Import all the essential libraries required to build, train, evaluate, and analyze a deep learning model, commonly used in EEG-based seizure detection tasks:

-> NumPy is used for numerical computations and handling multi-dimensional arrays, which form the core structure of EEG signal data.

->TensorFlow and Keras provide the deep learning framework, where layers and the Model class are used to design neural network architectures such as CNNs and BiLSTMs.

->Training optimization is handled using callbacks like EarlyStopping, which stops training when validation performance no longer improves, ModelCheckpoint, which saves the best-performing model during training, and ReduceLROnPlateau, which lowers the learning rate when learning stagnates to improve convergence and prevent overfitting.

->Scikit-learn is used for model validation and performance evaluation, with Leave-One-Out cross-validation ensuring robust testing by training on all samples except one iteratively, and evaluation metrics such as accuracy, precision, recall, F1-score, ROC-AUC, and confusion matrix measuring classification effectiveness.

->Matplotlib and Seaborn are visualization libraries used to plot EEG signals, training curves, and confusion matrices for interpretability and analysis.

->SciPy's signal module provides signal processing tools such as filtering and spectral analysis, which are crucial for EEG preprocessing.

->OS module supports file and directory management for dataset handling, while the warnings module suppresses non-critical runtime warnings to keep outputs clean and focused.

In [1]:
```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, Model
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceL
from sklearn.model_selection import LeaveOneOut
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import signal as scipy_signal
import os
import warnings
warnings.filterwarnings('ignore')
```

```
2025-12-15 04:00:47.770600: I tensorflow/core/util/port.cc:113] oneDNN custom o
perations are on. You may see slightly different numerical results due to float
ing-point round-off errors from different computation orders. To turn them off,
set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2025-12-15 04:00:47.843706: I tensorflow/core/platform/cpu_feature_guard.cc:21
0] This TensorFlow binary is optimized to use available CPU instructions in per
formance-critical operations.
To enable the following instructions: AVX2 AVX_VNNI FMA, in other operations, r
ebuild TensorFlow with the appropriate compiler flags.
2025-12-15 04:00:48.847290: W tensorflow/compiler/tf2tensorrt/utils/py_utils.c
c:38] TF-TRT Warning: Could not find TensorRT
```

GPU configuration:

This function configures TensorFlow to efficiently use a GPU for deep learning training and prints detailed diagnostic information in a Jupyter Notebook text (Markdown) cell.

->It first checks whether any physical GPUs are available using TensorFlow's device listing utilities; if GPUs are detected, it enables **memory growth**, which allows TensorFlow to allocate GPU memory dynamically instead of reserving all memory at startup, preventing out-of-memory errors and improving resource sharing.

->The function then reports the number and names of detected GPUs. To accelerate training on modern NVIDIA RTX GPUs, it enables **mixed precision training**, which uses both 16-bit (FP16) and 32-bit (FP32) floating-point computations; this leverages **Tensor Cores** to significantly speed up matrix operations while maintaining numerical stability.

->**Soft device placement** is enabled so TensorFlow can automatically place operations on the GPU when possible and fall back to the CPU if needed.

->The function also retrieves and prints low-level **GPU hardware details** such as compute capability, which helps verify compatibility and performance characteristics.

->If GPU configuration is attempted after TensorFlow has already initialized devices, a runtime error is safely caught and reported. When no GPU is available, the function clearly indicates that computation will run on the CPU.

->Finally, the function is called immediately so that GPU configuration is applied as soon as the notebook or script is executed.

In [2]:
```
def configure_gpu():
```

```
    print("="*80)
    print("GPU CONFIGURATION")
    print("="*80)

    # Check available GPUs
    gpus = tf.config.list_physical_devices('GPU')

    if gpus:
        try:
            # Enable memory growth for all GPUs
            for gpu in gpus:
                tf.config.experimental.set_memory_growth(gpu, True)

            print(f"Found {len(gpus)} GPU(s):")
            for i, gpu in enumerate(gpus):
                print(f"  GPU {i}: {gpu.name}")

            # Enable mixed precision training for RTX GPUs (faster training)
            # RTX GPUs have Tensor Cores that accelerate FP16 operations
            policy = tf.keras.mixed_precision.Policy('mixed_float16')
            tf.keras.mixed_precision.set_global_policy(policy)
            print(f"Mixed precision enabled: {policy.name}")
            print(" (Utilizes Tensor Cores on RTX GPU for faster training)")

            # Set TensorFlow to use GPU
            tf.config.set_soft_device_placement(True)
            print("GPU acceleration enabled")

            # Display GPU compute capability
            gpu_details = tf.config.experimental.get_device_details(gpus[0])
            print(f"GPU Details: {gpu_details}")

        except RuntimeError as e:
            print(f"GPU configuration error: {e}")
    else:
        print("No GPU found. Running on CPU.")

    print("="*80 + "\n")

# Call GPU configuration at import time
configure_gpu()
```

```
================================================================================
=
GPU CONFIGURATION
================================================================================
=
Found 1 GPU(s):
  GPU 0: /physical_device:GPU:0
Mixed precision enabled: mixed_float16
 (Utilizes Tensor Cores on RTX GPU for faster training)
GPU acceleration enabled
GPU Details: {'compute_capability': (8, 6), 'device_name': 'NVIDIA GeForce RTX
3060 Laptop GPU'}
================================================================================
=
```

```
2025-12-15 04:00:52.455757: I external/local_xla/xla/stream_executor/cuda/cud
a_executor.cc:984] could not open file to read NUMA node: /sys/bus/pci/devices/
0000:01:00.0/numa_node
Your kernel may have been built without NUMA support.
2025-12-15 04:00:52.539921: I external/local_xla/xla/stream_executor/cuda/cud
a_executor.cc:984] could not open file to read NUMA node: /sys/bus/pci/devices/
0000:01:00.0/numa_node
Your kernel may have been built without NUMA support.
2025-12-15 04:00:52.540013: I external/local_xla/xla/stream_executor/cuda/cud
a_executor.cc:984] could not open file to read NUMA node: /sys/bus/pci/devices/
0000:01:00.0/numa_node
Your kernel may have been built without NUMA support.
2025-12-15 04:00:52.541716: I external/local_xla/xla/stream_executor/cuda/cud
a_executor.cc:984] could not open file to read NUMA node: /sys/bus/pci/devices/
0000:01:00.0/numa_node
Your kernel may have been built without NUMA support.
```

STEP 1: DATA LOADING AND PREPARATION

This function loads preprocessed EEG signal data and associated patient information in a format suitable for deep learning models and explains the rationale behind using .npz files.

->It scans a specified directory and loads all .npz files, where each file typically represents one patient and stores multiple related arrays in a single compressed container.

->For each file, the function extracts the patient ID from the filename, then loads the EEG signals—structured as multi-dimensional arrays with dimensions representing samples, channels, and time steps—and the corresponding binary

labels indicating seizure or non-seizure events.

->These arrays are then appended to lists so that data from multiple patients can be processed together during training or cross-validation.

->The function also loads demographic embeddings from a separate .npy file, which encode patient-level information (such as age or gender) in a numerical format that can be fused with EEG features in neural networks.

->.npz files are used in deep learning because they efficiently store multiple NumPy arrays in a single compressed file, preserve exact numerical precision without loss, load significantly faster than text-based formats like CSV, and maintain consistent array shapes required by neural networks.

->This makes them ideal for large, high-dimensional data such as EEG signals, where fast I/O, memory efficiency, and structural integrity are critical for stable and reproducible model training.

In [3]:
```python
def load_preprocessed_data(data_dir, demographic_file):

    print("Loading preprocessed EEG data and demographic embeddings:")

    eeg_data = []
    labels = []
    patient_ids = []

    # Load all .npz files from directory
    for file in sorted(os.listdir(data_dir)):
        if file.endswith('.npz'):
            print(f"Loading {file}....")
            patient_id = file.split('.')[0]
            data = np.load(os.path.join(data_dir, file))

            # eeg_data.append(data['eeg_signal'])  # Shape: (n_samples, n_char
            # labels.append(data['labels'])  # Shape: (n_samples,) - 0: non-se
            eeg_data.append(data['X'])  # Shape: (n_samples, n_channels, time_
            labels.append(data['y'])  # Shape: (n_samples,) - 0: non-seizure,
            patient_ids.append(patient_id)
            print(f"Loaded {file}")

    # Load demographic embeddings
    demographics = np.load(demographic_file)  # Shape: (14, 14)

    print(f"Loaded data for {len(patient_ids)} patients")
    print(f"Demographics shape: {demographics.shape}")

    return eeg_data, labels, demographics, patient_ids
```

STEP 2: TIME-FREQUENCY REPRESENTATION (STFT)

Convert raw EEG time-domain signals into a **time–frequency representation** using the **Short-Time Fourier Transform (STFT)**, which is especially useful for seizure detection because seizures exhibit distinctive frequency patterns that change over time.

->The `compute_stft` function takes a single EEG segment (organized by channels) and applies STFT independently to each channel using a specified sampling frequency (`fs`), window length (`nperseg`), and overlap (`noverlap`);

->STFT works by splitting the signal into short, overlapping time windows and applying the Fourier Transform to each window, allowing the model to observe how signal frequencies evolve over time rather than assuming stationarity.

->The complex STFT output is converted to magnitude values using the absolute function, since magnitude spectra capture signal energy distribution and are more meaningful for neural networks.

->The result is a 3D tensor representing channels, frequency bins, and time frames, which closely resembles image-like data suitable for CNNs.

->The `prepare_data_with_stft` function applies this transformation to the entire dataset patient-wise and segment-wise, enabling optional preprocessing control through the `use_stft` flag.

->Using STFT enhances deep learning performance in EEG analysis because seizures are characterized by transient oscillations and spectral shifts that are difficult to detect in raw time-domain signals but become clearly separable in the time–frequency domain.

In [4]:
```
def compute_stft(eeg_segment, fs=256, nperseg=128, noverlap=64):
    n_channels = eeg_segment.shape[0]
    stft_results = []

    for ch in range(n_channels):
        f, t, Zxx = scipy_signal.stft(eeg_segment[ch], fs=fs,
                                      nperseg=nperseg, noverlap=noverlap)
        stft_results.append(np.abs(Zxx))

    return np.array(stft_results)  # Shape: (n_channels, freq_bins, time_frame
```

```python
def prepare_data_with_stft(eeg_data, use_stft=True):
    if not use_stft:
        return eeg_data

    print("Computing STFT for time-frequency representation...")
    processed_data = []

    for patient_data in eeg_data:
        patient_stft = []
        for segment in patient_data:
            stft_segment = compute_stft(segment)
            patient_stft.append(stft_segment)
        processed_data.append(np.array(patient_stft))

    return processed_data
```

STEP 3: DATA AUGMENTATION

This function performs **data augmentation** on EEG segments to artificially increase dataset diversity and improve the generalization ability of deep learning models.

->It first creates a copy of the original EEG segment to preserve the raw data, then applies one of several biologically plausible transformations based on the selected augmentation type.

->In the **noise augmentation**, small Gaussian noise is added to simulate real-world recording disturbances such as sensor noise or environmental interference, helping the model become robust to slight signal variations.

->The **time-shift augmentation** randomly shifts the EEG signal along the time axis, which teaches the model that seizure patterns are invariant to small temporal misalignments and reduces sensitivity to exact onset positions.

->The **channel dropout augmentation** randomly zeros out a subset of EEG channels, mimicking electrode failures or poor contact and encouraging the model to learn spatially distributed patterns rather than relying on a few dominant channels.

->Overall, these augmentation strategies help prevent overfitting, improve robustness to noise and missing data, and make the model more reliable when applied to unseen EEG recordings in real clinical settings.

In [5]:
```python
def augment_eeg_data(eeg_segment, augmentation_type='noise'):
    augmented = eeg_segment.copy()

    if augmentation_type == 'noise':
        # Add Gaussian noise (SNR ~20dB)
        noise = np.random.normal(0, 0.1, augmented.shape)
        augmented = augmented + noise

    elif augmentation_type == 'shift':
        # Time shifting
        shift = np.random.randint(-10, 10)
        augmented = np.roll(augmented, shift, axis=-1)

    elif augmentation_type == 'dropout':
        # Random channel dropout (10% of channels)
        n_channels = augmented.shape[0]
        dropout_channels = np.random.choice(n_channels,
                                            size=int(0.1 * n_channels),
                                            replace=False)
        augmented[dropout_channels] = 0

    return augmented
```

STEP 4: ATTENTION MECHANISM

This custom **AttentionLayer** implements a learnable attention mechanism that allows a neural network to focus on the most informative time steps in sequential EEG data, which is crucial for seizure detection where only certain temporal regions contain discriminative seizure activity.

->The layer extends Keras's base `Layer` class and defines trainable parameters in the `build` method, including a **weight matrix** initialized using Glorot (Xavier) initialization for stable gradient flow and a **bias vector** initialized to zeros.

->During the forward pass in the `call` method, the input tensor—structured as batches of time sequences with extracted features—is linearly transformed and passed through a **tanh activation** to produce attention scores that capture the relevance of each time step.

->These scores are normalized using a **softmax function** across the temporal dimension, converting them into attention weights that sum to one and represent the relative importance of each time step.

->The original inputs are then scaled by these weights through element-wise

multiplication, emphasizing seizure-relevant temporal patterns while suppressing less informative background activity.

->The `get_config` method ensures that the layer can be properly serialized and reloaded, making it compatible with model saving and deployment workflows in deep learning applications.

In [6]:
```python
class AttentionLayer(layers.Layer):
    def __init__(self, **kwargs):
        super(AttentionLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        # Attention weight matrix
        self.W = self.add_weight(name='attention_weight',
                                 shape=(input_shape[-1], input_shape[-1]),
                                 initializer='glorot_uniform',
                                 trainable=True)
        self.b = self.add_weight(name='attention_bias',
                                 shape=(input_shape[-1],),
                                 initializer='zeros',
                                 trainable=True)
        super(AttentionLayer, self).build(input_shape)

    def call(self, inputs):
        # Compute attention scores
        # inputs shape: (batch_size, time_steps, features)
        attention_scores = tf.nn.tanh(tf.matmul(inputs, self.W) + self.b)
        attention_weights = tf.nn.softmax(attention_scores, axis=1)

        # Apply attention weights
        attended_output = inputs * attention_weights

        return attended_output

    def get_config(self):
        return super(AttentionLayer, self).get_config()
```

STEP 5: CNN-BiLSTM MODEL WITH ATTENTION AND DEMOGRAPHICS

This function builds a **hybrid CNN–BiLSTM deep learning model with attention and demographic fusion** for EEG-based seizure detection, combining spatial, temporal, and patient-specific information in a single architecture.

->The model uses two inputs: one for EEG signals and one for demographic

embeddings, allowing it to learn both signal-level and subject-level patterns. If STFT features are used, the EEG input is reshaped to make the time–frequency representation compatible with one-dimensional convolutions.

->The **CNN blocks** consist of stacked Conv1D layers with increasing filter sizes that learn local spatial patterns across EEG channels and time, such as rhythmic discharges or spike-like activity, while **batch normalization** stabilizes training, **max pooling** reduces temporal resolution and noise, and **dropout** prevents overfitting.

->The extracted features are then passed to **Bidirectional LSTM layers**, which model long-range temporal dependencies in both forward and backward directions, enabling the network to capture seizure onset, evolution, and offset patterns more effectively.

->An **attention mechanism** is applied next to dynamically emphasize the most seizure-relevant time segments, improving interpretability and detection accuracy. Global average pooling compresses the attended temporal features into a fixed-length representation.

->In parallel, demographic data is processed through a dense layer to learn compact patient embeddings, which are concatenated with EEG-derived features to incorporate patient-specific variability.

->The **classification head** consists of fully connected layers with dropout for robust feature learning, and the final softmax output layer performs binary classification between seizure and non-seizure classes.

->The model is compiled using the **Adam optimizer** for efficient gradient-based learning, **sparse categorical cross-entropy loss** suitable for integer class labels, and evaluation metrics including accuracy, precision, and recall to assess clinical relevance.

In [7]:
```
def build_cnn_bilstm_model(input_shape, demographic_dim=14, use_stft=False):

    # Input Layers
    eeg_input = layers.Input(shape=input_shape, name='eeg_input')
    demographic_input = layers.Input(shape=(demographic_dim,), name='demograph

    # CNN Layers for Spatial Feature Extraction
    # Extract spatial patterns across EEG channels
    if use_stft:
        # For STFT input: (channels, freq_bins, time_frames)
        x = layers.Reshape((input_shape[0], -1))(eeg_input)
    else:
        x = eeg_input
```

```
    # First Conv1D block
    x = layers.Conv1D(filters=64, kernel_size=5, padding='same',
                      activation='relu', name='conv1')(x)
    x = layers.BatchNormalization(name='bn1')(x)
    x = layers.MaxPooling1D(pool_size=2, name='pool1')(x)
    x = layers.Dropout(0.3, name='dropout1')(x)

    # Second Conv1D block
    x = layers.Conv1D(filters=128, kernel_size=5, padding='same',
                      activation='relu', name='conv2')(x)
    x = layers.BatchNormalization(name='bn2')(x)
    x = layers.MaxPooling1D(pool_size=2, name='pool2')(x)
    x = layers.Dropout(0.3, name='dropout2')(x)

    # Third Conv1D block
    x = layers.Conv1D(filters=256, kernel_size=3, padding='same',
                      activation='relu', name='conv3')(x)
    x = layers.BatchNormalization(name='bn3')(x)
    x = layers.MaxPooling1D(pool_size=2, name='pool3')(x)
    x = layers.Dropout(0.3, name='dropout3')(x)

    #  BiLSTM Layers for Temporal Feature Extraction
    # Capture past and future temporal dependencies
    x = layers.Bidirectional(layers.LSTM(128, return_sequences=True,
                                         name='bilstm1'), name='bi_lstm1')(x)
    x = layers.Dropout(0.4, name='dropout4')(x)

    x = layers.Bidirectional(layers.LSTM(64, return_sequences=True,
                                         name='bilstm2'), name='bi_lstm2')(x)
    x = layers.Dropout(0.4, name='dropout5')(x)

    # Attention Mechanism
    # Focus on seizure-relevant temporal and spatial patterns
    x = AttentionLayer(name='attention')(x)

    # Global pooling to aggregate temporal information
    x = layers.GlobalAveragePooling1D(name='global_pool')(x)

    # Demographic Integration
    # Concatenate demographic embeddings with extracted features
    # This helps model learn patient-specific patterns
    demographic_dense = layers.Dense(32, activation='relu',
                                     name='demographic_dense')(demographic_inpu

    # Concatenate EEG features with demographic features
    combined = layers.Concatenate(name='concatenate')([x, demographic_dense])

    # Classification Head
    combined = layers.Dense(128, activation='relu', name='dense1')(combined)
    combined = layers.Dropout(0.5, name='dropout6')(combined)

    combined = layers.Dense(64, activation='relu', name='dense2')(combined)
```

```
    combined = layers.Dropout(0.5, name='dropout7')(combined)

    # Output layer: Binary classification (seizure vs non-seizure)
    output = layers.Dense(2, activation='softmax', name='output')(combined)

    # Build and Compile Model
    model = Model(inputs=[eeg_input, demographic_input], outputs=output)

    # Compile with Adam optimizer and categorical crossentropy loss
    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=0.001),
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy', keras.metrics.Precision(), keras.metrics.Recall()]
    )

    return model
```

STEP 6: LEAVE-ONE-PATIENT-OUT (LOPO) CROSS-VALIDATION

This function implements **Leave-One-Patient-Out (LOPO) cross-validation**, a rigorous evaluation strategy widely used in EEG-based seizure detection to ensure true patient-independent generalization.

->In this approach, data from one patient is held out as the test set while the model is trained on data from all remaining patients, and this process is repeated so that each patient serves as the test subject exactly once.

->For every fold, the function concatenates EEG segments and labels from the training patients, aligns demographic embeddings with each EEG segment by repetition, and keeps the held-out patient’s data strictly unseen during training.

-> A CNN–BiLSTM–Attention model is then built and trained using GPU acceleration, with multiple **model checkpoints** to save the best-performing models based on validation accuracy and validation loss, as well as periodic snapshots to safeguard against training interruptions.

->**Early stopping** halts training when validation loss stops improving, preventing overfitting, while **learning rate reduction on plateau** improves convergence by lowering the learning rate when optimization stagnates.

->**TensorBoard logging** enables real-time monitoring of GPU usage, losses, and metrics.

->After training, the best model (based on validation accuracy) is reloaded and evaluated on the held-out patient using clinically meaningful metrics such as accuracy, precision, recall, F1-score, AUC–ROC, and the confusion matrix.

->These metrics are stored fold-wise to provide a comprehensive performance summary across all patients.

->Overall, this LOPO framework closely mimics real-world clinical deployment, where models must generalize to entirely unseen patients rather than benefiting from subject-specific data leakage.

In [8]:
```
def lopo_cross_validation(eeg_data, labels, demographics, patient_ids,
                          input_shape, epochs=50, batch_size=32):
    n_patients = len(patient_ids)
    results = {
        'accuracy': [],
        'precision': [],
        'recall': [],
        'f1_score': [],
        'auc_roc': [],
        'confusion_matrices': []
    }

    # Create checkpoint directory if it doesn't exist
    checkpoint_dir = './model_checkpoints'
    os.makedirs(checkpoint_dir, exist_ok=True)
    print(f"Model checkpoints will be saved to: {checkpoint_dir}")

    print(f"\n{'='*80}")
    print(f"Starting LOPO Cross-Validation with {n_patients} patients")
    print(f"{'='*80}\n")

    # Iterate through each patient as test set
    for test_idx in range(n_patients):
        print(f"\n{'-'*80}")
        print(f"FOLD {test_idx + 1}/{n_patients}: Testing on Patient {patient_
        print(f"{'-'*80}")

        # Split data: Train on all except one patient
        train_indices = [i for i in range(n_patients) if i != test_idx]

        X_train = np.concatenate([eeg_data[i] for i in train_indices], axis=0)
        y_train = np.concatenate([labels[i] for i in train_indices], axis=0)
        demo_train = np.concatenate([np.tile(demographics[i], (len(labels[i]),
                                     for i in train_indices], axis=0)

        X_test = eeg_data[test_idx]
        y_test = labels[test_idx]
        demo_test = np.tile(demographics[test_idx], (len(y_test), 1))

        print(f"Training samples: {len(X_train)} | Testing samples: {len(X_tes
```

```
        print(f"Training seizures: {np.sum(y_train)} | Testing seizures: {np.s

        # Build Model
        model = build_cnn_bilstm_model(input_shape=input_shape,
                                       demographic_dim=demographics.shape[1])

        #Create multiple checkpoints

        # Checkpoint 1: Save best model based on validation ACCURACY
        checkpoint_acc = ModelCheckpoint(
            filepath=os.path.join(checkpoint_dir, f'best_model_fold_{test_idx+
            monitor='val_accuracy',
            mode='max',  # Maximize accuracy
            save_best_only=True,
            save_weights_only=False,  # Save entire model architecture + weigh
            verbose=1
        )

        # Checkpoint 2: Save best model based on validation LOSS
        checkpoint_loss = ModelCheckpoint(
            filepath=os.path.join(checkpoint_dir, f'best_model_fold_{test_idx+
            monitor='val_loss',
            mode='min',  # Minimize loss
            save_best_only=True,
            save_weights_only=False,
            verbose=1
        )

        # Checkpoint 3: Save model at every epoch
        checkpoint_epoch = ModelCheckpoint(
            filepath=os.path.join(checkpoint_dir, f'model_fold_{test_idx+1}_ep
            monitor='val_accuracy',
            mode='max',
            save_best_only=False,  # Save every epoch
            save_weights_only=False,
            period=5,  # Save every 5 epochs to avoid too many files
            verbose=0
        )

        # Early stopping: Stop training if validation loss doesn't improve and
        early_stop = EarlyStopping(
            monitor='val_loss',
            patience=10,
            restore_best_weights=True,
            verbose=1
        )

        # Reduce learning rate when validation loss plateaus
        reduce_lr = ReduceLROnPlateau(
            monitor='val_loss',
            factor=0.5,
            patience=5,
            min_lr=1e-6,
```

```
        verbose=1
    )

    # TensorBoard logging for GPU monitoring and training visualization
    tensorboard_callback = tf.keras.callbacks.TensorBoard(
        log_dir=os.path.join(checkpoint_dir, f'logs/fold_{test_idx+1}'),
        histogram_freq=1,
        write_graph=True,
        update_freq='epoch'
    )

    # Train Model with GPU Acceleration
    print(f"\nTraining model for fold {test_idx + 1} on GPU...")
    print("Monitoring: val_accuracy (↑) and val_loss (↓)")
    print(f"Checkpoints saving to: {checkpoint_dir}/")

    history = model.fit(
        [X_train, demo_train], y_train,
        validation_split=0.2,
        epochs=epochs,
        batch_size=batch_size,
        callbacks=[
            checkpoint_acc,      # Save best accuracy model
            checkpoint_loss,     # Save best loss model
            checkpoint_epoch,    # Save periodic checkpoints
            early_stop,          # Early stopping
            reduce_lr,           # Learning rate reduction
            tensorboard_callback # TensorBoard logging
        ],
        verbose=1,
        use_multiprocessing=True,  # Enable multiprocessing for data loadi
        workers=4  # Number of parallel workers
    )

    # Load Best Model for Evaluation
    print(f"\nLoading best model based on validation accuracy...")
    best_model_path = os.path.join(checkpoint_dir, f'best_model_fold_{test
    model = keras.models.load_model(best_model_path, custom_objects={'Atte

    #Evaluate on Test Patient
    print(f"\nEvaluating on test patient {patient_ids[test_idx]}...")
    y_pred_probs = model.predict([X_test, demo_test], verbose=0)
    y_pred = np.argmax(y_pred_probs, axis=1)

    # Calculate Metrics
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, zero_division=0)
    recall = recall_score(y_test, y_pred, zero_division=0)
    f1 = f1_score(y_test, y_pred, zero_division=0)

    # AUC-ROC (only if both classes present in test set)
    try:
        auc = roc_auc_score(y_test, y_pred_probs[:, 1])
```

```
        except:
            auc = 0.0

        cm = confusion_matrix(y_test, y_pred)

        # Store results
        results['accuracy'].append(accuracy)
        results['precision'].append(precision)
        results['recall'].append(recall)
        results['f1_score'].append(f1)
        results['auc_roc'].append(auc)
        results['confusion_matrices'].append(cm)

        # Print fold results
        print(f"\n{'='*40}")
        print(f"FOLD {test_idx + 1} RESULTS:")
        print(f"{'='*40}")
        print(f"Accuracy:  {accuracy*100:.2f}%")
        print(f"Precision: {precision*100:.2f}%")
        print(f"Recall:    {recall*100:.2f}%")
        print(f"F1-Score:  {f1*100:.2f}%")
        print(f"AUC-ROC:   {auc:.4f}")
        print(f"\nConfusion Matrix:")
        print(cm)
        print(f"\nBest model saved at: {best_model_path}")
        print(f"{'='*40}\n")

        # Clean up
        del model
        keras.backend.clear_session()

    print(f"\n{'='*80}")
    print(f"All model checkpoints saved in: {checkpoint_dir}/")
    print(f"TensorBoard logs saved in: {checkpoint_dir}/logs/")
    print(f"To view training logs, run: tensorboard --logdir={checkpoint_dir}/
    print(f"{'='*80}\n")

    return results
```

STEP 7: RESULTS ANALYSIS AND VISUALIZATION

This function summarizes and interprets the results obtained from **Leave-One-Patient-Out (LOPO) cross-validation**, providing both quantitative performance statistics and visual insights into model behavior across patients.

->It first computes the **mean and standard deviation** of key evaluation metrics—accuracy, precision, recall, F1-score, and AUC–ROC—across all folds, which reflects how consistently the model performs when tested on unseen patients.

->The function then checks whether a predefined **target accuracy of 95%** has been achieved, offering practical guidance if the target is not met by suggesting strategies such as additional data augmentation or hyperparameter tuning.

->To enhance interpretability, it generates multiple visualizations: a line plot showing how performance metrics vary across patients, a box plot illustrating the distribution and variability of metrics, an **average confusion matrix** that highlights overall classification behavior between seizure and non-seizure classes, a bar chart displaying **per-patient accuracy** to identify subjects for whom the model performs better or worse.

->These plots help detect inter-patient variability, potential bias, and robustness of the model.

->Finally, the function saves all visual outputs as a high-resolution image file, making it suitable for reporting, research documentation, and clinical analysis.

In [9]:
```
def analyze_lopo_results(results, patient_ids):
    print(f"\n{'='*80}")
    print("OVERALL LOPO CROSS-VALIDATION RESULTS")
    print(f"{'='*80}\n")

    # Calculate mean and std for each metric
    metrics = ['accuracy', 'precision', 'recall', 'f1_score', 'auc_roc']

    for metric in metrics:
        values = np.array(results[metric])
        mean_val = np.mean(values)
        std_val = np.std(values)
        print(f"{metric.upper():12s}: {mean_val*100:.2f}% ± {std_val*100:.2f}%
```

```
    print(f"\n{'='*80}")

    # Check if target accuracy achieved
    mean_accuracy = np.mean(results['accuracy'])
    if mean_accuracy >= 0.95:
        print(f"\n✓ TARGET ACHIEVED: Model accuracy ({mean_accuracy*100:.2f}%)
    else:
        print(f"\n✗ Target not met: Model accuracy ({mean_accuracy*100:.2f}%)
        print("   Consider: More data augmentation, hyperparameter tuning, or

    # Visualization
    fig, axes = plt.subplots(2, 2, figsize=(15, 12))

    # Plot 1: Metrics per fold
    ax1 = axes[0, 0]
    fold_indices = np.arange(1, len(patient_ids) + 1)
    ax1.plot(fold_indices, np.array(results['accuracy'])*100, 'o-', label='Acc
    ax1.plot(fold_indices, np.array(results['precision'])*100, 's-', label='Pr
    ax1.plot(fold_indices, np.array(results['recall'])*100, '^-', label='Recal
    ax1.plot(fold_indices, np.array(results['f1_score'])*100, 'd-', label='F1-
    ax1.axhline(y=95, color='r', linestyle='--', label='95% Target')
    ax1.set_xlabel('Fold (Patient)', fontsize=12)
    ax1.set_ylabel('Score (%)', fontsize=12)
    ax1.set_title('Performance Metrics per LOPO Fold', fontsize=14, fontweight
    ax1.legend()
    ax1.grid(True, alpha=0.3)

    # Plot 2: Box plot of metrics
    ax2 = axes[0, 1]
    box_data = [np.array(results[m])*100 for m in ['accuracy', 'precision', 'r
    bp = ax2.boxplot(box_data, labels=['Accuracy', 'Precision', 'Recall', 'F1-
    ax2.axhline(y=95, color='r', linestyle='--', label='95% Target')
    ax2.set_ylabel('Score (%)', fontsize=12)
    ax2.set_title('Distribution of Performance Metrics', fontsize=14, fontweig
    ax2.grid(True, alpha=0.3, axis='y')

    # Plot 3: Average confusion matrix
    ax3 = axes[1, 0]
    avg_cm = np.mean(results['confusion_matrices'], axis=0)
    sns.heatmap(avg_cm, annot=True, fmt='.1f', cmap='Blues', ax=ax3,
                xticklabels=['Non-Seizure', 'Seizure'],
                yticklabels=['Non-Seizure', 'Seizure'])
    ax3.set_title('Average Confusion Matrix', fontsize=14, fontweight='bold')
    ax3.set_ylabel('True Label', fontsize=12)
    ax3.set_xlabel('Predicted Label', fontsize=12)

    # Plot 4: Per-patient performance
    ax4 = axes[1, 1]
    patient_labels = [f'P{i+1}' for i in range(len(patient_ids))]
    x_pos = np.arange(len(patient_labels))
    ax4.bar(x_pos, np.array(results['accuracy'])*100, alpha=0.7, color='steelb
    ax4.axhline(y=95, color='r', linestyle='--', linewidth=2, label='95% Targe
    ax4.set_xlabel('Patient', fontsize=12)
```

```
    ax4.set_ylabel('Accuracy (%)', fontsize=12)
    ax4.set_title('Per-Patient Accuracy', fontsize=14, fontweight='bold')
    ax4.set_xticks(x_pos)
    ax4.set_xticklabels(patient_labels, rotation=45)
    ax4.legend()
    ax4.grid(True, alpha=0.3, axis='y')

    plt.tight_layout()
    plt.savefig('lopo_results.png', dpi=300, bbox_inches='tight')
    print(f"\nVisualization saved as 'lopo_results.png'")
    plt.show()
```

STEP 8: MAIN EXECUTION

Main execution function for seizure detection with LOPO cross-validation

In [10]:
```
# File and parameter configuration
DATA_DIR = "preprocessed_data_2"      # Directory with .npz files
DEMOGRAPHIC_FILE = "demographic_embeddings.npy"

USE_STFT = False
EPOCHS = 50
BATCH_SIZE = 32

# Reproducibility
np.random.seed(42)
tf.random.set_seed(42)
```

In [11]:
```
eeg_data, labels, demographics, patient_ids = load_preprocessed_data(
    DATA_DIR,
    DEMOGRAPHIC_FILE
)
```

```
Loading preprocessed EEG data and demographic embeddings:
Loading PN00-1.npz....
Loaded PN00-1.npz
Loading PN00-2.npz....
Loaded PN00-2.npz
Loading PN00-3.npz....
Loaded PN00-3.npz
Loading PN00-4.npz....
Loaded PN00-4.npz
Loading PN00-5.npz....
Loaded PN00-5.npz
Loading PN01-1.npz....
Loaded PN01-1.npz
Loading PN03-1.npz....
Loaded PN03-1.npz
Loading PN03-2.npz....
Loaded PN03-2.npz
Loading PN05-2.npz....
Loaded PN05-2.npz
Loading PN05-3.npz....
Loaded PN05-3.npz
Loading PN05-4.npz....
Loaded PN05-4.npz
Loading PN06-1.npz....
Loaded PN06-1.npz
Loading PN06-2.npz....
Loaded PN06-2.npz
Loading PN06-3.npz....
Loaded PN06-3.npz
Loading PN06-4.npz....
Loaded PN06-4.npz
Loading PN06-5.npz....
Loaded PN06-5.npz
Loading PN07-1.npz....
Loaded PN07-1.npz
Loading PN09-1.npz....
Loaded PN09-1.npz
Loading PN09-2.npz....
Loaded PN09-2.npz
Loading PN09-3.npz....
Loaded PN09-3.npz
Loading PN10-1.npz....
Loaded PN10-1.npz
Loading PN10-10.npz....
Loaded PN10-10.npz
Loading PN10-2.npz....
Loaded PN10-2.npz
Loading PN10-3.npz....
Loaded PN10-3.npz
Loading PN10-4.5.6.npz....
Loaded PN10-4.5.6.npz
Loading PN10-7.8.9.npz....
Loaded PN10-7.8.9.npz
Loading PN11-1.npz....
```

```
Loaded PN11-1.npz
Loading PN12-1.2.npz....
Loaded PN12-1.2.npz
Loading PN12-3.npz....
Loaded PN12-3.npz
Loading PN12-4.npz....
Loaded PN12-4.npz
Loading PN13-1.npz....
Loaded PN13-1.npz
Loading PN13-2.npz....
Loaded PN13-2.npz
Loading PN13-3.npz....
Loaded PN13-3.npz
Loading PN14-1.npz....
Loaded PN14-1.npz
Loading PN14-2.npz....
Loaded PN14-2.npz
Loading PN14-3.npz....
Loaded PN14-3.npz
Loading PN14-4.npz....
Loaded PN14-4.npz
Loading PN16-1.npz....
Loaded PN16-1.npz
Loading PN16-2.npz....
Loaded PN16-2.npz
Loading PN17-1.npz....
Loaded PN17-1.npz
Loading PN17-2.npz....
Loaded PN17-2.npz
Loaded data for 41 patients
Demographics shape: (14, 14)
```

In [12]:
```
if USE_STFT:
    eeg_data = prepare_data_with_stft(eeg_data, use_stft=True)
```

In [13]:
```
print("EEG data:", len(eeg_data))
print("Labels:", len(labels))
print("Demographics:", len(demographics))
print("Patient IDs:", len(patient_ids))
```

```
EEG data: 41
Labels: 41
Demographics: 14
Patient IDs: 41
```

In [17]:
```
# demographics: (14, D)
# demographic_ids: list of patient IDs used when embeddings were created

# YOU MUST have this — if not, see note below
print(type(demographic_ids), len(demographic_ids))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[17], line 5
      1 # demographics: (14, D)
      2 # demographic_ids: list of patient IDs used when embeddings were create
d
      3
      4 # YOU MUST have this — if not, see note below
----> 5 print(type(demographic_ids), len(demographic_ids))

NameError: name 'demographic_ids' is not defined
```

In [14]:
```
# Assumes all patients have same shape: (n_samples, n_channels, time_steps)
input_shape = eeg_data[0].shape[1:]
print("Input shape per sample:", input_shape)
```

```
Input shape per sample: (27, 1280)
```

In [16]:
```
# ---- FIX: align demographics with patients ----

D = demographics.shape[1]  # embedding dimension

aligned_demographics = []

for pid in patient_ids:
    if pid < len(demographics):
        aligned_demographics.append(demographics[pid])
    else:
        aligned_demographics.append(np.zeros(D, dtype=np.float32))

demographics = np.stack(aligned_demographics)

print("After alignment:")
print(len(eeg_data), len(labels), len(demographics), len(patient_ids))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[16], line 8
      5 aligned_demographics = []
      7 for pid in patient_ids:
----> 8     if pid < len(demographics):
      9         aligned_demographics.append(demographics[pid])
     10     else:

TypeError: '<' not supported between instances of 'str' and 'int'
```

In [15]:
```
results = lopo_cross_validation(
    eeg_data=eeg_data,
    labels=labels,
    demographics=demographics,
    patient_ids=patient_ids,
    input_shape=input_shape,
    epochs=EPOCHS,
    batch_size=BATCH_SIZE
```

```
)
```

```
Model checkpoints will be saved to: ./model_checkpoints

================================================================================
Starting LOPO Cross-Validation with 41 patients
================================================================================


________________________________________________________________________________
FOLD 1/41: Testing on Patient PN00-1
________________________________________________________________________________
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[15], line 1
----> 1 results = lopo_cross_validation(
      2     eeg_data=eeg_data,
      3     labels=labels,
      4     demographics=demographics,
      5     patient_ids=patient_ids,
      6     input_shape=input_shape,
      7     epochs=EPOCHS,
      8     batch_size=BATCH_SIZE
      9 )

Cell In[8], line 33, in lopo_cross_validation(eeg_data, labels, demographics, p
atient_ids, input_shape, epochs, batch_size)
     31 X_train = np.concatenate([eeg_data[i] for i in train_indices], axis=0)
     32 y_train = np.concatenate([labels[i] for i in train_indices], axis=0)
---> 33 demo_train = np.concatenate([np.tile(demographics[i], (len(labels[i]),
1))
     34                                for i in train_indices], axis=0)
     36 X_test = eeg_data[test_idx]
     37 y_test = labels[test_idx]

Cell In[8], line 33, in <listcomp>(.0)
     31 X_train = np.concatenate([eeg_data[i] for i in train_indices], axis=0)
     32 y_train = np.concatenate([labels[i] for i in train_indices], axis=0)
---> 33 demo_train = np.concatenate([np.tile(demographics[i], (len(labels[i]),
1))
     34                                for i in train_indices], axis=0)
     36 X_test = eeg_data[test_idx]
     37 y_test = labels[test_idx]

IndexError: index 14 is out of bounds for axis 0 with size 14
```

In [ ]:
```
analyze_lopo_results(results, patient_ids)

print("\n" + "=" * 80)
print("LOPO Cross-Validation Complete!")
```

```
print("=" * 80)
```