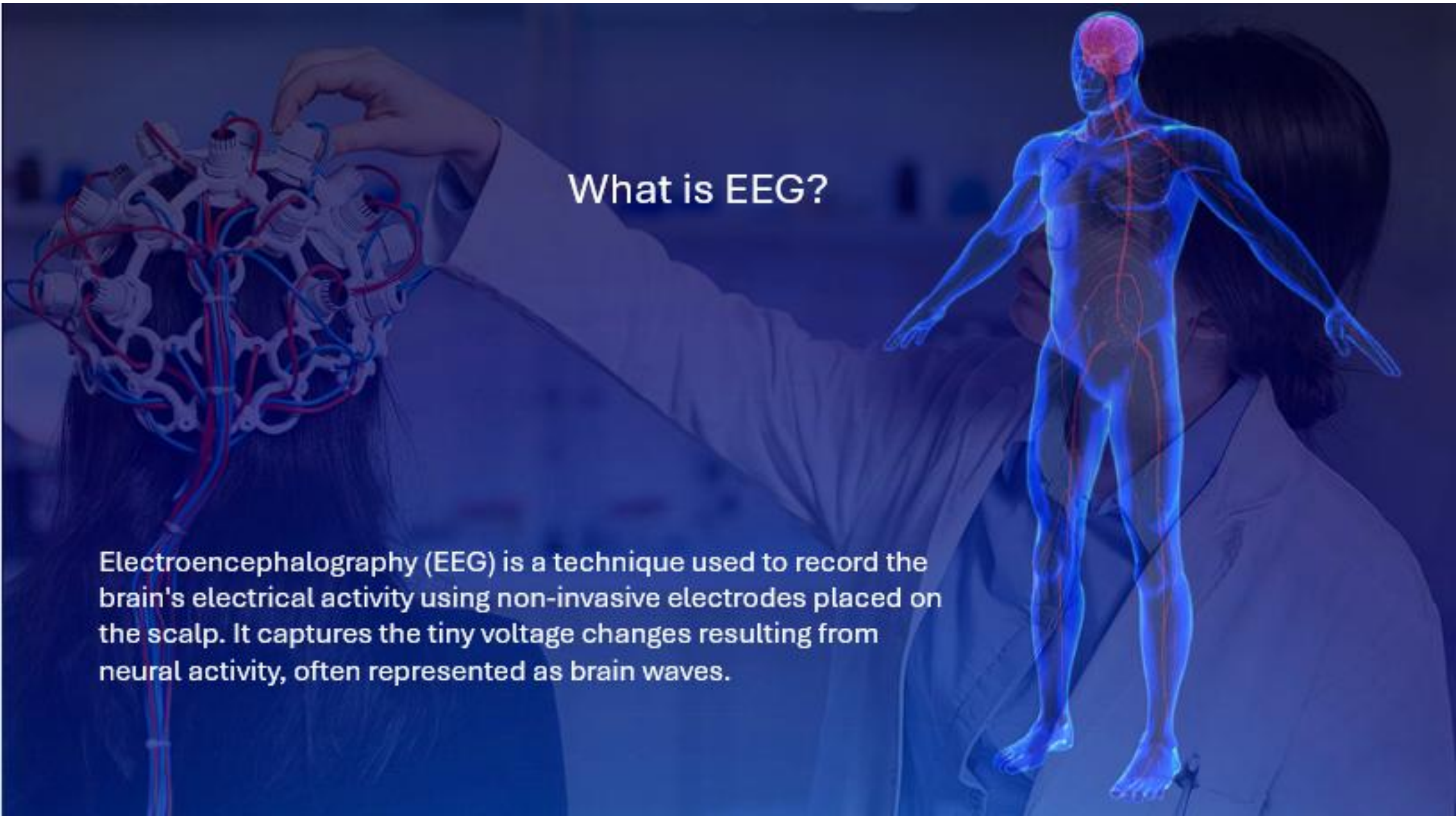

Smart EEG Analyzer

What is EEG?



Electroencephalography (EEG) is a technique used to record the brain's electrical activity using non-invasive electrodes placed on the scalp. It captures the tiny voltage changes resulting from neural activity, often represented as brain waves.

Why EEG?

EEG helps doctors detect abnormal brain activity in conditions like epilepsy, sleep disorders, and brain injuries.
It also enables AI-powered brain-computer interfaces that allow communication and control through brain signals — especially helpful in assistive tech for patients with disabilities.

AI & EEG

In our project, we used Machine Learning, to enhance EEG analysis by:

•Learning complex patterns from preprocessed EEG data that are difficult to detect manually.

•Classifying brain states such as normal and abnormal activity based on EEG signal features.

•Improving diagnostic assistance, particularly in tasks like:

- Abnormal EEG detection
- Supporting neurological screening
- Enabling future integration with assistive technologies (e.g., brain-controlled systems)

System Architecture

Frontend:
Built with HTML/CSS and JavaScript frameworks for interactivity.
Backend:
Developed using Flask (Python) to handle server-side operations and cloudflare database.
Machine Learning Model:
Utilizes a Multi-Layer Perceptron (MLP) neural network trained to classify EEG signals.
Data Handling:
EEG data is preprocessed and scaled before being fed into the model for prediction.

Machine Learning Methodology

1. **Dataset & Labeling:**
   - We utilized the **Temple University Hospital Abnormal EEG Corpus (TUAB v3.0.1)** — a clinically validated dataset designed for EEG signal classification.
   - Access was obtained by submitting a **Data Use Agreement (DUA)** through the official TUH EEG portal

Dataset Composition

- EEG recordings are stored in **European Data Format (EDF)**.
- Each file contains:
  - Multichannel brainwave signals from up to **36 scalp electrodes**
  - **Patient metadata** (age, gender, recording date)
  - Sampling frequencies typically between **250–500 Hz**

MNE?

MNE is an open-source Python library specialized for processing, analyzing, and visualizing **EEG** data.
It supports reading **EDF files**, applying filters, handling channels, and extracting features for machine learning.

EEG signals are loaded using the MNE Python library, which provides tools such as read_raw_edf() for reading files, filter() for bandpass filtering, and get_data() for accessing signal arrays — enabling efficient preprocessing and analysis.

- mne.io.read_raw_edf() – to read EDF EEG files
- raw.filter(l_freq, h_freq) – to apply bandpass filter
- raw.get_data() – to extract signal data as NumPy arrays

Data Representation

- EEG signals are loaded using the MNE Python library, producing:
  - EEG data is stored as a **2D floating-point array** with shape **(channels × samples)**.

    For example, **(36, 10,000)** means the recording has **36 EEG channels**, each containing **10,000 time points**.

    The number of samples represents how many times the signal was recorded per second — this is controlled by the **sampling rate**.

    If the sampling rate is **250 Hz**, it means the signal was recorded **250 times per second**, so **10,000 samples ≈ 40 seconds** of data.

    This format allows us to apply filters, extract features, and segment the data for machine learning.
  - Each channel is a 1D NumPy array of type float64, representing voltage values in microvolts (μV)
- These continuous signals are segmented into fixed-length windows (e.g., 1000 samples) for model input.

Data preprocessing

- Each EEG file is labeled as:
  - 0 → Normal
  - 1 → Abnormal
  - based on folder path and expert annotations.
- Every extracted window inherits the label of its parent recording.

Artifact Removal

- EEG recordings often include **non-brain signals** known as **artifacts**, which interfere with true brainwave patterns.
- Common sources of artifacts:
  - **Muscle movement** (e.g., jaw clenching)
  - **Eye blinks and body motion**
  - **Loose or poorly connected electrodes**
- Since these artifacts can distort the signal and confuse the model, we applied **automatic artifact detection** only on **normal EEG recordings** to avoid accidentally removing pathological information.
- We used the annotate_muscle_zscore() function from the **MNE library**, which:
  - Analyzes the EEG signal in the **30–90 Hz** range
  - Tags segments with strong muscle-related noise
  - Automatically removes those segments from the data

Filtering and Scaling

- Following artifact removal, we performed frequency filtering to isolate brainwave ranges relevant for analysis.
- A bandpass filter (0.5–40 Hz) was applied to:
  - Eliminate low-frequency drifts (<0.5 Hz), often caused by motion or sweat
  - Remove high-frequency noise (>40 Hz), including electrical interference and residual muscle signals
  - Retain only clinically meaningful frequencies (delta, theta, alpha, beta)
- Unlike artifact removal, which removes specific segments, filtering adjusts the full signal to clean its frequency content.
- We also converted EEG signals from Volts to Microvolts (μV) to meet clinical standards using a scaling factor of 1e6.
- These operations were performed using the Preprocessor() class from Braindecode, which:
  - Applies bandpass filtering
  - Performs unit conversion
  - Ensures uniform preprocessing across all recordings
- The result is a set of standardized, noise-reduced EEG signals ready for segmentation and feature extraction.

Segmentation (Windowing)

- After preprocessing, each EEG recording was split into **non-overlapping windows of 1000 samples** using Braindecode's create_fixed_length_windows() function.
- This standardizes input size across all recordings, regardless of their original duration.
- **Start offset** was set to 0, and **windows that didn’t fit** exactly were discarded (drop_last_window=True).
- Each window **inherited the binary label** (0 = normal, 1 = abnormal) from its parent EEG file.
- This step created a uniform dataset structure, making it suitable for deep learning and efficient training.

Class Balancing

- After segmentation, the number of **abnormal windows was significantly higher** than normal ones, which could bias the model.
- To address this, we used RandomOverSampler from imblearn to **duplicate samples from the minority class (normal)** and balance both classes.
- Before applying oversampling, the window data was **flattened to 2D** (samples, features)using reshape() to match the expected input format for the sampler.
- After resampling, the data was reshaped back into its original **3D format (samples × channels × time)** for model compatibility.

This ensured that both normal and abnormal EEG patterns were equally represented during training, improving model fairness and performance.

Handcrafted Feature Extraction

- After balancing, each EEG window (shape: channels × time) was processed to extract handcrafted features that describe the signal behavior numerically.
- We extracted features only from those selected in the selected_feature_names.txt file to save time and reduce overfitting.
- Features were computed for each channel independently and then concatenated into a single vector per window.

- Time-Domain Features

→ Captures basic statistical properties of the raw signal:

- Mean, Standard Deviation, Minimum, Maximum
- Skewness (asymmetry), Kurtosis (peakedness)

Feature Extraction

Frequency-Domain Features (PSD)

•EEG signals can be **analyzed by their frequency content**, We used **Welch's method** (a fast Fourier-based approach) to calculate the **average power** in standard brainwave bands:

- **Delta (0.5–4 Hz)** – deep sleep
- **Theta (4–8 Hz)** – drowsiness or meditation
- **Alpha (8–13 Hz)** – calm, relaxed state
- **Beta (13–30 Hz)** – active thinking
- **Gamma (30–40 Hz)** – high-level cognitive processing

This helps identify which brain states are dominant in the signal.

Hjorth Parameters

These are **three statistical measures** based on signal derivatives (how fast the signal changes):

1.Activity – Variance of the signal → how strong or active the signal is overall

2.Mobility – Standard deviation of the signal's first derivative → how quickly it changes

3.Complexity – Compares the first and second derivatives → how "irregular" or unpredictable the signal shape is

They offer a quick snapshot of signal dynamics and structure.

Catch22 Features

•**Catch22** is a toolkit that computes **22 standardized statistical features** from any time-series signal.

•It includes:

- **Entropy** – how random the signal is
- **Autocorrelation** – how much the signal repeats over time
- **Outlier detection** – checks for extreme spikes
- **Nonlinearity metrics** – captures irregular, chaotic behavior

These features summarize a wide range of behaviors in a compact format, useful for machine learning.

Feature Selection

After extracting a large number of features from each EEG window, not all of them were useful for classification.
To avoid training on irrelevant or noisy features, we applied a feature selection strategy based on model performance.

How it Worked

- We trained an initial version of the model using all features.
- We then measured the importance of each feature by checking how much it contributed to improving the model's F1-score (a balance between precision and recall).
- Based on this analysis, we kept only the most impactful features and discarded the rest.

Final Setup

- The names of the selected features were stored in a file: selected_feature_names.txt
- During feature extraction, we computed only the selected features, skipping everything else.

This reduced the dataset size, increased training speed, and improved model accuracy by avoiding overfitting.

Model

Train–Validation Split

- EEG feature data and labels were split into 80% training and 20% validation sets.
- Stratified split was used to maintain class balance.
- Ensures that the model is trained on diverse data and evaluated fairly.

Data Loaders

- Created Data Loader objects for batching and shuffling:
  - Batch size = 64
- Improves training performance and generalization.

Class Weights Handling

Even after oversampling, slight class imbalance can remain due to the train-validation split.
To address this, we:

- Computed final class weights using compute_class_weight() on the training set only.
- Applied these weights in CrossEntropyLoss() to ensure fair learning across both classes.
- Class weights balance the model's attention during training.

MLP Model Architecture

- Implemented a custom Multi-Layer Perceptron (MLP) with:
  - 3 hidden layers (256 → 128 → 64 neurons)
  - BatchNorm, ReLU activations, Dropout
  - Output layer: 2 neurons (normal vs abnormal)
- Designed to balance accuracy and computational efficiency
- Optimizer: Adam with learning rate = 1e-3
- Loss Function: Weighted Cross Entropy.
- Trained for up to 100 epochs with performance tracking.