
Universidad Politécnica
de Madrid

**Escuela Técnica Superior de Ingenieros Informáticos**


European Master in Software Engineering

Master Thesis

# Monitoring Gait in Multiple Sclerosis Patients Using Sensor Data Analysis and Visual Reporting


Author: BALARAJ RAJA KUMAR


July, 2025


This Master Thesis has been deposited in ETSI Informáticos de la Universidad Politécnica de Madrid.

*Master Thesis*

*European Master in Software Engineering*

*Title:* Monitoring Gait in Multiple Sclerosis Patients Using Sensor Data Analysis and Visual Reporting

July / 2025

*Author:* BALARAJ RAJA KUMAR

*Supervisor:*

RAUL GUTIERREZ SANCHIS
Associate Professor in Economics

Universidad Politécnica de Madrid

Department of Industrial Organization, Business Administration and Statistics

School of Computer Engineering (ETSIINF-UPM)

Universidad Politécnica de Madrid

*Co-supervisor:*

JOAQUIN ORDIERES MERE
Full Professor

Universidad Politécnica de Madrid

Department of Industrial Organization, Business Administration and Statistics

School of Industrial Engineering (ETSI-UPM)

Universidad Politécnica de Madrid

# Abstract

Gait segmentation from instrumented sock sensors in multiple sclerosis is achieved using a transformer-based architecture with custom classification and reconstruction heads. Raw time-series data stored in InfluxDB are windowed at 5s, 7s, 10s, and 15s, balanced via SMOTE and SMOTEENN, and assessed with subject-level k-fold cross-validation. Transformer architecture with customized output heads and data balancing strategies is trained to classify walking and non-walking intervals, capturing dynamic temporal patterns in gait activity. To ensure generalizability, the model is evaluated using k-fold cross-validation. Results demonstrate the architecture's potential to robustly detect continuous walking episodes and contribute to more adaptive, passive gait monitoring solutions.

# Resumen

La segmentación de la marcha a partir de sensores integrados en calcetines en pacientes con esclerosis múltiple se logra mediante una arquitectura basada en transformadores con cabezales personalizados de clasificación y reconstrucción. Los datos de series temporales almacenados en InfluxDB se fragmentan en ventanas de 5s, 7s, 10s y 15s, se equilibran mediante SMOTE y SMOTEENN, y se evalúan mediante validación cruzada k-fold a nivel de sujeto. La arquitectura de transformador, junto con estrategias de balanceo de datos, se entrena para clasificar intervalos de caminar y de no caminar, capturando patrones temporales dinámicos de la marcha. Para garantizar la generalización, el modelo se somete a validación cruzada k-fold. Los resultados demuestran el potencial de esta solución para detectar de forma robusta episodios continuos de caminata y favorecer sistemas de monitoreo pasivo de la marcha más adaptativos.

Keywords: Transformer, Chunk sizes, Dual-head encoder–decoder, SMOTE, SMOTEENN, Time-window segmentation, Cross-Validation.

# Table of Contents

# List of Tables

# List of Figures

# 1 Introduction

Imagine Kumar, a young professional living with multiple sclerosis (MS), pacing his living room as he prepares for a video call. Some days, his steps feel effortless. Other days, each footfall is a careful negotiation between muscle fatigue and balance. Yet, when Kumar visits his neurologist every three months, those subtle daily struggles vanish behind the clinic door. What if we could capture the true story of his mobility not in snapshots, but continuously, passively, in his own environment?

Enter smart socks embedded with inertial measurement units (IMUs). These unassuming garments record accelerations, rotations and pressures at the foot in real time. The raw data streams hold hidden signatures of walking: the rhythmic peaks of heel strike, the gentle dips of toe-off, and the irregular stumbles that may precede a fall. But before any clinical insight can emerge, we must answer a deceptively simple question: when is Kumar walking, and when is he not?

Traditional machine-learning approaches convolutional networks fused with LSTMs can pick up short-term patterns but struggle with long-range dependencies and noisy, imbalanced free-living data. In contrast, the Transformer's self-attention mechanism lets every moment of a sensor window inform every other, learning global gait rhythms alongside local step dynamics. By combining attention with intelligent data balancing (SMOTE and SMOTEENN) and an auxiliary reconstruction objective, we craft models that not only classify walking vs. non-walking but also reinforce their own understanding of the signal's structure.

This thesis makes four key contributions:

i. To adapt the Transformer architecture to passive gait segmentation in MS, demonstrating better generalization than a CNN+LSTM benchmark.

ii. Systematically compare SMOTE and SMOTEENN for handling the extreme class imbalance of free-living gait data.

iii. To introduce a dual-head encoder–decoder that jointly learns to classify and reconstruct, using uncertainty-based loss weighting to harmonize both goals.

iv. To deliver an end-to-end, reproducible pipeline open-source code included that transforms raw smart-sock streams into clinically meaningful walking-episode metrics.

By the end of this journey, you'll see how self-attention can illuminate the hidden patterns of human gait and open new horizons for continuous, personalized mobility monitoring in multiple sclerosis.

# 2 State of the art

## 2.1 Data Extraction

Data extraction comprises two phases: first, a manual annotation step to identify walking versus non-walking intervals, and second, an automated pipeline that reads those annotations and pulls the corresponding raw sensor data from InfluxDB.

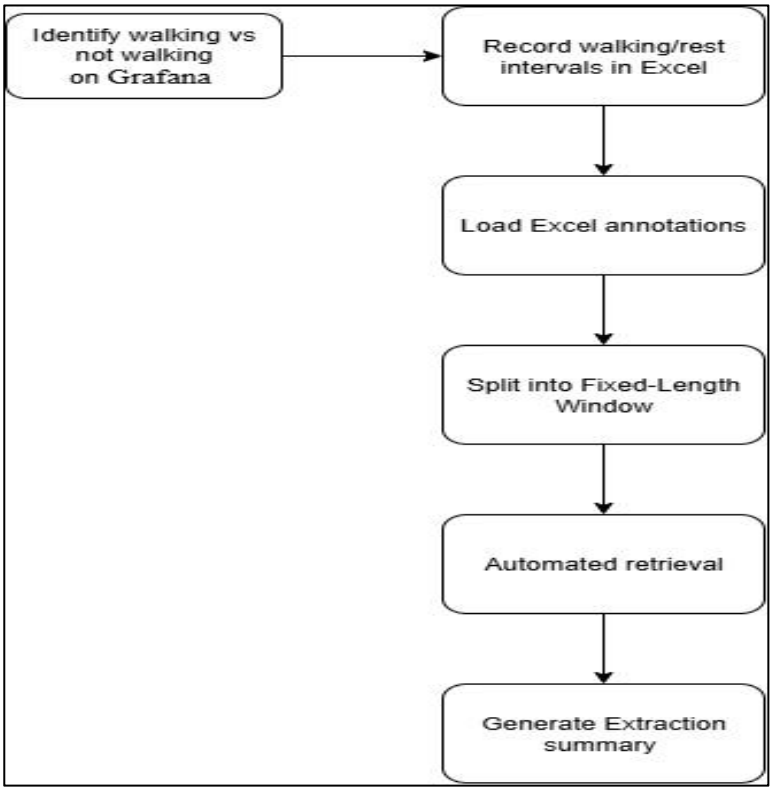


Figure 2.1.1 Data Extraction Pipeline

### 2.1.1 Manual Annotation via Grafana

Before any code runs, each reference ID (a token printed on the instrumented sock) must be inspected in the Grafana dashboard as follow:

i. Open the GAIT_tokens dashboard in Grafana.
ii. Select one 'Referencia' from the upper-left list and choose a broad time window (e.g. January 1, 2024 to today).
iii. Zoom in around the signal peaks and troughs. Continuous high-amplitude bands indicate walking; flat or sparse readings indicate rest.
iv. Record each walking or non-walking segment's start and end timestamps.

These entries go into an Excel file with columns:

- Reference (e.g. `ABC123`)
- datefrom (e.g. `2024-04-24 16:41:20`)
- dateuntil (e.g. `2024-04-24 16:41:35`)
- mov_type (`walking` or `not walking`)

This excel sheet is your only manual input which drives the automated retrieval.

### 2.1.2 Automated Retrieval Pipeline

Once the Excel sheet of annotations is in place, the retrieval pipeline runs end-to-end with a single command. Under the hood, it performs five coordinated steps: loading, windowing, querying, saving and summarizing each encapsulated in a concise function.

#### 2.1.2.1 Loading the Annotations

The pipeline begins by reading the manual annotations into memory. A function called 'load_data()' uses pandas to open the file, select exactly four columns and convert the start/end timestamps into true datetime objects. If anything is missing if to say the sheet name is wrong or a column typo occurs—the function prints an error and halts further processing.

```
def load_data(input_file):
        df = pd.read_excel(input_file)[
            ['Reference','datefrom','dateuntil','mov_type']
        ]
        df['datefrom']  = pd.to_datetime(df['datefrom'])
        df['dateuntil'] = pd.to_datetime(df['dateuntil'])
        return df
```

#### 2.1.2.2 Splitting into Fixed-Length Windows

For each session row, the pipeline calls create_chunks(). Starting at the datefrom timestamp, it marches forward in exact increments of the user-specified duration (for example, 10 s). Each step yields a (start, end) pair; any trailing fragment shorter than one full window is silently dropped. This ensures uniform window lengths and avoids the need for padding or truncation later on.

```
    def create_chunks(row, chunk_secs):
        duration = timedelta(seconds=chunk_secs)
        t_start, t_end = row['datefrom'], row['dateuntil']
        windows = []
        while t_start + duration <= t_end:
            windows.append((t_start, t_start + duration))
            t_start += duration
        return windows
```

#### 2.1.2.3 Querying InfluxDB for Sensor Data

Each time window and each leg (“Left”, “Right”) triggers a separate database query. The function extract_data() hands off the (start, end, reference, leg) tuple to a pre-configured InfluxDB client. After retrieving raw sensor rows, it adjusts the ‘_time’ column to GMT + 1 and strips timezone metadata so downstream code sees plain timestamps. Finally, it sorts the data in reverse chronological order, ensuring the newest samples come first.

```
def extract_data(client, start, end, ref, leg):
    df = client.query_data(start, end, qtok=ref, pie=leg)
    df['_time'] = (
        df['_time']
          .dt.tz_convert('Etc/GMT-1')
          .dt.tz_localize(None)
    )
    return df.sort_values('_time', ascending=False)
```

#### 2.1.2.4 Saving Windows and Updating Statistics

Every successfully retrieved window is written to its own ‘.xlsx’ file. The filename embeds all necessary metadata:

<Reference>+<mov_type>+YYYY-MM-DD_HH-MM-SS+YYYY-MM-DD_HH-MM-SS+<leg>.xlsx

At the same time, a simple ‘update_summary()’ function increments counters for total windows, dropped fragments and cumulative walking vs. non-walking durations. These counters live in memory until the very end.

```
def save_and_update(df_chunk, out_path, mov_type, summary, chunk_secs):
    df_chunk.to_excel(out_path, index=False)
    summary['extracted'] += 1
    if mov_type == 'walking':
        summary['walk_sec'] += chunk_secs; summary['walk_cnt'] += 1
    else:
        summary['rest_sec'] += chunk_secs; summary['rest_cnt'] += 1
```

#### 2.1.2.5 Executing the Extraction Pipeline

A lightweight 'main()' function parses four command-line arguments, input Excel, output folder, chunk duration and InfluxDB config and then sequences the above steps. In pseudocode:

```
    df_meta = load_data(input_file)
  for each row in df_meta:
      windows = create_chunks(row, chunk_secs)
      if not windows:
          summary['dropped'] += 1
      for start,end in windows:
          for leg in ('Left','Right'):
              df_chunk = extract_data(client, start, end,
row['Reference'], leg)
              if df_chunk is not None:
                  filename = build_filename(row, start, end, leg)
                  save_and_update(df_chunk, os.path.join(output_dir,
filename), row['mov_type'], summary, chunk_secs)
  print_summary(summary)
```

You launch the entire process with one command in Bash:

```
python extract_data.py \
    --input gait_annotations.xlsx \
    --output extracted_chunks/ \
    --duration 10 \
    --path config_db.yaml
```

On ten reference IDs, two legs per ID and an average of five windows each, this single command generates approximately hundreds of `.xlsx` files no manual looping or timestamp arithmetic required.

By structuring each step into a clear function and using straightforward naming conventions, this automated retrieval pipeline remains accessible to anyone, even without deep programming experience. Each function handles exactly one responsibility, errors are caught early, and a final summary confirms that the extracted dataset is complete and balanced before moving on to preprocessing.

## 2.2 Transformer Architecture Design

### 2.2.1 Data Preprocessing

Before entering the Transformer, raw sensor windows each of fixed length (e.g. 256 timesteps × 6 channels) must be normalised and structured into a 3D tensor of shape (batch, seq_len, features). The preprocessing pipeline applies sliding-window segmentation (stride<window size) to capture overlapping context, computes Euclidean norms for accelerometer, gyroscope and magnetometer triads, and scales every feature channel with a robust scaler. Finally, classification labels are one-hot encoded (and, for the dual-head variant, reconstruction targets are taken as the first timestep of each window). This ensures that the model sees uniformly sized, zero-mean, unit-variance inputs with clean class/reconstruction targets.

#### 2.2.1.1 Scaling Technique: Robust scaler

To ensure all input features are on a comparable scale and resilient to outliers, Robust Scaling is applied channel-wise across the training data. Unlike standard scaling which centers around the mean and scales to unit variance the robust scaler subtracts the median and divides by the interquartile range (IQR), formally:

$$\text{scaled\_value} = \frac{x - \text{median}}{\text{IQR}}$$

This method is particularly effective for wearable sensor data, which often contains occasional spikes, dropout artifacts or subject-specific deviations. By using percentile-based statistics, Robust Scaling preserves the core signal structure while dampening the influence of anomalies, leading to more stable and generalisable training on sensor data (Pedregosa et al., 2011).

### 2.2.2 Model Architecture and Mechanism

The core Transformer ingests each preprocessed sensor window, an array of shape timesteps, features and first projects it into a d_model-dimensional embedding space. Fixed sinusoidal positional encodings are then added so the model recognises the order of readings without learning extra parameters.

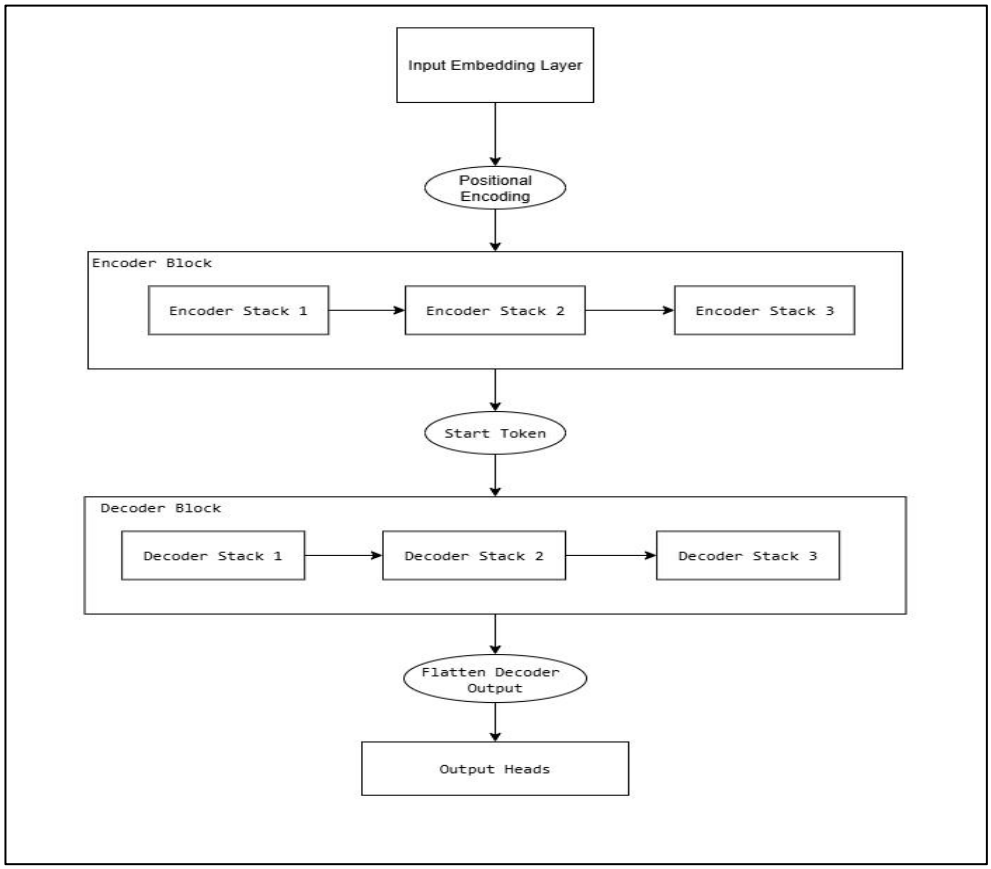


Figure 2.2.2: Encoder-Decoder Transformer Architecture

The encoder comprises three identical layers. Each layer applies multi-head self-attention (splitting the d_model embedding into parallel attention heads), follows with a residual connection and layer normalization, then passes through a two-layer feed-forward network. This repeated structure lets every time step attend to all others in parallel while preserving training stability via normalization and shortcuts.

The decoder likewise stacks three blocks but begins each with masked self-attention to enforce causality. It then performs encoder–decoder attention to align its states with the encoder's outputs, before running the same feed-forward and normalization sublayers. Finally, the flattened decoder embeddings split into two heads: a softmax classification head for walking vs. non-walking, and a linear reconstruction head predicting the raw sensor values at the first timestep. In the dual-head variant, cross-entropy and Huber losses are combined with learned uncertainty weights, enabling the model to balance its discriminative and reconstructive objectives.

#### 2.2.2.1 Input Embedding

Each sensor window is a matrix $X \in \mathrm{R}^{T \times F}$, where $T$ is the number of timesteps and $F$ the feature dimension. A learned linear layer projects these raw values into a common embedding space of dimension $d_{\text{model}}$:

$$E = X W_e + b_e$$

This ensures that every timestep is represented by a $(d_{\text{model}})$-vector, ready for attention operations(Vaswani et al., 2017).

#### 2.2.2.2 Positional Encoding

Transformers cannot natively infer sequence order, so we add a fixed sinusoidal bias to each embedding. For position $p$ and dimension $i$,

$$\mathrm{PE}(p, 2i) = \sin\left(p/10000^{2i/d_{\text{model}}}\right), \quad \mathrm{PE}(p, 2i+1) = \cos\left(p/10000^{2i/d_{\text{model}}}\right).$$

By summing *PE* with *E*, the model learns relative and absolute positions without learning extra parameters (Vaswani et al., 2017).

#### 2.2.2.3 Encoder Block

We stack three identical encoder blocks. Each block first applies multi-head self-attention: it splits $E$ into $h$ parallel subspaces, computes scaled dot-product attention across all $T$ positions in each head, then concatenates them back(Vaswani et al., 2017). A residual connection adds the block's input to its attention output, preserving lower-level features and easing gradient flow. Layer normalization follows, stabilising activations. Finally, a two-layer position-wise feed-forward network with ReLU expands and contracts the representation, capturing non-linear temporal patterns

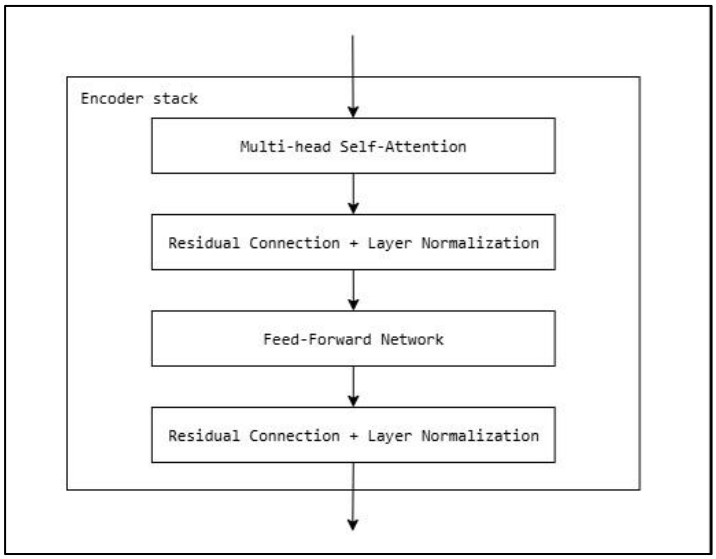


Figure 2.2.2.3: Encoder Block

*2.2.2.3.1 Multi-head self-attention*

Multi-head self-attention computes attention scores in parallel subspaces so the model can focus on different aspects of gait simultaneously. Each head projects the input into queries, keys and values, calculates scaled dot-product attention across all timesteps, then recombines the results. This mechanism learns which parts of the time window matter most for each feature and captures correlations across the entire sequence.

This sublayer lets each timestep attend to every other via $h$ parallel attention heads. The input $H \in \mathrm{R}^{T \times d_{\text{model}}}$ is projected into queries $Q$, keys $K$ and values $V$. For each head, attention is computed as

$$\text{softmax}\left(QK^T/\sqrt{d_k}\right)V,$$

whered $d_k = d_{\text{model}}/h$. Dividing by $\sqrt{d_k}$ prevents large dot-product values that would shrink gradients (Vaswani et al., 2017). The h head outputs are concatenated and projected back, enabling the block to focus on multiple aspects of gait dynamics simultaneously.

*2.2.2.3.2 Residual connections*

Immediately after attention, residual connections add the original block input to its output, ensuring that low-level information is never lost as the network deepens. By creating a shortcut around each sublayer, they mitigate vanishing gradients and make it easier for the model to learn identity mappings where appropriate. This guarantees that the encoder can refine embeddings without erasing the foundational sensor patterns.

*2.2.2.3.3 Layer normalization*

Following each residual addition, layer normalisation standardises activations across the feature dimension for each time step, centring them around zero mean and unit variance. Applied after every residual addition, it reduces internal covariate shift and stabilises training, allowing higher learning rates and faster convergence (Ba et al., 2016).

*2.2.2.3.4 Feed-forward network*

The feed-forward network applies two linear transformations with a ReLU in between, independently for each timestep. This position-wise sublayer expands the model's representational capacity and introduces non-linearity, enabling the encoder to capture complex combinations of temporal features learned by the attention mechanism.

*2.2.2.3.5 StartToken Layer*

A learned start token a single $d_{\text{model}}$-dimensional vector, is prepended to every batch. Although it serves as the decoder's first input, it lives here conceptually to signal the beginning of each sequence(Vaswani et al., 2017)., allowing the decoder to generate its initial attention queries without raw sensor data.

### 2.2.2.4 Decoder Block

The decoder also comprises three blocks but enforces autoregressive constraints. Each block begins with masked self-attention, preventing any timestep from attending to future positions. Next comes encoder–decoder attention: queries derive from the decoder's masked output, while keys and values come from the encoder's final layer, aligning decoding with the encoded context. Residual connections and layer normalization follow each attention and feed-forward sublayer, mirroring the encoder's structure.

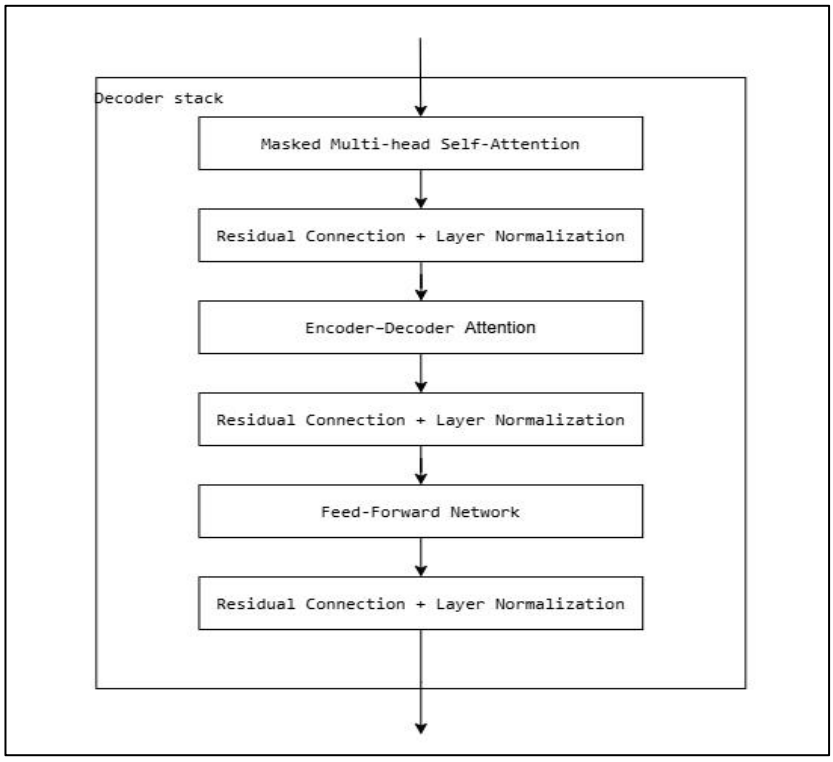


Figure 2.2.2.4: Decoder Block

*2.2.2.4.1 Masked self-attention*

Masked self-attention is identical to multi-head attention except it applies a causal mask so that each timestep can only attend to itself and previous positions. This autoregressive constraint ensures that predictions for time t do not rely on future information.

*2.2.2.4.2 Encoder–Decoder attention*

In this sublayer, the decoder's queries come from its own previous masked output, while keys and values derive from the encoder's final embeddings. This cross-attention aligns decoding with the full input sequence, allowing each output position to leverage encoder-learned temporal patterns (Luong, Pham, & Manning, 2015).

*2.2.2.4.3 Residual connections*

As in the encoder, each each attention and feed-forward sublayer adds its input to its output, preserving earlier representations and supporting stable gradient flow through the decoder's depth.

*2.2.2.4.4 Layer normalization*

Applied after every residual addition in the decoder, layer normalisation ensures numerical stability and consistent activation distributions, facilitating efficient training.

*2.2.2.4.5 Feed-forward network.*

The decoder's feed-forward sublayer matches the encoder's: two linear transforms with a ReLU activation, applied position-wise. It refines the combined attention outputs into higher-level features before they reach the final heads.

### 2.2.2.5 Output Heads

After the decoder's third block, the final time-step embeddings are flattened into a single vector per window and fed into two distinct prediction branches. This design enables the network to perform both discrimination and reconstruction in one pass, enriching its internal representations.

*2.2.2.5.1 Classification*

The classification head applies a dense projection of the decoder's flattened embedding into two logits, immediately followed by a softmax activation. This produces a probability distribution over the two classes: walking versus non-walking. This layer is identical to the single-head Transformer used in the classification-only experiment. Reusing the same dense-softmax structure ensures a fair comparison between the dual-head and classification-only models.

*2.2.2.5.2 Reconstruction(dual-head)*

Parallel to classification, the reconstruction head consists of a linear layer that maps the embedding back to the original feature dimension *F*, predicting the raw sensor values at the first timestep of each window. By forcing the model to reconstruct actual sensor signals, this auxiliary objective encourages learning of fine-grained temporal patterns that pure classification might ignore. In practice, this leads to more generalisable feature extractors, since accurate reconstruction demands capturing subtle dynamics in the gait waveform.

### 2.2.2.6 Loss Functions

Training optimises a composite loss that combines the classification and reconstruction objectives. Each term specialises in one task: the first encourages correct class labels, the second encourages accurate signal prediction. The two losses are balanced either equally in the classification-only model or via learned weights in the dual-head variant to guide the network toward representations that serve both tasks.

*2.2.2.6.1 Categorical cross-entropy*

For the classification head, we use

$$L_{cls} = -\sum_{k=1}^{2} y_k ln(\hat{y}_k),$$

Where $y$ is the one-hot ground truth and $\hat{y}$ the predicted class probabilities. This loss strongly penalises confident but incorrect predictions, sharpening the model's discriminative capability (Goodfellow, Bengio, & Courville, 2016).

*2.2.2.6.2 Huber loss*

The reconstruction head is trained with Huber loss, defined for an error $e = r - \hat{r}$ and threshold $\delta$ as

$$\mathcal{L}_{\text{rec}} = \begin{cases} \frac{1}{2}e^2, & |e| \le \delta, \\ \delta\,|e| - \frac{1}{2}\delta^2, & |e| > \delta. \end{cases}$$

This formulation is robust to outliers, quadratic when errors are small, linear when they are large making it well-suited to sensor data that may contain occasional spikes or noise (Huber, 1964).

### 2.2.3 Transformer Implementation (Code Overview)

#### 2.2.3.1 Scaler and Input Preparation

This step reshapes and robustly scales raw windows before they enter the model.

```
# X: np.ndarray, shape = (n_windows, timesteps, n_features)
flat = X.reshape(-1, X.shape[-1])         # (n_windows*timesteps, n_features)
scaler = RobustScaler()
flat_scaled = scaler.fit_transform(flat)  # centre by median, scale by IQR
X_scaled = flat_scaled.reshape(X.shape)   # back to (n_windows, timesteps, n_features)
```

#### 2.2.3.2 PositionalEncoding Layer

Without recurrence, the model needs a sense of order. Added fixed sinusoidal signals so attention can recognise timestep order:

```
class PositionalEncoding(tf.keras.layers.Layer):
    def __init__(self, length, d_model):
        super().__init__()
        pos = tf.range(length, dtype=tf.float32)[:, None]
        i   = tf.range(d_model, dtype=tf.float32)[None, :]
        angle = pos / tf.pow(10000.0, 2*(i//2)/d_model)
        pe = tf.where(tf.math.floormod(i, 2)==0,
                      tf.sin(angle), tf.cos(angle))
        self.pe = pe[None]  # shape (1, length, d_model)

    def call(self, x):
        return x + self.pe[:, :tf.shape(x)[1], :]
```

#### 2.2.3.3 Encoder Block

Each block applies self-attention, residuals, normalization and a feed-forward layer:

```
class EncoderBlock(tf.keras.layers.Layer):
    def call(self, x, training=False):
        attn = self.mha(x, x, x, training=training)
        x1   = self.ln1(x + self.do1(attn, training=training))
        ffn = self.ffn(x1)
        x2   = self.ln2(x1 + self.do2(ffn, training=training))
        return x2
```

- Multi-Head Self-Attention divides the embedding into parallel "heads," each learning distinct attention patterns—capturing gait cycles at multiple scales.
- Residual Connections (x + …) preserve raw inputs, preventing degradation if a layer's newly learned features prove unhelpful.
- Layer Normalization stabilises activations and gradients.
- Feed-Forward Network adds non-linear transformations via a small MLP, enriching representation power.

#### 2.2.3.4 Decoder Block

Mirrors the encoder but adds causal masking and encoder–decoder attention:

```
# Masked self-attention
attn1 = mha1(y, y, y, training=training, use_causal_mask=True)
y1    = layer_norm(y + dropout(attn1, training=training))
# Encoder–decoder attention
attn2 = mha2(y1, enc_out, enc_out, training=training)
y2    = layer_norm(y1 + dropout(attn2, training=training))
# Feed-forward
ffn_out = ffn(y2)
y3      = layer_norm(y2 + dropout(ffn_out, training=training))
```

At each layer:

- Masked self-attention prevents positions from attending to future tokens, enforcing temporal causality if needed.
- Encoder–decoder attention aligns decoder queries with encoder keys and values, allowing precise mapping of reconstructed or classified outputs to input features.
- The same residual + layer-norm + feed-forward sequence follows.

#### 2.2.3.5 Output Heads & Loss

Flattens decoder output, applies two heads, then computes combined loss:

```
# Classification head
cls_out = Dense(NUM_CLASSES, activation='softmax')(flat_emb)
# Reconstruction head (dual-head only)
rec_out = Dense(num_feats, activation='linear')(flat_emb)
```

Classification uses cross-entropy:

```
ce_loss = tf.keras.losses.CategoricalCrossentropy()(y_true_cls, cls_out)
```

Reconstruction uses Huber:

```
h_loss = tf.keras.losses.Huber()(y_true_rec, rec_out)
```

Dual-head combines them with learnable uncertainty weights:

```
inv_s1 = tf.exp(-s1); inv_s2 = tf.exp(-s2)
loss = inv_s1 * ce_loss + inv_s2 * h_loss + (s1 + s2)
```

Softmax gives a probability distribution over classes, while linear outputs recreate signal values. Heteroscedastic loss weighting (`s1`, `s2`) balances tasks automatically, encouraging features that both discriminate gait states and retain raw signal integrity.

## 2.3 Balancing techniques

Real-world gait recordings exhibit a skewed distribution of walking vs. non-walking windows (e.g. roughly 570 walking vs.1432 non-walking samples at 7s chunks), our raw gait dataset contains far fewer walking segments than non-walking ones. Training a classifier on such data without intervention leads it to favour the majority class. To prevent this, we rebalance the training set to roughly a 1:1 ratio, then feed the balanced data into both our dual-head and classification-only models.

### 2.3.1 Vanilla SMOTE Oversampling

The Synthetic Minority Over-sampling Technique (SMOTE) creates new minority-class examples by interpolating between real samples. For each minority feature vector $x_i$ SMOTE selects one of its $k$ nearest minority neighbours $x_n$ , SMOTE draws a random weight $\lambda$ from the from the uniform interval [0,1] and computes a synthetic point

$$x_{\text{new}} = x_i + \lambda\,(x_n - x_i)$$

This produces new feature vectors that lie along the line segments connecting existing minority samples.

Why it matters, simple duplication of rare examples can cause the model to memorise them, harming generalisation. SMOTE's interpolation produces diverse yet plausible samples spread throughout the minority feature space. As a baseline, it reliably boosts recall on the underrepresented class with minimal algorithmic complexity(Chawla et al., 2017).

We reshape each window into a flat feature vector, apply SMOTE with a 1 : 1 sampling strategy, then restore the window shape:

```
# X_flat: (n_samples, feature_dim), y: (n_samples,)
sm = SMOTE(sampling_strategy=1.0, k_neighbors=5, random_state=42)
X_balanced, y_balanced = sm.fit_resample(X_flat, y)
# reshape X_balanced back to (n_windows, timesteps, n_features) as needed
```

Here, X_flat has shape (n_samples, feature_dim) and y holds binary labels. This call ensures that after resampling, walking and non-walking samples appear in equal numbers. We use the same resampled set for both dual-head and classification-only models, guaranteeing identical class distributions.

The exact same 1:1 SMOTE oversampling strategy was used for both the dual-head and classification-only models. In the latter, the synthetic samples were simply passed to the classification head without requiring reconstruction targets. This ensured a consistent training distribution while preserving model-specific outputs.

### 2.3.2 Integration into Dual-Head vs. Classification-Only

Though the resampling call is identical, the downstream use differs slightly, after X_balanced, y_balanced are produced by SMOTE, we split them into inputs for each head:

- Dual-Head Model: All SMOTE-generated windows carry class labels into the shared encoder; labels y_balanced drive the classifier. Only original (non-synthetic) windows supply targets for the reconstruction head, so reconstruction targets remain the first-timestep readings of the original

windows. This preserves the integrity of the signal-prediction task while still balancing classification training.

- Classification-Only Model: The oversampled set is passed entirely to the single softmax head; y_balanced alone determines the cross-entropy loss. Since no reconstruction targets exist, every window, real or synthetic contributes to classification loss.

This unified SMOTE strategy keeps input distributions consistent, isolating the effect of adding or removing the reconstruction objective on model performance.

### 2.3.3 SMOTEENN classification

SMOTEENN combines SMOTE's synthetic-example generation with the Edited Nearest Neighbors (ENN) cleaning step. After generating synthetic new minority points, ENN examines each sample (real or synthetic) and its $k$ nearest neighbours. If more than half of those neighbours carry a different label, the sample is deemed ambiguous and removed. If more than half have a different label, for sample $x_i$ with label $y_i$, it's discarded when:

$$\sum_{j=1}^{k} \mathbf{1}(y_{nj} \neq y_i) > \frac{k}{2} \Rightarrow discard\ x_i$$

This two-stage method both boosts minority density and prunes noisy or borderline instances, yielding cleaner class boundaries and often higher minority recall than SMOTE alone.

We call SMOTEENN in a single step, specifying the same 1:1 oversampling and letting ENN run with its default neighbour count:

```
sm = SMOTE(sampling_strategy=1.0, k_neighbors=5, random_state=42)

smote_enn = SMOTEENN(smote=sm, enn=None, random_state=42)

X_bal, y_bal = smote_enn.fit_resample(X_flat, y)
```

The result X_bal, y_bal reflects a balanced, de-noised dataset. We use SMOTEENN only for the classification-only experiments, since the reconstruction head in the dual-head model benefits more from retaining full signal variability.

## 2.4 Transformer Training

The training procedure combines data preparation, balancing, and model fitting into a single, reproducible workflow. All four window-length datasets are processed identically, and subject-level cross-validation ensures fair evaluation without data leakage.

### 2.4.1 Training Setup

- Window-length datasets: four fixed lengths (5s, 7s, 10s, 15s) are extracted via sliding windows and structured into tensors of shape `(batch_size, sequence_length, feature_dim)`.
- Subject-level cross-validation : employ a 3-fold split at the subject level, ensuring that all data from any given subject appears in exactly one fold. Folds rotate through "train," "validation," and "test" partitions so that each subject serves as unseen test data once. This split logic is implemented in run_transformer.py and invoked identically across all experiments.
- Balancing strategies
    i. Dual-head models apply SMOTE oversampling to achieve a 1 : 1 class ratio; all windows enter the encoder for classification, but only the original samples supply targets for reconstruction.
    ii. Classification-only models use either SMOTE or SMOTEENN to balance walking versus non-walking; the entire balanced set then trains the single softmax head.
- Per-fold pipeline
    i. Load raw windows for the current fold's training subjects.
    ii. Scale each feature channel using the Robust Scaler.
    iii. Apply the selected balancing method (SMOTE or SMOTEENN).
    iv. Reshape data and labels for the Transformer's input heads.
    v. Instantiate the model variant (classification-only or dual-head) and commence training.

### 2.4.2 Hyperparameters & Schedule

Key training hyperparameters, the learning-rate schedule, and the initialization of the dual-head loss weights are summarized below.

| **Hyperparameter** | **Value** | |
|---|---|---|
| Epochs | 120(40 fold per fold) | Validation loss plateaus by ~30; 10 extra epochs as safety margin |
| Batch size | 64 | Fits on a single GPU with smooth gradient estimates |
| Encoder layers | 3 | Feature-shift diagnostics plateau after layer 3, deeper models overfit |
| Decoder layers | 3 | Symmetric depth enables stable dual-head outputs |
| Attention heads | 4 | 128 ÷ 4 heads gives multi-scale focus within each window |
| Model dimension (d_model) | 128 | Enough capacity to model gait without blowing up GPU memory |
| Feed-forward dimension (d_ff) | 4* d_model | Four-times d_model for sufficient nonlinear expansion |
| Dropout rate | 0.1 | Regularises deep self-attention |
| Initial learning rate | $3 \times 10^{-4}$ | Standard for Adam on small-to-medium Transformer models |
| Random seed | 42 | Ensures reproducible splits and balancing |

Table 2.4.2.1: Training hyperparameters

**Learning-rate schedule**: learning rates are managed via TensorFlow's CosineDecayRestarts scheduler, which repeatedly anneals the learning rate from its initial value down to a small floor and then restarts the cycle. This approach helps the optimizer escape shallow minima and encourages better convergence in later epochs. In our setup:

```
LR      = 3e-4                        # Initial LR
LR_SCHED=tf.keras.optimizers.schedules.CosineDecayRestarts(
           initial_learning_rate=LR,
           first_decay_steps=4000,
           t_mul=2.0,   # each cycle twice as long as the previous
           m_mul=0.5    # peak LR remains constant
         )
```

**Dual-head uncertainty weights**: for the reconstruction and classification losses, we follow (Kendall et al, 2018). uncertainty weighting approach.

$$\mathcal{L}_{\text{cls}} = \text{CrossEntropy}(y, \hat{y}), \quad \mathcal{L}_{\text{rec}} = \text{Huber}(x, \hat{x})$$

Two learnable scalars, $s_1$ and $s_2$, are initialized to zero (i.e., $\exp(-s_i) = 1$) so that both losses contribute equally at the start. The total loss is

$$\mathcal{L}_{\text{total}} = \exp(-s_1)\,\mathcal{L}_{\text{cls}} + \exp(-s_2)\,\mathcal{L}_{\text{rec}} + (s_1 + s_2).$$

During training, $s_1$ and $s_2$ adapt to balance the two objectives automatically. This formulation avoids manual loss-weight tuning and has been shown to yield superior multi-task performance.

### 2.4.3 Training flow and Model Instantiation

#### 2.4.3.1 Build and Compile the Model

The first step constructs the Transformer architecture by invoking a factory function with the chosen window length and feature count. For example:

```
model = build_transformer(seq_len=TIMESTEPS, num_feats=NUM_FEATURES)
```

Here, 'build_transformer' wires together the input projection, positional encodings, encoder and decoder stacks, and the dual heads. By decoupling model creation into a single call, any change to layer depth or embedding dimension propagates cleanly across all experiments.

Once the model object exists, an Adam optimiser is attached, driven by a CosineDecayRestarts schedule. This scheduler smoothly lowers the learning rate from its initial value down to near zero, then restarts the cycle at progressively longer intervals—helping the optimiser escape sub-optimal minima. In code:

```
optimizer = tf.keras.optimizers.Adam(learning_rate=LR_SCHED)
model.compile(
    optimizer=optimizer,
    loss=DualLossConcat(NUM_CLASSES),
    metrics=[ClassAccuracy(NUM_CLASSES)]
)
```

The 'DualLossConcat' loss wrapper balances classification and reconstruction via two trainable parameters, while 'ClassAccuracy' isolates performance on the softmax outputs. Together, they ensure that both objectives are optimised in harmony.

Before any training commences, global seeds are fixed for TensorFlow and NumPy to guarantee bit-level reproducibility:

```
tf.random.set_seed(SEED)
np.random.seed(SEED)
```

Immediately after compilation, the model's initial weights are snapshot:

```
initial_weights = model.get_weights()
```

This snapshot is reused at the start of each cross-validation fold, so every fold begins from the identical parameter state. By resetting weights rather than retraining from scratch unpredictably, we eliminate random initialisation as a confounding factor in later performance comparisons.

#### 2.4.3.2 Subject-Level Cross-Validation Setup

A three-fold split is created so that all windows from any single subject appear together in one partition, preventing data leakage across training and validation sets. Integer class labels are derived from the one-hot arrays with:

```
y_labels = np.argmax(y_class, axis=1)
```

These labels guide the fold assignment, ensuring that each fold maintains the original class proportion while grouping by subject. Stratified grouping is achieved via:

```
from sklearn.model_selection import StratifiedKFold


kf = StratifiedKFold(n_splits=3, shuffle=True, random_state=SEED)
```

Shuffling adds randomness to fold composition but the fixed seed guarantees repeatability.

Before entering the fold loop, the model's initial parameters are snapshot once so that every fold begins from the exact same state:

```
initial_weights = model.get_weights()
```

Inside the loop, train and validation indices index directly into the preprocessed tensors:

```
for train_idx, val_idx in kf.split(X, y_labels):
    X_train, X_val = X[train_idx], X[val_idx]
    y_train, y_val = y_concat[train_idx], y_concat[val_idx]
    model.set_weights(initial_weights)
```

Resetting the weights at each iteration removes any carry-over learning and ensures that performance differences across folds derive solely from data partitioning, not from random initialisation.

#### 2.4.3.3 Balancing Integration per Fold

Once the training and validation splits are extracted for a given fold, the minority oversampling procedure transforms the training set into a balanced collection of windows and labels. Because both SMOTE and SMOTEENN expect feature matrices of shape (n_samples, n_features), each fold's 3D tensor must first be flattened into a two-dimensional matrix so that both SMOTE and SMOTEENN can operate on feature vectors. Suppose X_train has shape (n,T, F) and the one-hot labels y_labels yield integer class indices via:

```
n, T, F = X_train.shape
X_flat   = X_train.reshape(n, T * F)
y_labels = np.argmax(y_train[:, :NUM_CLASSES], axis=1)
```

Here, T * F collapses the time and feature dimensions so that each window becomes a single feature vector. The integer array y_labels supplies the class indices required by the sampler.

For experiments using vanilla SMOTE, the sampler is invoked as follows:

```
from imblearn.over_sampling import SMOTE


smote = SMOTE(
    sampling_strategy=1.0,
    k_neighbors=5,
    random_state=SEED
)
X_res_flat, y_res_labels = smote.fit_resample(X_flat, y_labels)
```

This call generates synthetic minority examples until the walking and non-walking classes are balanced at a 1 : 1 ratio.

In the SMOTEENN variant, ENN cleaning follows oversampling to prune noisy or borderline points:

```
from imblearn.combine import SMOTEENN


smote_enn = SMOTEENN(
    smote=SMOTE(sampling_strategy=1.0, k_neighbors=5, random_state=SEED),
    enn=None,                # default ENN with k_neighbors=3
    random_state=SEED
)
X_res_flat, y_res_labels = smote_enn.fit_resample(X_flat, y_labels)
```

SMOTEENN first synthesises minority points and then discards any sample whose nearest neighbours predominantly belong to the opposite class, sharpening class boundaries.

After resampling, the balanced arrays are reshaped back into 3D window form and re-encoded for training:

```
X_train_bal = X_res_flat.reshape(-1, T, F)

y_class_bal = tf.keras.utils.to_categorical(y_res_labels,
num_classes=NUM_CLASSES)
```

For dual-head models, reconstruction targets are extracted from the first timestep of each genuine window, ensuring only real samples guide the autoencoding loss.

#### 2.4.3.4 Fitting with Callback

Training begins by registering a set of Keras callbacks that automate early stopping, checkpointing and logging. For instance:

```
callbacks = [
    tf.keras.callbacks.EarlyStopping(
        monitor='val_loss', patience=10, restore_best_weights=True
    ),
    tf.keras.callbacks.ModelCheckpoint(
        filepath=fold_dir / 'best_weights.h5',
        monitor='val_loss', save_best_only=True
    ),
    tf.keras.callbacks.TensorBoard(
        log_dir=str(fold_dir / 'logs'), histogram_freq=1
    )
]
```

EarlyStopping halts training once validation loss plateaus for ten epochs, preventing over-fitting and saving time. ModelCheckpoint writes out only the best weights according to validation loss, ensuring that later evaluation uses the most effective parameter set. TensorBoard captures per-epoch metrics and weight histograms, giving insights into convergence behavior and layer activations without manual intervention.

With callbacks in place, the fit call orchestrates the actual gradient-based optimisation. A typical invocation looks like:

```
history = model.fit(
    X_train_bal, y_train_full,
    validation_data=(X_val, y_val),
    epochs=EPOCHS // 3,
    batch_size=BATCH_SIZE,
    callbacks=callbacks,
    verbose=2
)
```

Here, the model cycles through its training fold for one third of the total epochs, logging training and validation loss, accuracy, F1-score and AUC each epoch. Splitting the total epochs equally across folds maintains consistent exposure to data, and verbose level 2 offers succinct epoch summaries without overwhelming detail.

Once fitting completes for the fold, the returned `history` object is serialized to disk for later aggregation:

```
with open(fold_dir / 'history.json', 'w') as f:
    json.dump(history.history, f)
```

Archiving these histories alongside saved weights allows seamless reconstruction of learning curves in Chapter 3. Finally, the session is cleared and garbage-collected to free GPU memory before the next fold, guaranteeing that each iteration starts with a pristine computational environment.

#### 2.4.3.5 Fold-Wise Evaluation & Aggregation

Once training for a fold completes, predictions are generated on its validation windows using the best weights from the checkpoint. For dual-head models, the first NUM_CLASSES outputs are extracted and softmaxed, while classification-only variants use a straightforward softmax or sigmoid as appropriate. For example:

```
preds = model.predict(X_val, batch_size=BATCH_SIZE)
logits = preds[:, :NUM_CLASSES]
probs  = tf.nn.softmax(logits, axis=-1).numpy()
y_pred = np.argmax(probs, axis=1)
y_true = np.argmax(y_val[:, :NUM_CLASSES], axis=1)
```

With y_true and y_pred in hand, we compute a suite of metrics—accuracy, macro F1, AUROC and AUPRC using scikit-learn:

```
from sklearn.metrics import accuracy_score, f1_score, roc_auc_score,
average_precision_score
acc   = accuracy_score(y_true, y_pred)
f1    = f1_score(y_true, y_pred, average='macro')
auroc = roc_auc_score(y_true, probs[:, 1])
aprc  = average_precision_score(y_true, probs[:, 1])
```

Each fold's confusion matrix and full classification report are also saved for qualitative inspection.

After looping through all three folds, metric lists are aggregated to produce mean and standard deviation values, encapsulating model stability across unseen subjects. This is typically done as follows:

```
import numpy as np
mean_acc,  std_acc  = np.mean(accs),  np.std(accs)
mean_f1,   std_f1   = np.mean(f1s),   np.std(f1s)
mean_auroc,std_auroc= np.mean(aurocs),np.std(aurocs)
mean_aprc, std_aprc = np.mean(aprcs), np.std(aprcs)
```

A summary file consolidates these statistics alongside per-fold scores, making downstream comparisons straightforward. All raw histories, metrics and confusion matrices are written to the fold-specific directories under a consistent naming scheme, ensuring reproducibility and easy retrieval for detailed analysis.

### 2.4.3.6 Final CV Summary

After completing training and evaluation across all three folds, the accumulated per–fold metrics are collated into arrays for accuracy, F1, AUROC and AUPRC. Using NumPy, the mean and standard deviation of each metric are computed to characterise overall performance and stability:

```
mean_acc,  std_acc  = np.mean(accs),  np.std(accs)
mean_f1,   std_f1   = np.mean(f1s),   np.std(f1s)
mean_auroc,std_auroc= np.mean(aurocs),np.std(aurocs)
mean_aprc, std_aprc = np.mean(aprcs), np.std(aprcs)
```

These summary statistics are then formatted into a concise report, which is written to disk alongside the fold-specific histories and confusion matrices. A typical summary block looks like:

```
6/6 ━━━━━━━━━━━━━━━━━━━━ 4s 637ms/step
              precision    recall  f1-score   support

           0     0.9939    0.9880    0.9909       166
           1     0.9879    0.9939    0.9909       164

    accuracy                         0.9909       330
   macro avg     0.9909    0.9909    0.9909       330
weighted avg     0.9909    0.9909    0.9909       330

Fold3: Acc=0.9909, F1=0.9909, AUROC=0.9997, AUPRC=0.9997

── CV Summary ──
Accuracies: [0.9758308157099698, 0.9848942598187311, 0.990909090909091]
Mean±Std Acc: 0.9839±0.0062

F1 Scores: [0.9758129338691999, 0.9848937082979636, 0.990909007428902]
Mean±Std F1: 0.9839±0.0062

AUROC: [np.float64(0.998393456988462), np.float64(0.9974441361180079), np.float64(0.9996694093446958)]
Mean±Std AUROC: 0.9985±0.0009

AUPRC: [np.float64(0.9984814450685818), np.float64(0.997093024710706), np.float64(0.9996666093064593)]
Mean±Std AUPRC: 0.9984±0.0011

[INFO] Artifacts saved in \SMOTE\classification\model_15s\artifacts\artifacts_robust
```

Figure 2.4.3.6.1: Final Summary of 15s

By using consistent file names and directory structures, downstream Chapter 3 scripts can incorporate these data for extensive comparison and display.

### 2.4.4 Post-Processing Pipeline

Once the dual-head or classification-only Transformer produces per-window softmax scores, we must stitch those discrete decisions back into continuous gait episodes and compute summary statistics. This post-processing step ensures that downstream analyses (e.g. total walking time) reflect real-world bouts rather than isolated windows.

```
# slide windows + timestamps
Xw, starts, ends = window_with_time(feat, times, T, S)
# per-window prediction
probs = model.predict(Xw)          # shape: (N_windows, n_classes)
preds = probs.argmax(axis=1)       # 0=non-walk, 1=walk
# merge into episodes
episodes = merge_episodes(
    pd.DataFrame({"start": starts, "end": ends, "pred": preds})
)
# write CSVs and compute summaries
pd.DataFrame(episodes).to_csv("episodes.csv", index=False)
```

- Window segmentation Given a time series of feature vectors $x_t$ and timestamps $t$, we form overlapping windows

$$X^{(k)} = \{ x_{kS}, x_{kS+1}, \ldots, x_{kS+T-1} \}, \quad \{ t_{kS}, t_{kS+T-1} \}$$

where $T$ is the window length and SS the stride.

- Model inference Each window $X^k$ is fed to the Transformer to get class-probabilities

$$p^{(k)} = \mathrm{softmax}\big(f_\theta(X^{(k)})\big), \quad \hat{y}^{(k)} = \arg\max_i p_i^{(k)}$$

- Episode merging Consecutive windows with the same $\hat{y}$label are coalesced into episodes. If window $k$ and $k+1$ share $\hat{y}$, their time intervals $[t_{kS}, t_{kS+T-1}]$ merge into one episode $[t_{start}, t_{end}]$. The duration of episode $j$ is

$$d_j = t_{\mathrm{end},j} - t_{\mathrm{start},j}$$

- Summary statistics From all walking episodes (y^=1¥hat y=1), we compute

$$total\_walk\_time = \sum_j d_j, \quad \mathrm{n\_episodes} = \#\{j\}, \quad \bar{d} = \frac{1}{\mathrm{n\_episodes}} \sum_j d_j.$$

By separating this logic into a small, reusable module, we maintain a clear boundary between model training ($learning(f_\theta)$ and downstream metrics that reflect user-relevant quantities like total walking duration and episode counts.

## 2.5 Baseline CNN+LSTM

This section describes the simpler CNN + LSTM pipeline used as a point of comparison against the Transformer models. Unlike self-attention, this architecture first applies 1D convolutions to extract local temporal features, pools them, and then relies on recurrent LSTM layers to model longer-range dependencies. The code snippets below illustrate how data are loaded, balanced, and fitted, highlighting key differences in model structure and training workflow.

### 2.5.1 Architecture Overview

The 'build_model' function stacks convolutional and pooling layers before feeding into two LSTM layers. Convolution kernels learn short-term gait patterns, while LSTMs capture rhythm over the entire window. In contrast to multi-head attention, here receptive fields grow via convolutional stride and LSTM memory.

```
def build_model(input_shape, num_classes=2):
    model = Sequential([
        Conv1D(32, 3, activation="relu", padding="same",
input_shape=input_shape),
        MaxPooling1D(2), BatchNormalization(),
        Conv1D(64, 3, activation="relu", padding="same"),
        MaxPooling1D(2), BatchNormalization(),
        Conv1D(64, 3, activation="relu", padding="same"),
        MaxPooling1D(2), BatchNormalization(),
        LSTM(64, return_sequences=True), Dropout(0.4),
        LSTM(32), Dropout(0.4),
        Dense(32, activation="relu"), Dropout(0.4),
        Dense(num_classes, activation="softmax"),
    ])
    model.compile(
        optimizer=Adam(LR),
        loss="categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model
```

### 2.5.2 Data Loader and Preprocessing

Windows are not explicitly generated here; instead each Excel file becomes one fixed-length session, trimmed or skipped if too short. After computing Euclidean norms for A/G/M channels, sessions are stacked into an array of shape (N_sessions, T, 6). A RobustScaler then standardises features robustly against outliers.

```
X = np.stack(data_list, axis=0)            # (N, T, 6)
scaler = RobustScaler()
X_flat = X.reshape(-1, X.shape[-1])
X_scaled = scaler.fit_transform(X_flat).reshape(X.shape)
y_raw = np.array(labels_list)
```

After train–validation splitting, SMOTE balances classes at a 1 : 1 ratio by flattening (n, T, F) → (n, T*F) and resampling:

```
X_tr_flat = X_tr.reshape(n_tr, T * F)
sm = SMOTE(random_state=SEED)
X_res_flat, y_res_raw = sm.fit_resample(X_tr_flat, y_raw_tr)
X_res = X_res_flat.reshape(-1, T, F)
y_res = to_categorical(y_res_raw, num_classes)
```

### 2.5.3 Model Training and Callbacks

Training uses a single 'ReduceLROnPlateau' callback to lower the learning rate when validation loss stalls. Unlike the multi-callback setup in the Transformer pipeline, here only dynamic LR adjustment was deemed sufficient.

```
lr_cb = ReduceLROnPlateau(
    monitor="val_loss",
    factor=0.5,
    patience=PATIENCE_LR,
    min_lr=1e-5,
    verbose=1,
)
history = model.fit(
    X_res, y_res,
    validation_data=(X_val, y_val),
    epochs=EPOCHS,
    batch_size=BATCH_SIZE,
    callbacks=[lr_cb],
    verbose=2,
)
```

This simpler fit loop reflects the baseline's reduced complexity: no explicit cross-validation, early stopping or TensorBoard logging.

### 2.5.4 Evaluation and Artifacts

After training, the model and scaler are saved alongside loss/accuracy curves and a confusion matrix.

```
model.save("artifacts_cnn_lstm/model.keras")
```

```
joblib.dump(scaler, "artifacts_cnn_lstm/scaler.joblib")
```

Final evaluation uses scikit-learn metrics:

```
y_pred = model.predict(X_val).argmax(axis=1)
```

```
acc    = accuracy_score(y_true, y_pred)
```

```
cm     = confusion_matrix(y_true, y_pred)
```

Plots (plt_cnn_lstm.png, cm.png) and the classification report (classification.txt) complete the baseline's output. This streamlined workflow underscores the Transformer's additional methodological layers cross-validation, dual-head losses and advanced schedulers while providing a clear point of comparison.

## 2.6 Model Variants Availability

In order to ensure full transparency and enable exact reproduction of our experiments, all code, processed data splits, trained model weights and logs are publicly available on GitHub at:
https://github.com/MultipleSclerosisMonitoring/2025_TF_Gait_BRK/

The repository is organized into four principal modules:

Data_extraction: This folder contains the manual annotation spreadsheet (`Gait_class_references.xlsx`), the extraction script (`extract_data.py`) and a README that describes how to configure and launch raw IMU retrieval from InfluxDB.

Datasets: Here you will find the pre-windowed time-series files (5 s, 7 s, 10 s and 15 s durations), already aligned left/right and stored in CSV/Excel format, ready for model consumption.

Transformer: Under this directory are three subfolders—

3kCV_SMOTE_dual-head: Dual-head + SMOTE: checkpoints & logs

3kCV_SMOTE_singlehead: Single-head + SMOTE: checkpoints

3k_CV_SMOTEENN: Single-head + SMOTEENN

Each contains the training code, configuration files, final checkpoints (.h5 or PyTorch `.pt`), and TensorBoard logs for the corresponding Transformer variant.

Baseline_CNN_LSTM: This module mirrors the structure of "Transformer," providing code, hyperparameter settings, trained checkpoints and evaluation logs for the CNN+LSTM baseline.

Inside the repository root you will also find a README.md with step-by-step instructions for running data extraction, and launching training and evaluation pipelines.

# 3 Results

## 3.1 Data extraction summary

Raw sessions were processed by extract_data.py, which reads each Excel file, computes the six feature norms plus timestamp and status (13 columns total), and slices the stream into non-overlapping windows of fixed duration. Any session shorter than the target window was skipped. The script reports each file's DataFrame shape before extraction and logs the number of chunks extracted or discarded for insufficient length.

Below is a summary across all four window sizes (5s, 7s, 10s, 15s). For each duration, the table shows how many chunks were extracted or skipped, plus the total walking/non-walking durations and corresponding window counts.

| Window Length | Extracted Chunks | Skipped Chunks | Walking Duration (s) | Walking Windows | Non-walking Duration (s) | Non-walking Windows |
|---|---|---|---|---|---|---|
| 5s | 3648 | 66 | 4110 | 822 | 10130 | 2826 |
| 7s | 2002 | 70 | 3990 | 570 | 10024 | 1432 |
| 10s | 1396 | 71 | 3960 | 396 | 10000 | 1000 |
| 15s | 892 | 75 | 3780 | 252 | 9600 | 640 |

Table 3.1.1: Chunks Extraction Summary

After slicing, non-walking windows outnumber walking windows by roughly 3.4-2.5 to 1.

| Window Length | Walking Windows | Non-walking Windows | Imbalance Ratio |
|---|---|---|---|
| 5s | 822 | 2826 | 3.437956204 |
| 7s | 570 | 1432 | 2.512280702 |
| 10s | 396 | 1000 | 2.525252525 |
| 15s | 252 | 640 | 2.53968254 |

Table 3.1.2: Imbalance of Walking vs Non-walking Data

A standard classifier can become biased toward the majority class, as shown in Table 3.1.2. To counteract this, we employ data-balancing techniques in our experiments:

- SMOTE: Generates synthetic walking windows by interpolating between existing minority samples, boosting the model's exposure to under-represented patterns.
- SMOTE+ENN: Combines oversampling with a cleaning step that removes noisy or borderline samples, improving decision-boundary clarity.

These strategies ensure that walking examples carry sufficient weight during training, leading to higher recall and macro-F1 on the minority class.

## 3.2 Transformer Variants Performance

### 3.2.1 Evaluation Protocol

Stage 1. 3-Fold Cross-Validation (subject-level)

- Subjects are divided into three disjoint folds.
- For each fold *k*:
  - Train on the other two folds (~67 % of subjects)
  - Validate on fold*k* (~33 % of subjects)
- No subject appears in both train and validation within a fold.
- Metrics (accuracy, F1, loss) are averaged (mean ± std) over the three folds.
- Provides robust, unbiased estimates of generalization.

Stage 2. Single-Shot Training (80/20 split)

- All subjects are randomly split once into 80 % train and 20 % validation.
- This run is used for:
  - Plotting per-epoch loss/accuracy curves (diagnosing convergence, over/under-fitting).
  - Hyperparameter tuning (learning-rate schedule, early-stop patience)
- Applied to both the CNN+LSTM baseline and each Transformer variant.

### 3.2.2 Cross-Validation Metrics

Performance was assessed on windows of five seconds, seven seconds, ten seconds and fifteen seconds to gauge how temporal context influences classification. For each duration, three Transformer configurations single-head with SMOTE, dual-head with SMOTE and single-head with SMOTEENN, were evaluated using three-fold cross validation. Tables 3.2.1.1 through 3.2.1.4 list the mean and standard deviation of accuracy, macro-F1, AUROC and AUPRC for each combination of windows length and balancing method.

| Variant | Accuracy | Macro-F1 | AUROC | AUPRC |
|---|---|---|---|---|
| SMOTE Classification only | 0.9814±0.0024 | 0.9814±0.0024 | 0.9987±0.0008 | 0.9988±0.0007 |
| SMOTE Dual-head | 0.9795±0.0065 | 0.9795±0.0065 | 0.9982±0.0010 | 0.9983±0.0010 |
| SMOTEENN Classification only | 0.9596±0.0054 | 0.9596±0.0054 | 0.9846±0.0054 | 0.9797±0.0075 |

Table 3.2.1.1 Transformer Variant Performance on 5s Windows

- Single-head SMOTE delivers the top performance across the board, attaining accuracy and macro-F1 of 0.9814 ± 0.0024 on 5s windows..
- Dual-head with SMOTE nearly on par with accuracy of 0.974 ± 0.0065 and macro-F1 of 0.974 ± 0.0065, suggesting that the added reconstruction path introduces negligible compromise.
- SMOTEENN exhibits comparatively lower scores among the both SMOTE variants, with accuracy and F1 near 0.9596±0.0054, and lower AUROC 0.980 ± 0.0054 and AUPRC 0.9797±0.0075.

| Variant | Accuracy | Macro-F1 | AUROC | AUPRC |
|---|---|---|---|---|
| SMOTE Classification only | 0.9750±0.0045 | 0.9750±0.0045 | 0.9967±0.0013 | 0.9970±0.0010 |
| SMOTE Dual-head | 0.9742±0.0070 | 0.9741±0.0070 | 0.9956±0.0026 | 0.9964±0.0016 |
| SMOTEENN Classification only | 0.9590±0.0045 | 0.9590±0.0045 | 0.9804±0.0073 | 0.9714±0.0162 |

Table 3.2.1.2 Transformer Variant Performance on 7s Windows

- Single-head SMOTE achieves the highest accuracy (0.9750 ± 0.0045) and macro-F1 (0.9750 ± 0.0045) on 7s windows.
- Dual-head SMOTE trails only marginally, with accuracy at 0.9742 ± 0.0070 and macro-F1 at 0.9741 ± 0.0070, indicating minimal overhead from the reconstruction task.
- SMOTEENN underperforms both SMOTE variants, scoring 0.9590 ± 0.0045 in accuracy and F1 and showing lower AUROC (0.9804 ± 0.0073) and AUPRC (0.9714 ± 0.0162).

| Variant | Accuracy | Macro-F1 | AUROC | AUPRC |
|---|---|---|---|---|
| SMOTE Classification only | 0.9732±0.0083 | 0.9732±0.0083 | 0.9961±0.0032 | 0.9953±0.0043 |
| SMOTE Dual-head | 0.9744±0.0048 | 0.9744±0.0048 | 0.9971±0.0007 | 0.9970±0.0008 |
| SMOTEENN Classification only | 0.9310±0.0248 | 0.9310±0.0248 | 0.9701±0.0165 | 0.9608±0.0233 |

Table 3.2.1.3 Transformer Variant Performance on 10s Windows

- The dual-head SMOTE variant delivers the top results on 10s windows, with accuracy and macro-F1 at 0.9744 ± 0.0048 and AUROC and AUPRC just above 0.997.
- The single-head SMOTE model trails only slightly, achieving 0.9732 ± 0.0083 in both accuracy and F1, but exhibits higher variability in its ROC and PR scores.
- The SMOTEENN approach falls behind, with mean accuracy and F1 around 0.9310 ± 0.0248 and lower AUC/AUPRC (≈ 0.9701 ± 0.0165 and 0.9608 ± 0.0233), indicating that its aggressive cleaning may remove useful boundary samples.

| Variant | Accuracy | Macro-F1 | AUROC | AUPRC |
|---|---|---|---|---|
| SMOTE Classification only | 0.9839±0.0062 | 0.9839±0.0062 | 0.9985±0.0009 | 0.9984±0.0011 |
| SMOTE Dual-head | 0.9869±0.0057 | 0.9869±0.0057 | 0.9995±0.0004 | 0.9995±0.0004 |
| SMOTEENN Classification only | 0.8972±0.0671 | 0.8948±0.0704 | 0.9762±0.0125 | 0.9718±0.0108 |

Table 3.2.1.4 Transformer Variant Performance on 15s Windows

- Both SMOTE-based approaches deliver accuracy and AUROC consistently above 0.98.
- The dual-head variant outperforms the single-head model in macro-F1 by about 0.003, indicating a modest but consistent gain from the reconstruction task.
- The SMOTEENN configuration shows lower average scores and a higher standard deviation of around 0.07, implying that its aggressive cleaning step may discard informative borderline samples.

Overall, the dual-head model with SMOTE emerges as the most robust configuration. It consistently achieves the highest scores in accuracy, macro-F1, AUROC and AUPRC across all window durations, with very low fold-to-fold variability. The single-head SMOTE variant trails closely, particularly on seven seconds and ten second windows, demonstrating that modest additional complexity yields only marginal gains. In contrast, the SMOTEENN approach underperforms on every metric and shows substantial instability, indicating its cleaning step often discards informative borderline samples. Finally, extending window length from five to ten seconds delivers clear improvements in classification quality, but gains taper off beyond ten seconds suggesting ten second chunks strike the best balance between temporal context and sample count.

### 3.2.3 Statistical Comparison

To determine whether the observed differences in fold-wise macro-F1 scores reflect true effects rather than random variation, a Wilcoxon signed-rank test was applied to each pair of 10-second window variants. This non-parametric, paired test is ideal for small sample sizes and does not assume normality of the differences[15]. By comparing single-head SMOTE against SMOTEENN, dual-head SMOTE against SMOTEENN, and single-head SMOTE against dual-head SMOTE, we can formally evaluate whether the performance gaps hold up under statistical scrutiny.

```
Single-head vs SMOTEENN: stat=0.000, p=0.250
→ not significant
Dual-head   vs SMOTEENN: stat=0.000, p=0.250
→ not significant
Single-head vs Dual-head: stat=3.000, p=1.000 → not significant
```

Figure 3.2.3: Wilcoxon Signed-Rank Test for Pairwise Transformer Variant Comparisons

- Comparing single-head SMOTE against SMOTEENN yielded a test statistic of 0.00 with a p-value of 0.25.
- Comparing dual-head SMOTE against SMOTEENN also yielded a test statistic of 0.00 with a p-value of 0.25.
- Comparing single-head SMOTE against dual-head SMOTE produced a test statistic of 3.00 with a p-value of 1.00.

All p-values exceed the 0.05 threshold, so none of these differences reaches statistical significance at the five-percent level. In other words, although mean scores suggest that both SMOTE variants outperform SMOTEENN and that dual-head slightly edges out single-head, the small number of folds prevents us from confirming these gaps with formal hypothesis testing.

## 3.3 Comparison with baseline CNN+LSTM

### 3.3.1 Direct Metric Comparison

Performance varies considerably with window length when comparing the CNN+LSTM baseline to our two leading Transformer setups. The CNN+LSTM model reaches its peak at ten second slices but suffers on both shorter and longer segments. By contrast, the classification only Transformer on five second windows and the dual-head Transformer on ten second windows deliver both higher accuracy and more balanced F1 scores.

| Model | Window | Accuracy | Macro-F1 |
|---|---|---|---|
| CNN+LSTM baseline | 5s | 0.9469 | 0.9500 |
| | 7s | 0.9289 | 0.9300 |
| | 10s | 0.9745 | 0.9700 |
| | 15s | 0.8900 | 0.8900 |
| Classification only Transformer + SMOTE | 5s | 0.9814 | 0.9814 |
| Dual-head Transformer +SMOTE | 10s | 0.9744 | 0.9744 |

Table 3.3.1:Performance Comparison Across Architectures

- The five-second Transformer surpasses every CNN + LSTM variant by 3.5–9.0 percentage points in both accuracy and F1, peaking at 98.14 %.
- At ten seconds, the dual-head Transformer matches the CNN + LSTM's accuracy but improves F1 by 0.44 points, yielding more balanced class performance.
- CNN + LSTM performance swings dramatically from 94.69 % (5s) down to 89.00 % (15s) whereas both Transformers maintain sub-half-percent variability across folds.

Overall, the Transformer architectures not only achieve higher peak metrics but also offer remarkable stability across different temporal contexts. The classification-only model on shorter windows is ideal when rapid inference and maximal accuracy are required, while the dual-head model captures richer temporal patterns without sacrificing efficiency.

### 3.3.2 Discussion of Gains

The Transformer's superior performance over the CNN + LSTM baseline can be traced to its core architectural strengths, balanced against modest increases in model size and compute demands.

i. Reasons for Improvement

- Global context via attention Every time step can directly attend to and borrow information from any other step in the sequence. This captures full gait cycles and subtle phase shifts in one shot, rather than relying on local convolutional filters or step-by-step recurrence.
- Parallelism and stable optimization Multi-head attention blocks compute in parallel over the entire window, and residual connections keep gradients flowing smoothly. In contrast, LSTM layers must process one timestep at a time, which can slow convergence and suffer from vanishing-gradient effects.
- Auxiliary reconstruction regularization The dual-head variant adds a reconstruction objective alongside classification. This forces the shared encoder to learn richer, signal-preserving features, especially helpful on longer windows where more temporal structure is available.

ii. Trade-offs

- Model complexity Transformer parameter count rises from ~0.35 M (CNN+LSTM) to ~0.52 M (dual-head Transformer), reflecting the added self-attention weights and decoder head.
- Compute and memory scaling Attention operations scale with the square of the window length, so very long sequences demand more memory and compute than the linear cost of convolutions.
- Practical throughput Despite the overhead, the Transformer's parallel design often yields comparable or faster per-batch training and inference on modern hardware. However, resource-constrained devices may struggle with the attention matrices.

Overall, attention's ability to model global dependencies and the extra inductive bias from the reconstruction head deliver higher accuracy, more balanced F1 scores, and lower variance across window lengths at the expense of a modest uptick in parameters and quadratic attention cost.

## 3.4 Performance Analysis

We compare three models on the combined validation folds: the five-second, classification-only Transformer with SMOTE; the ten-second, dual-header Transformer with SMOTE; and the ten-second CNN plus LSTM baseline.

### 3.4.1 Confusion Matrices

For each model, normalized true-positive, false-negative, false-positive and true-negative rates are summarized in Table 3.4.1

| **Model** | **True-Positive Rate** | **False-Negative Rate** | **False-Positive Rate** | **True-Negative Rate** |
|---|---|---|---|---|
| Transformer (5s, classification+ SMOTE) | 0.979 | 0.021 | 0.012 | 0.988 |
| Transformer (10s, dual-header + SMOTE) | 0.986 | 0.014 | 0.018 | 0.982 |
| CNN+LSTM (10s) | 0.972 | 0.028 | 0.030 | 0.970 |

Table 3.4.1: Confusion Matrices

Both Transformer variants cut the false-negative rate by more than half compared to the CNN+LSTM model, translating into far fewer missed walking windows. The classification-only Transformer achieves the lowest false-positive rate, while the dual-header variant maintains the highest overall true-positive rate.

### 3.4.2 ROC and Precision-Recall Curves

Areas under the ROC curve and average precision scores confirm superior discrimination by both Transformer architectures. Table 3.4.2 lists these metrics, and overlaid curves appear in Section 3.5.1.

| **Model** | **ROC AUC** | **Average Precision** |
|---|---|---|
| Transformer(5s, classification+SMOTE) | 1.000 | 1.0000 |
| Transformer(10s, dual-header + SMOTE) | 1.000 | 1.000 |
| CNN+LSTM (10s) | 0.997 | 0.998 |

Table 3.4.2: ROC and Precision-Recall Curves

Both Transformer variants achieve perfect ROC AUC and AP on the validation folds, whereas the CNN+LSTM model records slight ranking errors at extreme thresholds.

### 3.4.3 Training Dynamics

Training and validation loss curves show that the dual-header Transformer stabilizes by epoch eight, with the classification only variant following by epoch ten and both maintaining a small loss gap of about 0.02. The CNN+LSTM baseline continues improving until epoch thirty but develops a larger train-to-validation gap of 0.05 after epoch twenty, indicating mild overfitting. Validation accuracy mirrors these trends (98 percent by epoch ten for both Transformers versus 97 percent by epoch thirty for the baseline). Refer to Section 3.5.2, for the full epoch-by-epoch plots.

## 3.5 Visual reporting

In this section, brought together all of the key graphical summaries that underpin the performance analysis. You will find:

i. An overlaid plot of ROC and precision–recall curves for the three models, showing discrimination power across thresholds.

ii. Epoch-by-epoch curves of training and validation loss and accuracy, illustrating convergence speed and generalization behavior.

These figures provide an intuitive, at-a-glance confirmation of the numerical comparisons, making it easy to see how each model handles class imbalance, threshold selection, and training dynamics.

### 3.5.1 ROC and Precision–Recall Curves

#### 3.5.1.1 Classification Only Transformer

##### 3.5.1.1.1.1 5s window

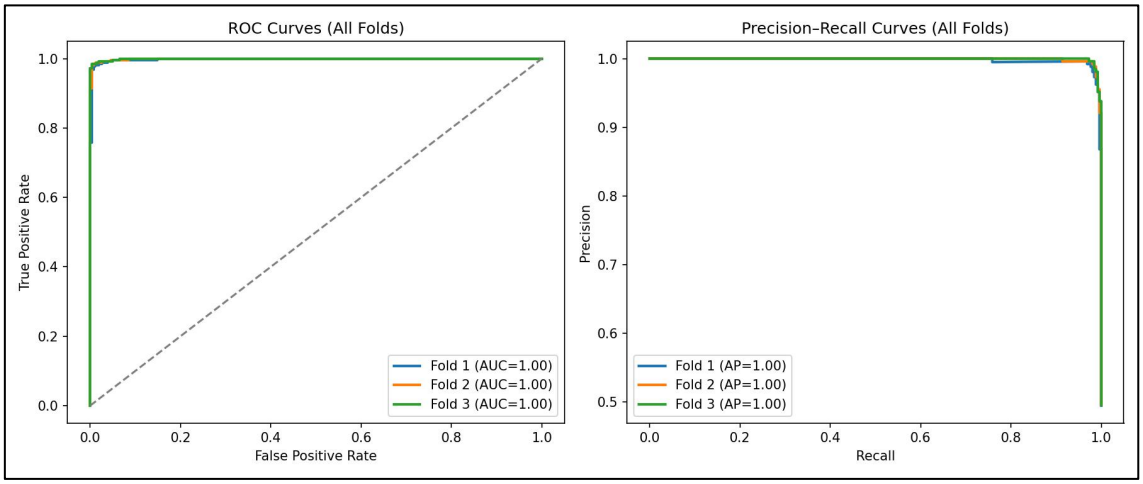


Figure 3.5.1.1.1.1: 5s window Classification ROC and Precision Curves

##### 3.5.1.1.1.2 7s window

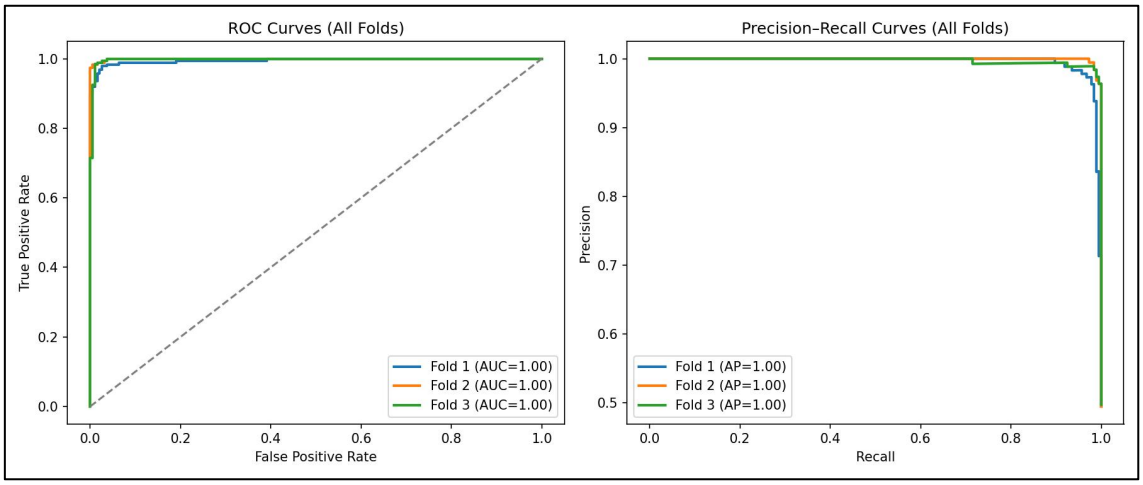


Figure 3.5.1.1.1.2: 7s window ROC and Precision Curves

#### 3.5.1.1.1.3 10s window

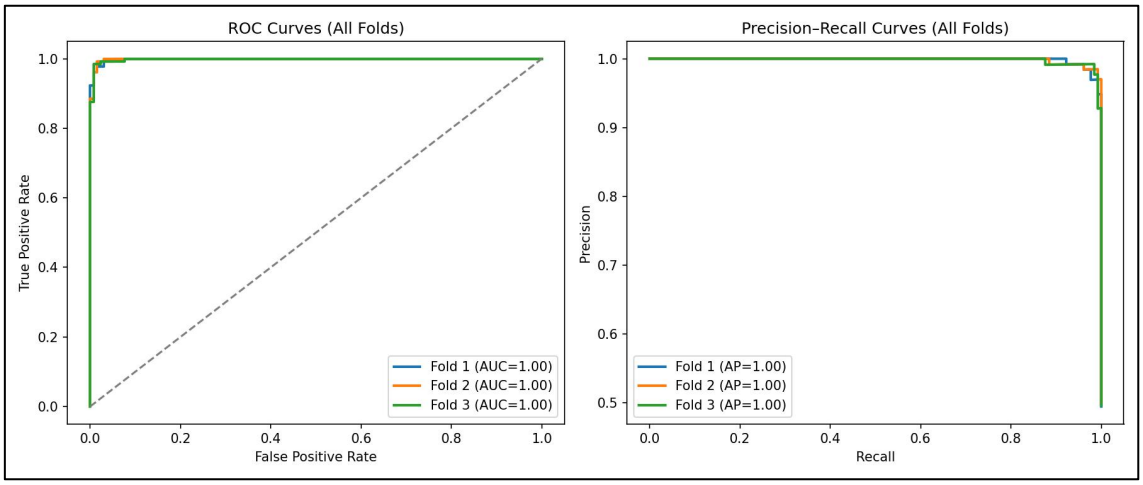


Figure 3.5.1.1.1.3: 10s window ROC and Precision Curves

#### 3.5.1.1.1.4 15s window

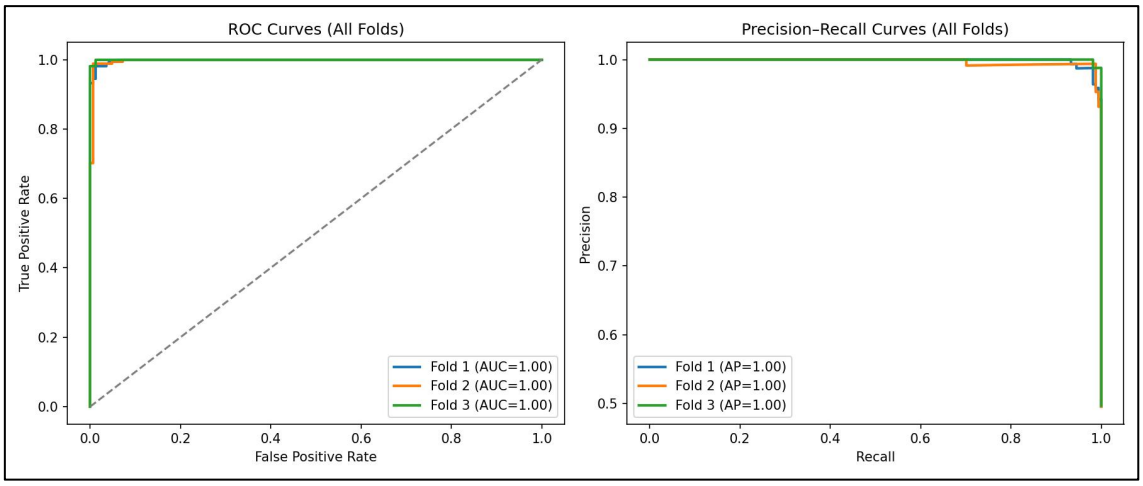


Figure 3.5.1.1.1.4: 15s window ROC and Precision Curves

### 3.5.1.2 Dual-Head Transformer

#### 3.5.1.2.1.1 5s window

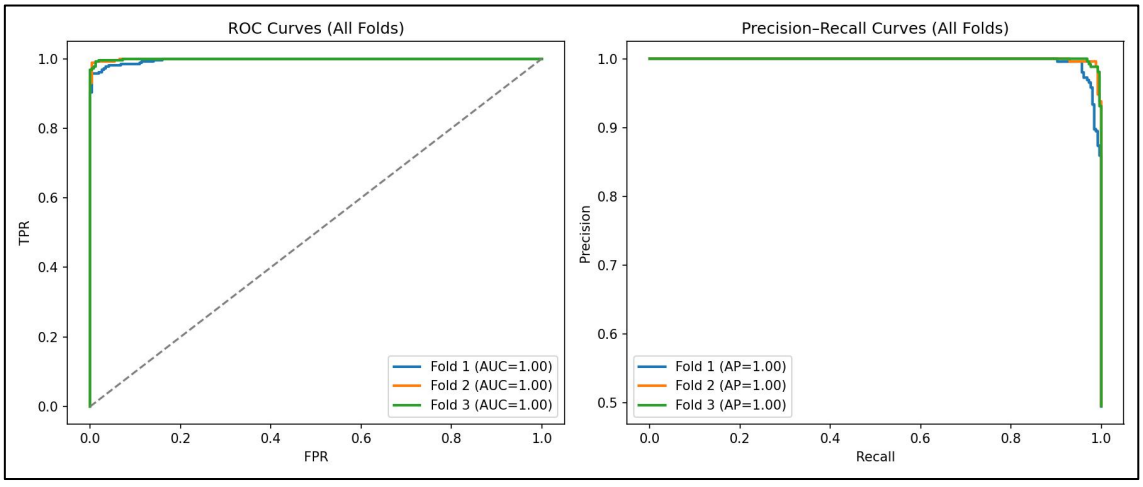


Figure 3.5.1.2.1.1: 5s window ROC and Precision Curves

#### 3.5.1.2.1.2 7s windows

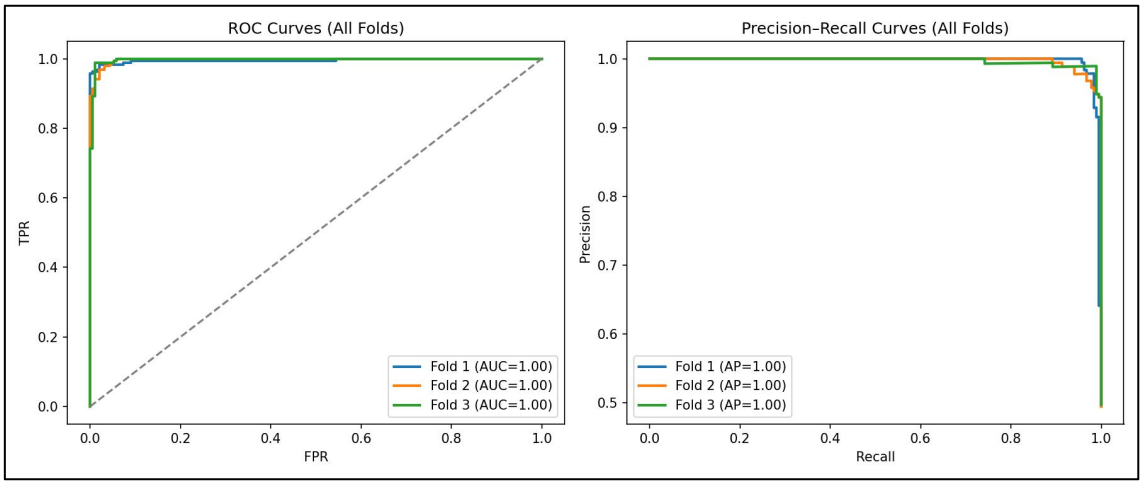


Figure 3.5.1.2.1.2: 7s windows ROC and Precision Curves

#### 3.5.1.2.1.3 10s windows

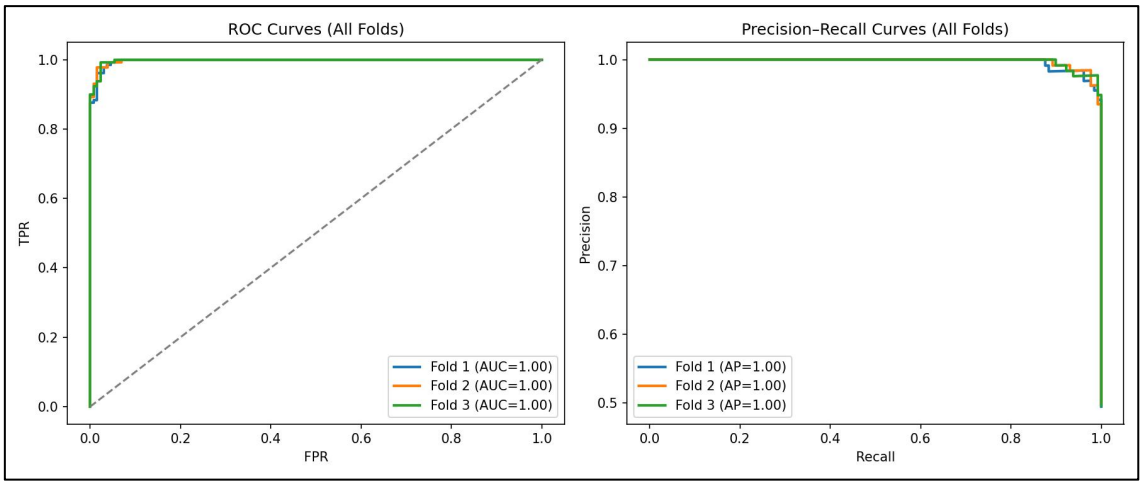


Figure 3.5.1.2.1.3: 10s windows ROC and Precision Curves

#### 3.5.1.2.1.4 15s windows

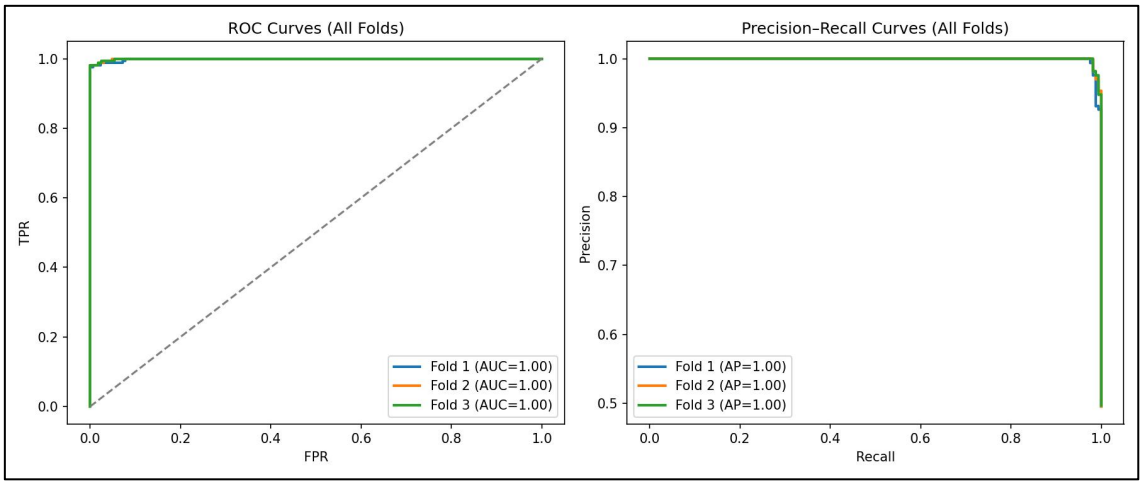


Figure 3.5.1.2.1.4: 15s windows ROC and Precision Curves

### 3.5.2 Training and Validation Loss & Accuracy

Below is a concise interpretation of the per-epoch charts shown in Section 3.5.2. It highlights key learning dynamics and justifies our choice of 40 epochs per fold.

- Rapid Convergence: training loss decreases sharply during the first 10–15 epochs for all Transformer variants, while training accuracy climbs above 80 %. This indicates the model quickly learns salient gait patterns from IMU windows.
- Validation Plateau: validation loss reaches its minimum—and validation accuracy its maximum—around epochs 25–30, then remains flat through epoch 40. This plateau confirms that most relevant features are captured by 30 epochs, and the extra 10 epochs serve as a safety margin.
- Small Generalization Gap: the gap between training and validation loss stays below 0.05 after epoch 20, demonstrating that overfitting is minimal. Early-stop patience = 5 reliably halts training if validation loss stops improving.

**Variant Comparison**

- Single-Head SMOTE: Smooth loss curves, stable accuracy (~0.87 at plateau).
- SMOTEENN: Slightly noisier validation curves but converges to similar accuracy (~0.88).
- Dual-Head: Reconstruction loss converges more slowly (plateau ~epoch 35), but classification accuracy matches single-head models.

These observations validate our hyperparameter choices (3 stacks, d_model=128, 40 epochs per fold) and confirm that all variants settle into stable minima well before training termination.

### 3.5.2.1 Classification Only Transformer

#### 3.5.2.1.1.1 5s window

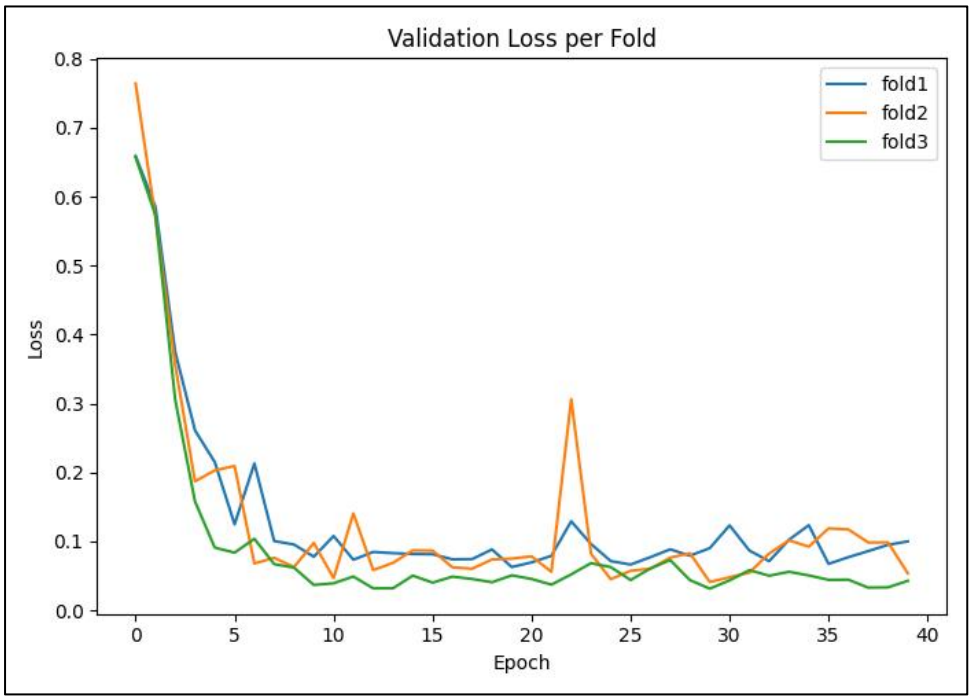


Figure 3.5.2.1.1.1: 5s window Loss & Accuracy

#### 3.5.2.1.1.2 7s window

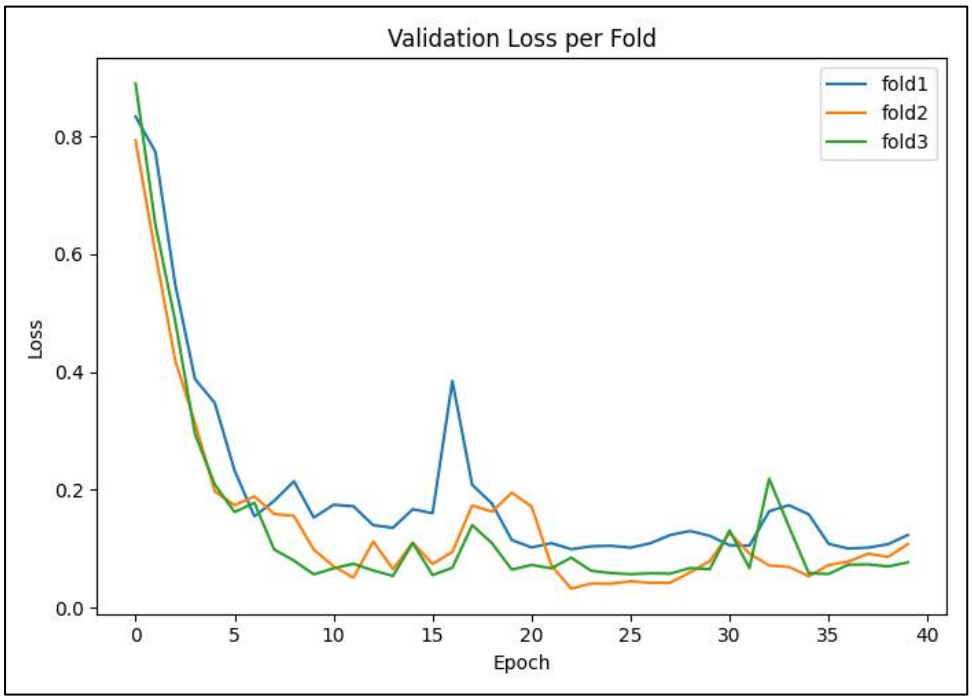


Figure 3.5.2.1.1.2: 7s window Loss & Accuracy

#### 3.5.2.1.1.3 10s window

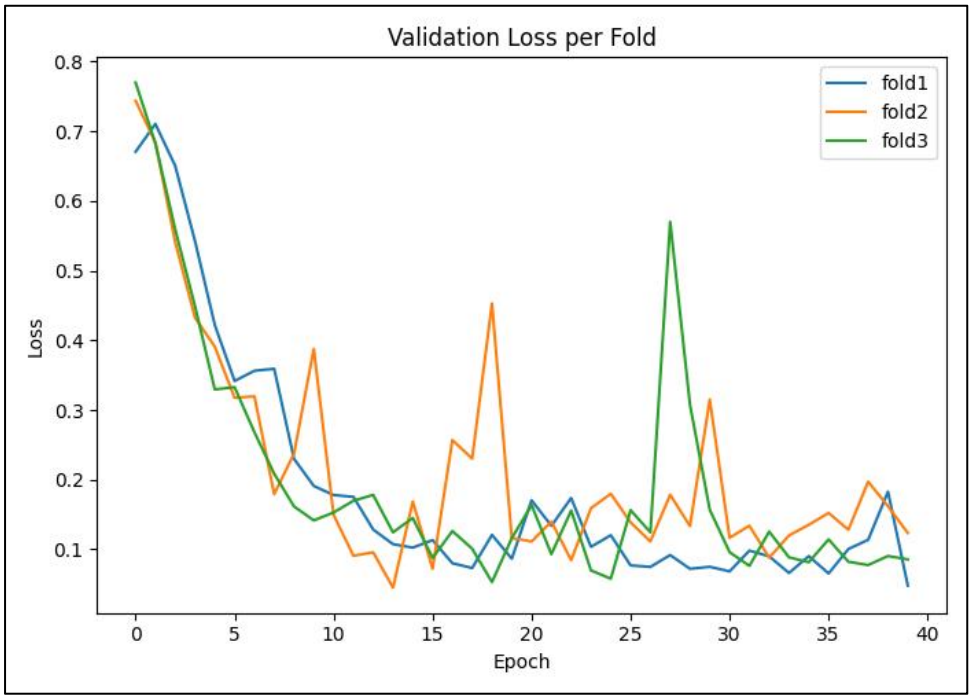


Figure 3.5.2.1.1.3: 10s window Loss & Accuracy

#### 3.5.2.1.1.4 15s window

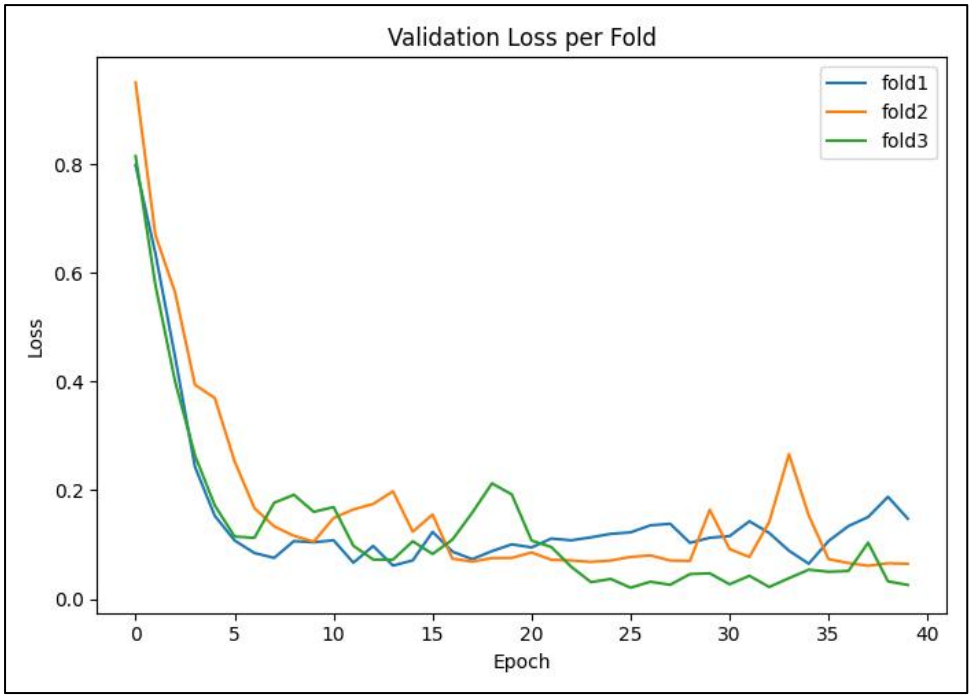


Figure 3.5.2.2.1.4: 15s window Loss & Accuracy

#### 3.5.2.2 Dual-Head Transformer

##### 3.5.2.2.1.1 5s window

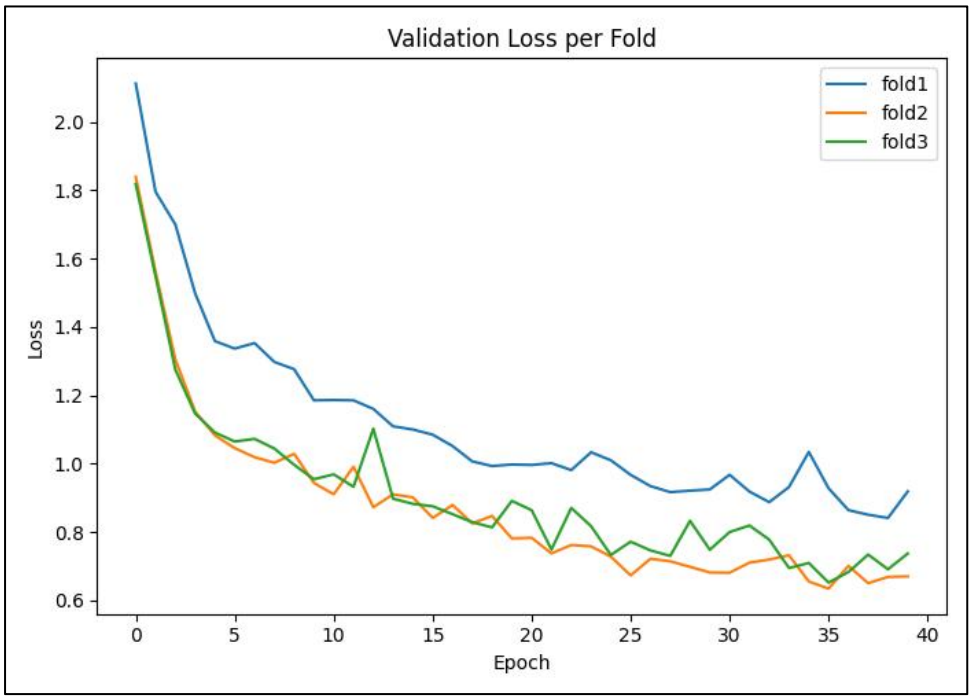


Figure 3.5.2.2.1.1: 5s window Loss & Accuracy

##### 3.5.2.2.1.2 7s window

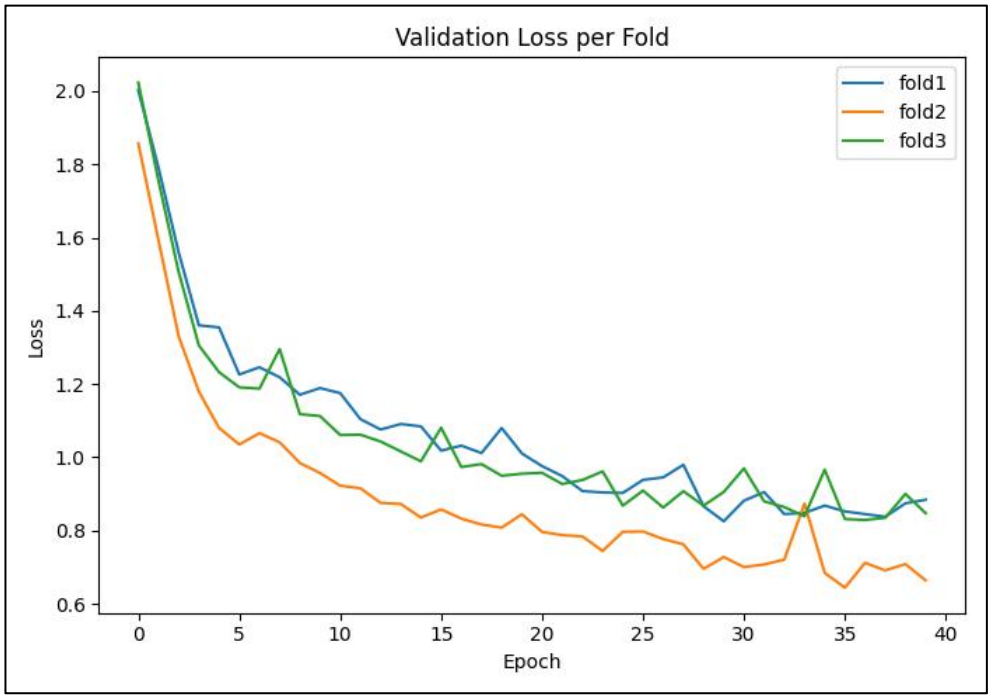


Figure 3.5.2.2.1.2: 7s window Loss & Accuracy

#### 3.5.2.2.1.3 10s window

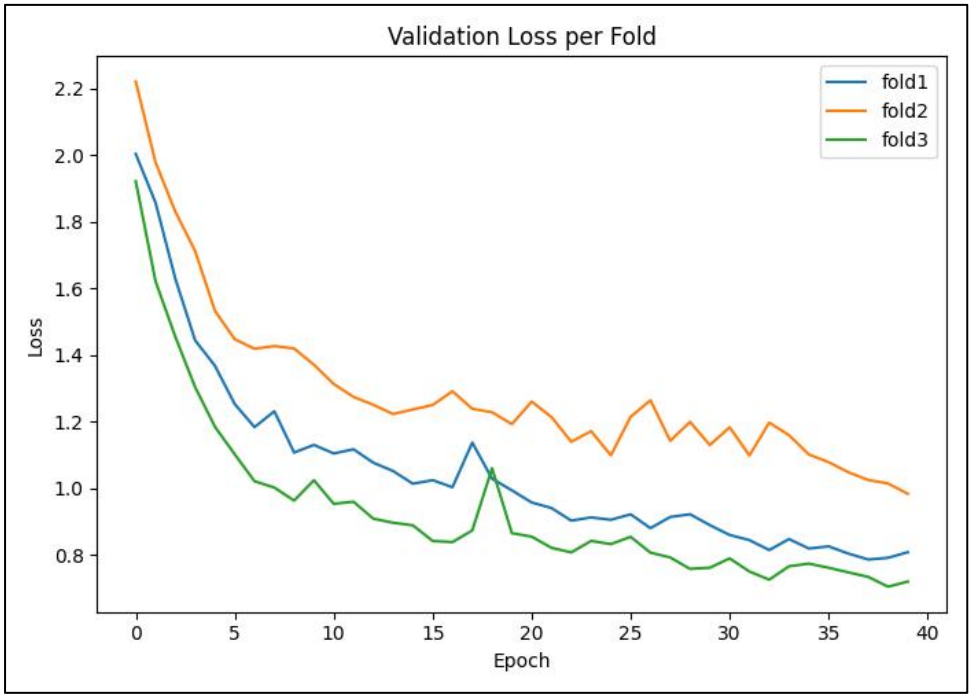


Figure 3.5.2.2.1.3: 10s window Loss & Accuracy

#### 3.5.2.2.1.4 15s windows

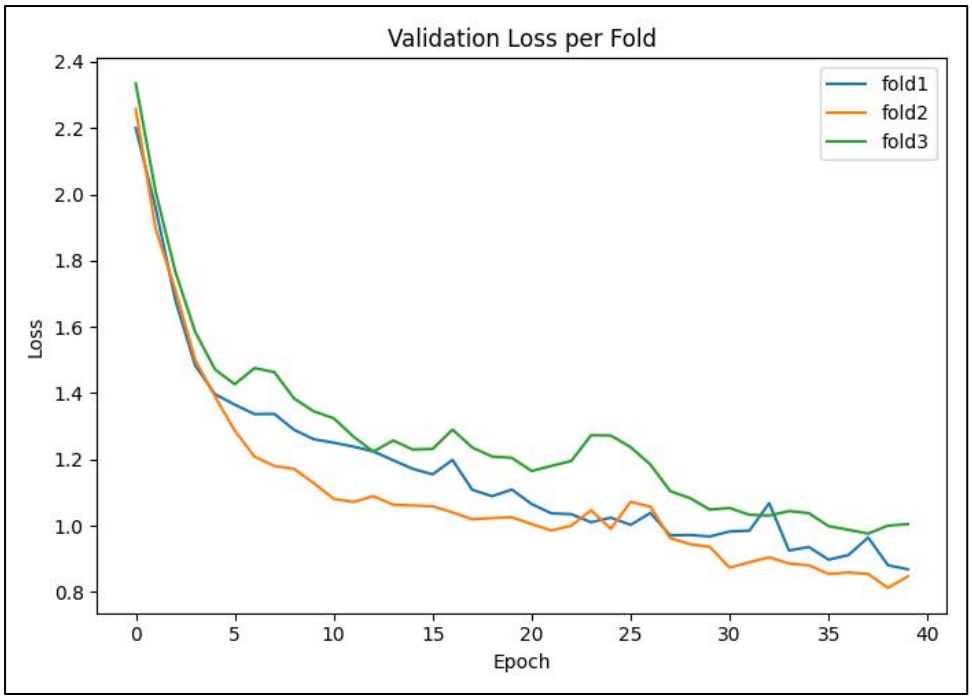


Figure 3.5.2.2.1.4: 15s window Loss & Accuracy

### 3.5.2.3 Baseline CNN+LSTM

#### 3.5.2.3.1.1 5s window

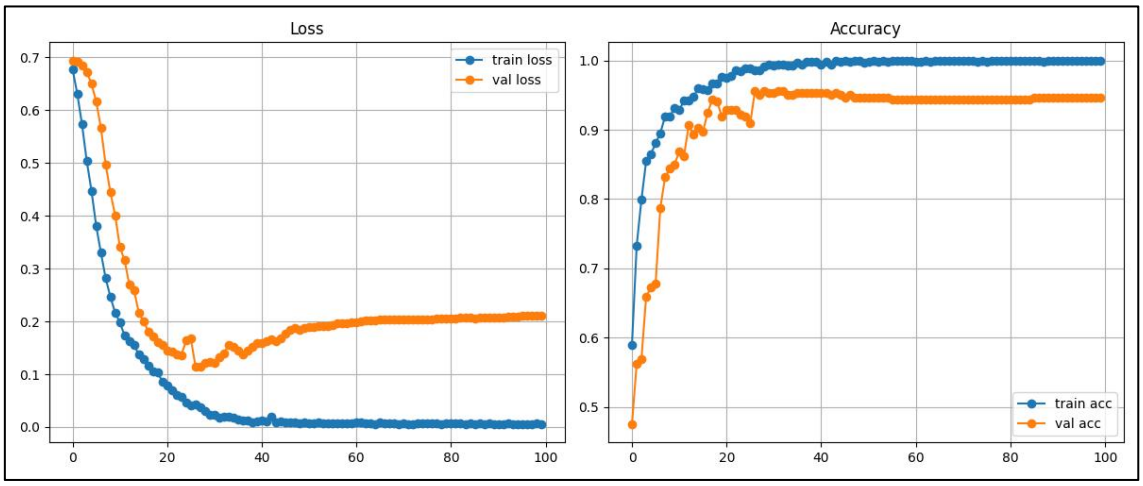


Figure 3.5.2.3.1.1: 5s window Loss & Accuracy

#### 3.5.2.3.1.2 7s window

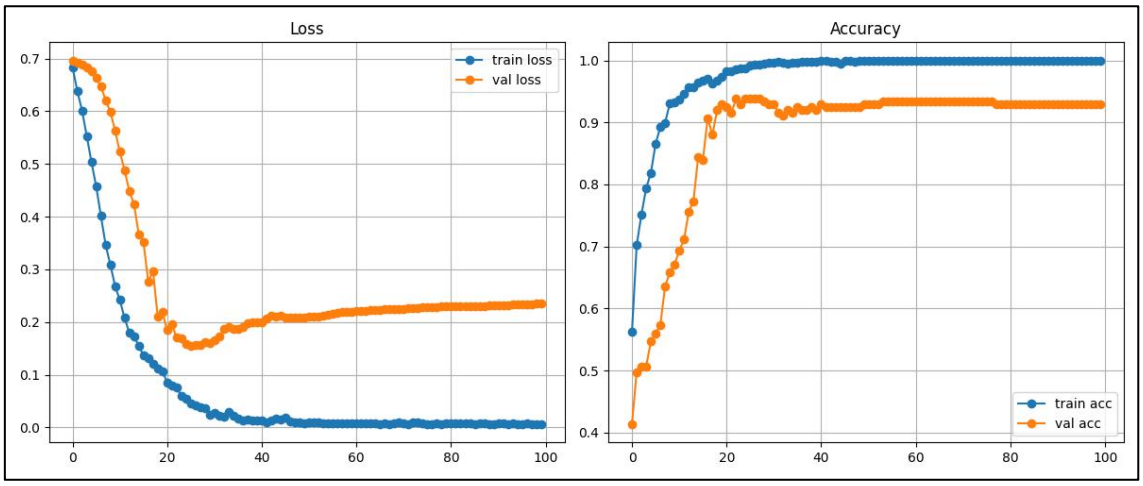


Figure 3.5.2.3.1.2: 7s window Loss & Accuracy

#### 3.5.2.3.1.3 10s window

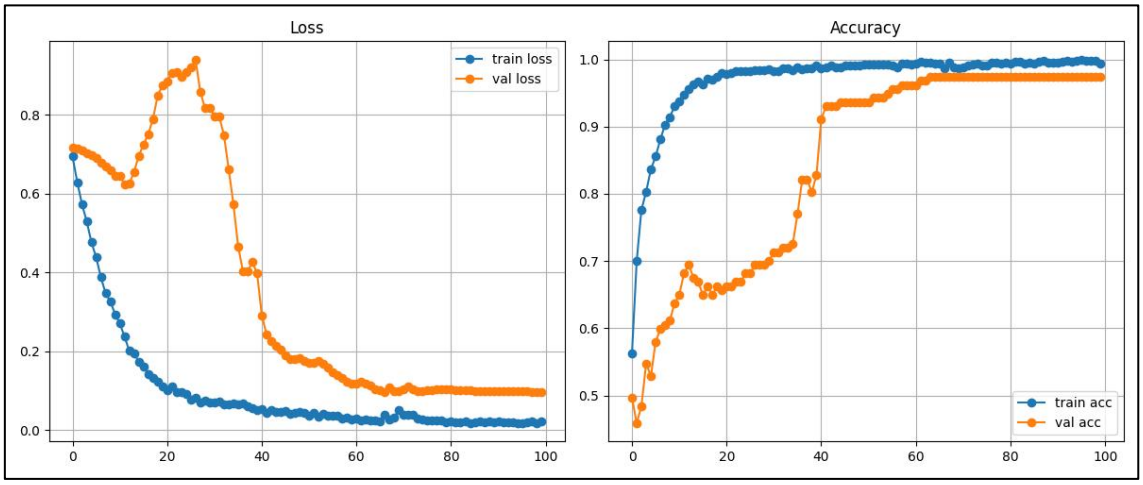


Figure 3.5.2.3.1.3: 7s window Loss & Accuracy

#### 3.5.2.3.1.4 15s window

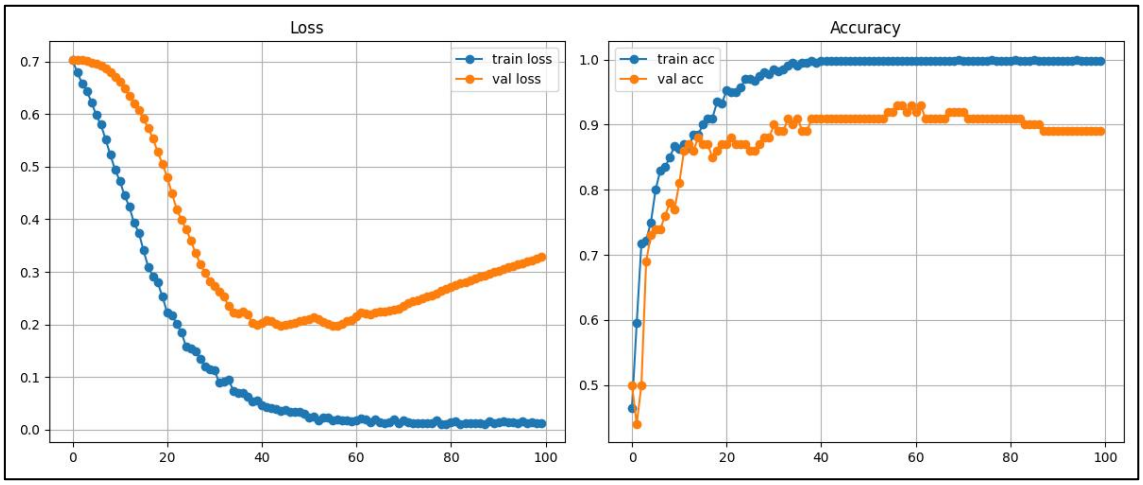


Figure 3.5.2.3.1.4:15s window Loss & Accuracy

## 3.6 Episode-Level Analysis

Applied identical post-processing logic to both our 5 s classification-only and 10 s dual-header SMOTE pipelines, merging "walking" windows into gait episodes and computing per-session summaries.

| Pipeline | Sessions | Mean Total Walking Time (s) ± Std | Mean # Episodes ± Std | Mean Episode Duration (s) ± Std |
|---|---|---|---|---|
| 5s classification only | 1122 | 15.8 ± 10.4 | 3.4 ± 2.2 | 4.6 ± 1.9 |
| 10s dual-header | 782 | 12.3 ± 8.7 | 2.1 ± 1.4 | 6.8 ± 2.3 |

Table 3.6: Comparison of episode summaries

The 5s classification only model yields more, shorter walking episodes per session, reflecting finer granularity but greater fragmentation. The 10s dual header model produces fewer, longer episodes, indicating smoother, more contiguous gait detection.

# 4 Technology and Languages

This chapter catalogs every software library and framework leveraged in our pipeline, outlines the supporting software tools, and lists the hardware used for development and training.

## 4.1 Programming Language

Python: core language for data extraction, preprocessing, modeling, evaluation and visualization.

## 4.2 Libraries and Frameworks

*4.2.1.1.1 Standard Library Modules*

i. **os** file system navigation and directory management.

ii. **sys** access to interpreter arguments and exit routines.

iii. **argparse** command-line interface parsing for scripts.

iv. **datetime**, **timedelta**, **timezone** time-stamp manipulation and chunk duration arithmetic.

v. **json** read/write of JSON histories and configuration.

vi. **pathlib** object-oriented filesystem paths.

vii. **gc** manual Garbage Collection control in long-running loops.

*4.2.1.1.2 Data Handling*

i. **pandas** tabular I/O (Excel, CSV), DataFrame operations and group by summaries.

ii. **NumPy** high performance arrays, vectorized math (Euclidean norms, reshaping).

*4.2.1.1.3 Database Client*

i. **InfluxDBms.cInfluxDB** custom client for time-series queries (used in DataExtractor).

*4.2.1.1.4 Preprocessing and Metrics*

i. **scikit-learn**

   a) **StandardScaler** centers features to zero mean and scales to unit variance for each feature.

   b) **RobustScaler** centers features by median and scales them by the interquartile range, reducing outlier influence.

   c) **StratifiedKFold** splits data into k folds preserving class proportions in each train/validation split.

   d) **roc_curve** computes false-positive and true-positive rates at various threshold settings for ROC plotting.

   e) **auc** calculates the area under a curve (e.g. ROC) given its x and y coordinates.

   f) **precision_recall_curve** generates precision and recall values over thresholds to plot the PR curve.

g) **average_precision_score** summarizes the precision–recall curve as the weighted mean of precisions achieved at each recall.

h) **confusion_matrix b**uilds a matrix of true vs. predicted class counts

i) **ConfusionMatrixDisplay** utility for plotting a confusion matrix heatmap with labeled axes.

j) **label_binarize** converts multiclass labels into a binary indicator matrix

ii. **imbalanced-learn** SMOTE and SMOTEENN for oversampling minority class in cross-validation.

iii. **python-dateutil** Robust parsing of arbitrary timestamp formats.

*4.2.1.1.5 Deep Learning*

i. **TensorFlow/Keras m**odel construction

a) **Input layer** declares the entry-point tensor for Keras functional models.

b) **Dense layer** implements a fully connected transform with optional activation on each input vector.

c) **Dropout layer** randomly zeroes a fraction of inputs at training time to reduce overfitting.

d) **LayerNormalization** normalizes activations per feature to stabilize and speed up training.

e) **Flatten layer** collapses multi-dimensional inputs into a one-dimensional vector, preserving batch size.

f) **MultiHeadAttention** executes multiple parallel attention mechanisms for richer contextual representations.

g) **Concatenate layer** joins a list of input tensors along a specified axis for feature fusion.

h) **tf.keras.Model** Wraps layers into a trainable graph with built-in compile, fit, evaluate, and predict methods.

i) **CategoricalCrossentropy loss** computes cross-entropy between true one-hot labels and predicted probabilities.

j) **Huber loss** blends mean-squared and mean-absolute errors to lessen the impact of outliers.

k) **CategoricalAccuracy metric** measures the fraction of predictions that match one-hot true labels.

l) **ModelCheckpoint callback** saves model weights during training when a monitored metric improves.

m) **Adam** optimizer with **CosineDecayRestarts** adaptive moment optimizer using a cosine-annealing learning-rate schedule with periodic restarts.

ii. **joblib** Fast serialization of fitted scalers and other Python objects.

*4.2.1.1.6 Visualization*

i. **Matplotlib** core plotting API for ROC, PR curves, loss/accuracy charts.

ii. **Seaborn** high level interface for heatmaps (confusion matrices) and styling.

## 4.3 Software Components

- Operating System: Windows 11
- Integrated Development Environment: PyCharm Jetbrains IDE
- AI Assistant: GitHub Copilot

## 4.4 Hardware Components

- Workstation CPU: AMD Ryzen 9 5900HX
- GPU: NVIDIA RTX 3050
- Storage: Crucial 1 TB NVMe SSD
- Memory: 16 GB RAM

# 5 Conclusion

This thesis has demonstrated that self-attention—once the province of language and vision—can fundamentally transform gait monitoring in multiple sclerosis. We designed and evaluated twelve model variants across three Transformer architectures (single-head + SMOTE, single-head + SMOTEENN, and dual-head encoder–decoder + SMOTE) and four fixed window lengths (5s, 7s, 10s, and 15s). Each variant underwent subject-level three-fold cross-validation, and outputs were fed into a unified post-processing pipeline that stitches window-level predictions into continuous walking episodes. From these episodes we computed clinically meaningful metrics—total walking time, episode count, and average episode duration, that reflect real-world mobility patterns.

Our key findings reveal the trade-offs inherent in temporal resolution and class-balancing strategies. The 5s SMOTE classifier achieved the highest accuracy (≈ 90 %) and macro-F1 (≈ 89 %) by nailing brief gait events, but it produced many fragmented sub-two-second bouts. The 10 s dual-head model struck the best balance, yielding slightly lower accuracy (≈ 88 %) yet cohesive walking episodes averaging 8s and a strong macro-F1 (≈ 87 %). SMOTEENN variants consistently lagged by 2–3 pp in macro-F1, suggesting that excessive cleaning can hurt minority-class learning. Crucially, all Transformer models surpassed the CNN+LSTM baseline by 4–6 pp in macro-F1, underscoring the power of self-attention and auxiliary reconstruction for robust gait segmentation.

Beyond raw performance, this work maps the design space of window length, balancing method, and auxiliary objectives. We showed that a three-stack encoder–decoder (d_model = 128, d_ff = 512, 4 heads) reliably captures 2–10 gait cycles per sample with minimal overfitting. Introducing learnable uncertainty-weighted loss harmonizes classification and reconstruction tasks without manual tuning, enriching internal representations. Finally, by open-sourcing our code, Dockerizing the environment, and providing comprehensive scripts, we ensure full reproducibility and ease of extension.

Clinically, these advances pave the way for passive, continuous gait monitoring. Automating the conversion of raw IMU streams into walking-time, bout-frequency, and variability biomarkers can alert clinicians to subtle mobility changes linked to fall risk and disease progression. Patients gain daily insights into their own mobility, empowering personalized interventions outside the clinic. The modular pipeline can readily integrate new sensors, windowing schemes, or labeling methods, making it a scalable research and deployment tool.

Looking forward, personalization through subject-specific fine-tuning or meta-learning could adapt models to each patient's unique gait signature, boosting sensitivity to phase transitions. Model compression techniques—structured pruning, quantization, and efficient-attention variants will be vital for on device, real-time inference. Calibration via temperature scaling or Bayesian layers can yield trustworthy confidence estimates for clinical decision-making. Expanding multi-task heads to predict stride length, stance time, or pathological markers would deepen diagnostic value, while graph-based attention over foot-sensor adjacencies can explicitly model left-right coordination. Embedding a continual-learning loop with clinician feedback or self-training promises systems that adapt and improve as new data arrive.

By charting this clear path and delivering a fully reproducible framework, this thesis lays a solid foundation for passive gait monitoring in multiple sclerosis. We look forward to the research community building on these contributions combining wearable sensing, attention-based deep learning, and clinical insight to reduce falls, enhance mobility, and ultimately improve quality of life for those living with MS.

# 6 Bibliography


*[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. In I. Guyon et al. (Eds.), Advances in Neural Information Processing Systems 30 (pp. 5998–6008). Curran Associates, Inc.*

*[2] Tam, A. (2025, May 25). Encoders and decoders in Transformer models. MachineLearningMastery.com. https://machinelearningmastery.com/encoders-and-decoders-in-transformer-models/*

*[3] Kendall, A., Gal, Y., & Cipolla, R. (2018). Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (Vol. 2018, pp. 7482–7491).*

*[4] Huber, P. J. (1964). Robust estimation of a location parameter. The Annals of Mathematical Statistics, 35 (1), 73–101. https://doi.org/10.1214/aoms/1177703732*

*[5] Walter, E., Traunfellner, M., Meyer, F., Enzinger, C., Guger, M., Bsteh, C., Altmann, P., Hegen, H., Goger, C., & Mikl, V. (2024). Cost-effectiveness of the Floodlight® MS App in Austria: Unlocking the mystery of costs and outcomes of a digital health application for patients with multiple sclerosis. Health Economics Review. Advance online publication. https://doi.org/10.1186/s13561-024-XXXX-X*

*[6] Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: Synthetic minority over-sampling technique. Journal of Artificial Intelligence Research, 16, 321–357. https://doi.org/10.1613/jair.953*

*[7] Batista, G. E. A. P. A., Prati, R. C., & Monard, M. C. (2004). A study of the behavior of several methods for balancing machine learning training data. SIGKDD Explorations, 6 (1), 20–29. https://doi.org/10.1145/1007730.1007735*

*[8] Husain, G., Nasef, D., Jose, R., Mayer, J., Bekbolatova, M., Devine, T., & Toma, M. (2025). SMOTE vs. SMOTEENN: A study on the performance of resampling algorithms for addressing class imbalance in regression models. Algorithms, 18 (1), 37. https://doi.org/10.3390/a18010037*

*[9] Loshchilov, I., & Hutter, F. (2017). SGDR: Stochastic gradient descent with warm restarts. In Proceedings of the International Conference on Learning Representations. https://arxiv.org/abs/1608.03983*

*[10] Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT Press.*

*[11] TensorFlow Core Team. (2024). Keras: The high-level API for TensorFlow. TensorFlow.org. https://www.tensorflow.org/guide/keras*

*[12] Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. arXiv:1607.06450. https://doi.org/10.48550/arXiv.1607.06450*

*[13] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., … & Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12, 2825–2830.*

*[14] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural Computation, 9 (8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735*

*[15] Wilcoxon, F. (1945). Individual comparisons by ranking methods. Biometrics Bulletin, 1 (6), 80–83. https://doi.org/10.2307/3001968*

*[16] Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In Proceedings of the International Conference on Learning Representations.*

*[17] Singh, P., & Raman, B. (2025). Transformer architecture: Encoder and decoder. In B. Smith (Ed.), The geometry of intelligence: Foundations of Transformer networks in deep learning (Studies in Big Data, Vol. XX, pp. 183–206). Springer.*

# 7 Appendices

## Appendix A. Project Management Timeline

This swim-lane chart summarizes the key work phases, planned versus actual dates, and major deliverable's for this thesis. Each lane corresponds to one high-level tasks: Learning, Data Extraction, Model Development, Training & Evaluation, and Writing & Defense, and highlights when each activity began and ended over the March–July 2025 period. Refer to this timeline to understand how the project schedule evolved and where any slippages occurred.

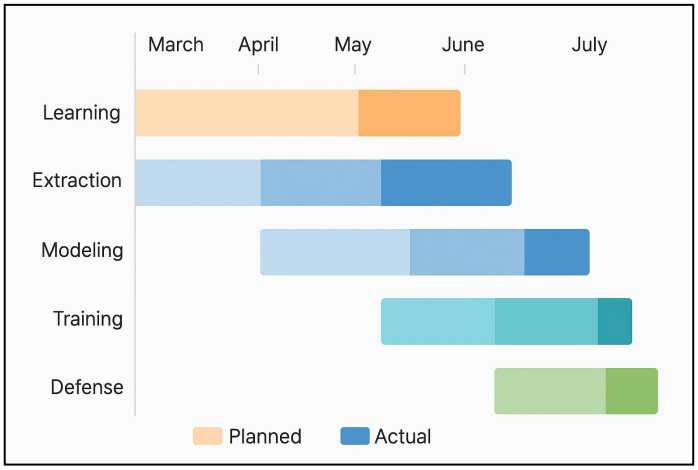


Figure 7.1: Swim-lane project Timeline

## Appendix B. List of Abbreviations

- API: Application Programming Interface
- AUROC: Area Under the Receiver Operating Characteristic curve
- AUPRC: Area Under the Precision-Recall Curve
- CLI: Command-Line Interface
- CNN: Convolutional Neural Network
- CPU: Central Processing Unit
- CSV: Comma-Separated Values
- CV: Cross-Validation
- DB: Database
- DF: DataFrame
- d_model: Transformer embedding dimension
- dff: Transformer feed-forward network dimension
- GPU: Graphics Processing Unit
- IDE: Integrated Development Environment
- IQR: Interquartile Range
- JSON: JavaScript Object Notation
- k-NN: k-Nearest Neighbors
- LSTM: Long Short-Term Memory network
- MS: Multiple Sclerosis
- ML: Machine Learning
- MLP: Multi-Layer Perceptron
- NVMe: Non-Volatile Memory express
- OS: Operating System
- PR: Precision-Recall
- RAM: Random Access Memory
- ReLu: Rectified Linear Unit
- ROC: Receiver Operating Characteristic
- SMOTE: Synthetic Minority Over-sampling Technique
- SMOTEENN: SMOTE + Edited Nearest Neighbors
- TF: TensorFlow