconfident probabilities from certain windows. The geometric mean has the advantage of tempering extreme values and emphasizes consensus across predictions. The formulation is as follows:

$$p_{\text{trial}} = \frac{1}{Z} \cdot \exp\left(\frac{\sum_i w_i \cdot \log(p_i)}{\sum_i w_i}\right)$$

where Z is a normalization constant to ensure the resulting vector forms a valid probability distribution. This approach may provide better calibration in noisy scenarios but was not fully explored due to time constraints.

- **Implementation** Inference for individual models is primarily handled in `utils/training.py` via the `predict` function.

4.8 MI Ensemble Strategy: Weighted Rank Averaging

To combine the predictions of our MI models, we adopt a method known as **Weighted Rank Averaging**, which proved significantly more reliable than traditional soft voting.

Why Not Soft Voting?

In soft voting, we would average the predicted probability distributions from each model:

$$\hat{P} = \frac{1}{M} \sum_{m=1}^{M} P^{(m)} \in \mathbb{R}^{N \times C}$$

where:

- M is the number of models,
- N is the number of samples,
- C is the number of classes,
- $P^{(m)} \in \mathbb{R}^{N \times C}$ is the probability matrix from model m, where row i is the probability vector for sample i.

However, our models were trained under different settings (e.g., adversarial training, variable window sizes), and their outputs were often poorly calibrated — meaning the confidence levels were not comparable across models. This led to cases where one overconfident model could dominate the average, even when incorrect.

Weighted Rank Averaging: What and Why

To mitigate this, we use a rank-based strategy. Instead of aggregating raw probabilities, we aggregate the **ranks of probabilities** for each class across models. This makes the ensemble more robust to calibration mismatches.

Step-by-step procedure:

1. Each model $m \in \{1, \ldots, M\}$ produces a matrix $P^{(m)} \in \mathbb{R}^{N \times C}$ of predicted probabilities.
2. For each class $c \in \{1, \ldots, C\}$, we compute the rank of the probability values across all N samples:
$$R^{(m)}_{:,c} = \texttt{rankdata}(P^{(m)}_{:,c}, \text{method="average"})$$
where higher probabilities receive higher ranks (e.g., 1st, 2nd, ..., N-th).
3. These ranks are then aggregated across models using weights w_m:
$$R_{i,c} = \frac{1}{\sum_{m=1}^{M} w_m} \sum_{m=1}^{M} w_m \cdot R^{(m)}_{i,c}$$
where weights are chosen manually during practice. giving us the weighted average rank of class c for sample i.

4. We normalize the ranks to form a pseudo-probability distribution:

$$\tilde{P}_{i,c} = \frac{R_{i,c}}{\sum_{j=1}^{C} R_{i,j}}$$

5. Finally, we predict the class with the highest normalized rank:

$$\hat{y}_i = \arg\max_c \tilde{P}_{i,c}$$

Concrete Example (Binary Classification)

Suppose we have $M = 2$ models, $C = 2$ classes, and $N = 3$ samples. Let:

$$P^{(1)} = \begin{bmatrix} 0.9 & 0.1 \\ 0.7 & 0.3 \\ 0.2 & 0.8 \end{bmatrix} \quad P^{(2)} = \begin{bmatrix} 0.6 & 0.4 \\ 0.2 & 0.8 \\ 0.3 & 0.7 \end{bmatrix}$$

Now we rank each class column-wise across samples (higher value = higher rank):

$$\text{Ranks for class 0:} \quad R^{(1)}_{:,0} = [3,\ 2,\ 1], \quad R^{(2)}_{:,0} = [3,\ 1,\ 2]$$

$$\text{Ranks for class 1:} \quad R^{(1)}_{:,1} = [1,\ 2,\ 3], \quad R^{(2)}_{:,1} = [1,\ 3,\ 2]$$

Average ranks (equal weights $w_1 = w_2 = 1$):

$$R = \frac{1}{2}\left(\begin{bmatrix} 3 & 1 \\ 2 & 2 \\ 1 & 3 \end{bmatrix} + \begin{bmatrix} 3 & 1 \\ 1 & 3 \\ 2 & 2 \end{bmatrix} \right) = \begin{bmatrix} 3.0 & 1.0 \\ 1.5 & 2.5 \\ 1.5 & 2.5 \end{bmatrix}$$

Normalize row-wise:

$$\tilde{P} = \begin{bmatrix} 0.75 & 0.25 \\ 0.375 & 0.625 \\ 0.375 & 0.625 \end{bmatrix} \Rightarrow \hat{y} = [\text{class 0},\ \text{class 1},\ \text{class 1}]$$

- **Implementation** The Weighted Rank Averaging strategy is implemented in `utils/rank_ensemble.py` and is utilized within `inference/inference.py` to combine predictions from multiple MI models.

Summary

Weighted Rank Averaging avoids being misled by overconfident or underconfident models. By focusing on *relative rankings* rather than raw probability values, it offers a calibration-agnostic and more stable ensembling method — especially useful when base learners are diverse in architecture and training dynamics.

Performance Summary

Table 1: Validation and Test (Leaderboard) Scores of Individual Models and Ensembles

Model	Validation Score	Leaderboard Score
MTCFormer 1 (ours)	0.704	0.718
MTCFormer 2 (ours)	0.612	0.71
MTCFormer 3 (ours)	0.657	0.721
Ensemble (Soft Voting)	–	0.719
Ensemble (Rank Averaging)	–	**0.740 (best scores 0.759)**
SSVEPFormer (baseline)	0.590	0.628
Deep4Net (baseline)	0.550	0.665

5 SSVEP Pipeline

5.0.1 Preprocessing

The preprocessing steps for the SSVEP model are mostly shared with the MI pipeline, except for the bandpass range and selected channels.

- **Channel Selection:** EEG channels used: OZ, PO7, PO8, and PZ.
- **Auxiliary Sensor Norms:** The L2 norms of accelerometer and gyroscope signals are computed using:

$$\mathrm{Acc}_{\mathrm{norm}} = \sqrt{\mathrm{Acc}_x^2 + \mathrm{Acc}_y^2 + \mathrm{Acc}_z^2}, \quad \mathrm{Gyro}_{\mathrm{norm}} = \sqrt{\mathrm{Gyro}_x^2 + \mathrm{Gyro}_y^2 + \mathrm{Gyro}_z^2}$$

  Only the norm is retained; raw axes are discarded.
- **Validation Signal:** Included as an additional auxiliary channel.
- **Notch Filtering:** Powerline noise is suppressed using IIR notch filters at 50 and 100.
- **Bandpass Filtering:** A zero-phase FIR filter is used to retain frequencies between 8 and 14, targeting the SSVEP frequency.

5.0.2 Data Segmentation and normalization

To enhance training data diversity, each trial is segmented using a sliding window:

- **Window size:** 500 samples
- **Stride:** 50 samples

This segmentation scheme creates overlapping views of the same trial.

Each segment is then **normalized** to have zero mean and unit variance using **Z-score normalization**.

Implementation All preprocessing-related utilities are defined in `utils/preprocessing.py`, although some parts—such as computing the L2 norms of the accelerometer and gyroscope signals—are handled directly in `convert_csv_to_fif.py`.

5.0.3 Training Configuration

- **Data Augmentation:** We apply heuristic additive Gaussian noise with:

  $$\tilde{x} = x + \mathcal{N}(0, 0.1^2)$$

  No adversarial augmentation was used.

- **Domain Adversarial Training:** Not used — we set `domain_lambda` $= 0$ during training, effectively disabling the domain classifier and gradient reversal path.

- **Architecture configuration:** `depth = 1`, `kernel_size = 10`, `n_times = 500`, `chs_num = 7` (4 EEG + 3 auxiliary), `class_num = 4`, `class_num_domain = 30`, `modulator_dropout = 0.7`, `mid_dropout = 0.7`, `output_dropout = 0.7`, `weight_init_mean = 0.0`, `weight_init_std = 0.05`

- **Optimizer:** Adam optimizer with learning rate = 0.001

- **Loss Calculation:** Like in the MI pipeline, we apply window-wise reliability weighting using the mean validation signal:

  $$\mathcal{L}_i = w_i \cdot \text{CE}(f(x_i), y), \quad \mathcal{L}_{\text{total}} = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}_i$$

- **Learning Rate Scheduler:** MultiStepLR with decay $\gamma = 0.1$ applied at epoch 250

- **Implementation** This model is fully implemented in `train/train_ssvep_model.py`, where all preprocessing and training steps are handled end-to-end.

5.0.4 Inference Technique

For inference, we follow the same methodology outlined in Section 4.7 (MI Pipeline). Each trial is segmented into overlapping windows and passed through the model to obtain per-window class probabilities.

To aggregate these predictions into a final class probability per trial, we apply a weighted averaging scheme, where each window's contribution is scaled by the mean of its associated `validation` signal—favoring cleaner, less noisy windows.

If all weights are close to zero (indicating universally low signal quality), we fallback to an unweighted mean across all window predictions.

This strategy ensures that the final prediction is both robust to local noise and sensitive to signal integrity—exactly as done in our MI inference setup.

5.0.5 Results (SSVEP Pipeline)

The table below summarizes the performance of our SSVEP model alongside two commonly used baselines. Our proposed MTCFormer-based pipeline achieved a strong validation accuracy and leaderboard score, especially when paired with our best-performing MI system.

Due to limited time, we were not able to fully explore more advanced architectural variations or training strategies (e.g., deeper convolutional depths, adversarial training, or multi-scale frequency decomposition). We believe further tuning and ablation could unlock additional improvements.

Model	Validation Accuracy	Leaderboard Score
MTCFormer (ours)	**0.710**	**0.740 (best scores 0.759)**
SSVEPFormer + Complex Representations	0.653	0.690
Deep4Net	0.560	0.620

Table 2: Comparison of validation and leaderboard performance for different SSVEP models.

6 Reproducibility and Usage

To ensure full reproducibility and ease of use, we provide all necessary scripts, configuration files, and instructions in our repository. This includes training, inference, and evaluation pipelines, along with clear directory structures and expected inputs.

A step-by-step tutorial for reproducing our results — including how to set up the environment, run training for each model, and perform inference using pre-trained checkpoints — is included in the `README.md` file of the repository.

Each of the training and inference scripts is carefully documented with inline comments, explaining the logic and purpose of each step to make the code easy to understand and extend.

We encourage readers to refer to the `README.md` for detailed guidance on installation, usage, and troubleshooting.

7 Challenges We Faced Along the journey

Classical Machine Learning Pipelines: We attempted several traditional pipelines using handcrafted features such as Hjorth parameters, power spectral density (PSD), and band power features. These were followed by models like gradient boosting and support vector machines (SVMs). However:

- Gradient Boosters often exhibited noisy and unstable training dynamics.
- Linear models like SVMs often converged to trivial solutions — such as assigning the same probability to all samples.
- Filtering-based pipelines (e.g., CSP or CCA after bandpass and notch filtering) for SSVEP were difficult to tune and failed to generalize.

Signal Repair via Reflection Interpolation: To handle segments where the validation signal was zero (i.e., unreliable EEG), we experimented with a technique we termed *Interpolation by Reflection*. The idea was to reflect both sides of the surrounding EEG signal into the corrupted region to avoid sharp discontinuities and mitigate interpolation artifacts. Unfortunately, the method had limited success — artifacts persisted, especially after filtering. We suspect this was due to unavoidable phase discontinuities at segment borders, which caused filters (especially IIRs) to overshoot, introducing ringing artifacts.

Evaluation Metrics Instability: During validation, we found that the `F1-score` was sometimes misleading, especially in trivial prediction scenarios (e.g., where all predictions defaulted to one class). To combat this, we routinely inspected the confusion matrix and monitored the `Balanced Accuracy Score` as a more stable alternative.

Unknown Signal Units and Scaling Issues: One subtle but frustrating challenge was the absence of known physical units for EEG and sensor data. While z-score normalization was applied before training, performing EDA (e.g., with MNE) was tricky due to unknown scaling. We had to manually tune scaling factors to make the data visually interpretable.

Tabular Deep Learning Attempts: We also explored tabular deep learning approaches using architectures like TabNet on Hjorth parameters, CSP/CCA projections, and PSD features. However, training was highly unstable, and performance was significantly worse than even basic deep learning baselines.

Very Small Test and Validation Sets: One of the most fundamental challenges was the extremely limited size of the test and validation sets. This made it difficult to build robust models that generalized well, especially when small overfitting could significantly affect leaderboard placement. While we considered using cross-validation to improve robustness, we ultimately decided against it. Applying CV to deep learning models would've added overwhelming complexity to an already intricate pipeline — and while feasible for classical models, it was nearly impractical in our context.

Stabilizing Domain Adaptation: One of the key challenges we encountered was stabilizing the domain adaptation process. The domain loss is controlled by a hyperparameter λ, which governs the strength of the alignment pressure across subjects. We observed that setting λ too high at the start of training caused the model to become unstable, as the feature extractor was overwhelmed by the need to enforce domain invariance before learning task-relevant representations. To address this, we implemented a scheduler to gradually increase λ throughout training. However, due to time constraints, we were unable to extensively explore or tune this schedule. Consequently, we opted for a small fixed value of $\lambda = 0.01$. While this setting was not ideal, it still produced a noticeable improvement for motor imagery (MI) models. Interestingly, domain adaptation did not yield meaningful gains for SSVEP models, suggesting that subject variability may be less critical in that setting.

Settling for a Lower-Performing but Cleaner System:

One notable issue was that our best submission during development achieved a public leaderboard score of 0.759. Unfortunately, we had to settle for a lower-scoring version (0.740) in the final submission. This was due to one of the MTCFormer models in the ensemble being trained within a very unclean and disorganized codebase. We were unable to integrate or sanitize that component in time, and chose not to include it to preserve the overall reproducibility and clarity of the system.

Starting From Zero: Entering the competition with no prior experience in BCI or EEG processing was itself a massive challenge. At the beginning, even the terminology and file formats were foreign to us. Every part of the pipeline — from understanding the data to building domain-adaptive models — had to be learned from scratch, often through trial and error.

8 Conclusion and Reflections

I hope this system description has succeeded in delivering enough theoretical insight to help you understand the motivations and mechanics behind our proposed solution. It was written with care to highlight the reasoning behind each design decision — from preprocessing to architecture and training strategies.

Thank you for taking the time to read through our work. It has been a pleasure to share the ideas and methods that went into building this system.

— *Mohammed Ahmed Abd-Al-Azim Metwally*