Project Report

Anomaly Detection in ECG Signals

using LSTM and GRU Autoencoders

Students:

- **HAMDI Aya G03**
- **AMIEUR Zineb Ichrake G03**
- **MEFLAH Yousra G02**

Supervisor:

- **Dr. Nassima DIF**

Année universitaire 2024-2025

Contents

1 Introduction

Electrocardiography (ECG) is a vital technique for monitoring the electrical activity of the heart and diagnosing cardiovascular abnormalities such as arrhythmias, myocardial infarctions, and other irregularities. Automated ECG anomaly detection plays a crucial role in providing scalable, efficient, and accurate diagnostic assistance.

Autoencoders, a class of unsupervised neural networks, are particularly effective for anomaly detection in time series data. When trained solely on normal ECG signals, autoencoders learn to reconstruct these signals with high fidelity. When an anomalous ECG is passed through the autoencoder, it typically results in a higher reconstruction error due to the deviation from the normal training distribution. This property makes autoencoders a powerful tool for detecting anomalies.

In this study, we employ recurrent neural network (RNN) based autoencoders, specifically Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) architectures, to detect anomalies in ECG signals by modeling their sequential nature.

2 Dataset Overview

2.1 Dataset Description

The ECG dataset used in this work is publicly available and can be accessed from the following link:

`http://storage.googleapis.com/download.tensorflow.org/data/ecg.csv`

The dataset consists of time-series ECG samples, where each row represents an individual ECG recording. Each record contains 140 numerical values representing the time-series signal amplitudes, followed by a label column:

- **1** – Represents a **normal** ECG signal.
- **0** – Represents an **anomalous** ECG signal.

2.2 Columns

The dataset includes:

- `feature_0`, `feature_1`, ..., `feature_139` – these represent ECG signal values over time.
- `label` – binary label indicating whether the signal is normal (1) or abnormal (0).

2.3 Label Distribution

The dataset contains both normal and anomalous signals. For training purposes, only the normal signals (label = 1) were used to allow the autoencoder to learn the distribution of healthy ECG patterns. Abnormal signals (label = 0) are used exclusively for testing and evaluation.

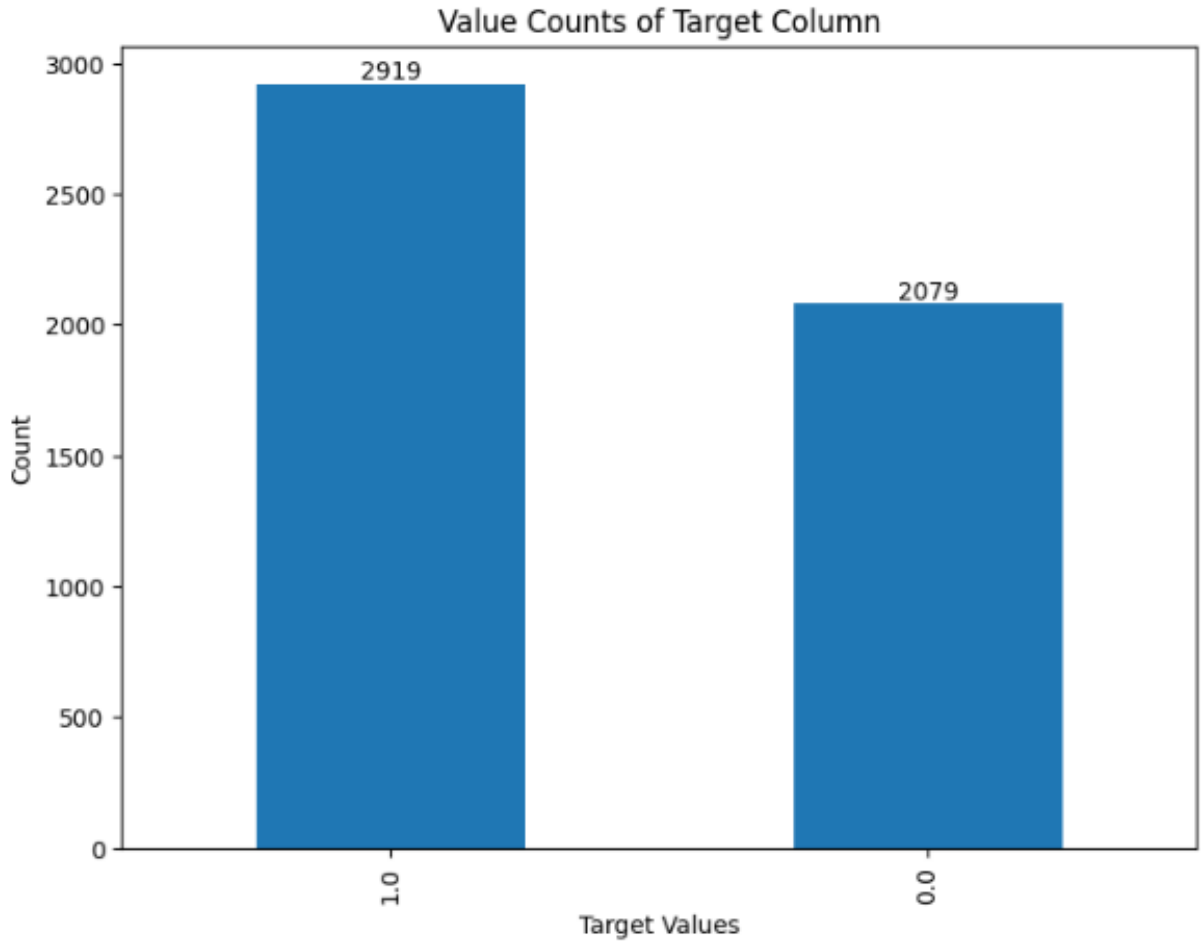

Figure 1: The Apriori Algorithm

2.4 Data Splitting Strategy

The normal data was split into training, validation, and test sets as follows:

1. **Initial Train-Validation Split**: The dataset containing only normal signals was split into 85% training and 15% validation sets using `train_test_split` with a fixed random seed for reproducibility.

2. **Further Splitting**: The validation set obtained from the previous step was further split into two parts—70% for final validation and 30% for testing—again using the same random seed for consistency.

Listing 1: Dataset Splitting Code

```
# Initial Train-Validation Split:
train_df, val_df = train_test_split(
  normal_df,
  test_size=0.15,
  random_state=42
)
```

```
# Further Split for Validation and Test:
val_df, test_df = train_test_split(
  val_df,
  test_size=0.30,
  random_state=42
)
```

3 Model Architecture

3.1 LSTM Autoencoder

To capture the temporal dependencies in ECG time-series data, we employed a Long Short-Term Memory (LSTM) based autoencoder. This architecture consists of two main components: an encoder and a decoder, both built using LSTM layers.

3.1.1 Encoder

The encoder compresses the input time series into a lower-dimensional latent representation (embedding). It consists of two LSTM layers stacked sequentially:

- **Input**: A sequence of shape (seq_len, n_features).
- **First LSTM Layer**: Takes the input and maps it to a hidden representation with dimensionality $2 \times$ embedding_dim.
- **Second LSTM Layer**: Further processes the sequence and outputs the final hidden state of size embedding_dim.

The final embedding is extracted from the last hidden state of the second LSTM and reshaped accordingly for input into the decoder.

Listing 2: Encoder Architecture

```
class Encoder(nn.Module):
  def __init__(self, seq_len, n_features, embedding_dim=64):
    ...
    self.rnn1 = nn.LSTM(..., hidden_size=2*embedding_dim)
    self.rnn2 = nn.LSTM(..., hidden_size=embedding_dim)
  def forward(self, x):
    ...
    x, (_, _) = self.rnn1(x)
    x, (hidden_n, _) = self.rnn2(x)
    return hidden_n.reshape((self.n_features, self.embedding_dim))
```

3.1.2 Decoder

The decoder reconstructs the original input sequence from the compressed embedding produced by the encoder. It also uses two LSTM layers:

- **Input**: The compressed embedding vector.

- **First LSTM Layer**: Expands the embedding across the time dimension.
- **Second LSTM Layer**: Further processes the sequence to reconstruct temporal patterns.
- **Output Layer**: A linear layer maps the hidden states back to the original feature dimension.

Listing 3: Decoder Architecture

```
class Decoder(nn.Module):
  def __init__(self, seq_len, input_dim=64, n_features=1):
    ...
    self.rnn1 = nn.LSTM(..., hidden_size=input_dim)
    self.rnn2 = nn.LSTM(..., hidden_size=2*input_dim)
    self.output_layer = nn.Linear(2*input_dim, n_features)

  def forward(self, x):
    ...
    x, _ = self.rnn1(x)
    x, _ = self.rnn2(x)
    return self.output_layer(x)
```

3.1.3 Autoencoder Integration

The full LSTM autoencoder integrates the encoder and decoder into a single model. During the forward pass, the input sequence is first encoded and then decoded to reconstruct the original signal.

Listing 4: Autoencoder Architecture

```
class Autoencoder(nn.Module):
  def __init__(self, seq_len, n_features, embedding_dim=64):
    ...
    self.encoder = Encoder(...)
    self.decoder = Decoder(...)

  def forward(self, x):
    x = self.encoder(x)
    x = self.decoder(x)
    return x
```

This LSTM autoencoder learns to reconstruct only normal ECG patterns during training. When presented with abnormal signals, the reconstruction error is expected to increase significantly, serving as the basis for anomaly detection.

3.2 GRU Autoencoder

To explore an alternative to LSTM-based sequence modeling, we also implemented a Gated Recurrent Unit (GRU) based autoencoder. GRUs are known to be more computationally efficient while preserving the ability to model temporal dependencies in time-series data.

The GRU Autoencoder shares a similar structure with the LSTM version, composed of an encoder and a decoder, both utilizing stacked GRU layers.

3.2.1 GRU Encoder

The encoder compresses the time-series input into a fixed-size embedding using two GRU layers.

- **Input**: A sequence of shape (seq_len, n_features).
- **First GRU Layer**: Projects the input into a hidden representation of dimension $2 \times$ embedding_dim.
- **Second GRU Layer**: Reduces the output to a final embedding of dimension embedding_dim.

Listing 5: GRU Encoder Architecture

```
class GRUEncoder(nn.Module):
    def __init__(self, seq_len, n_features, embedding_dim=64):
        ...
        self.gru1 = nn.GRU(input_size=n_features,
        hidden_size=2*embedding_dim)
        self.gru2 = nn.GRU(input_size=2*embedding_dim,
         hidden_size=embedding_dim)

    def forward(self, x):
        x = x.reshape((1, self.seq_len, self.n_features))
        x, _ = self.gru1(x)
        x, hidden_n = self.gru2(x)
        return hidden_n.reshape((self.n_features, self.embedding_dim))
```

3.2.2 GRU Decoder

The decoder reconstructs the original sequence from the encoded representation. It also contains two GRU layers and an output linear layer.

- **Input**: The embedding vector from the encoder.
- **First GRU Layer**: Repeats the embedding across the sequence length and processes it.
- **Second GRU Layer**: Further refines the temporal output.
- **Output Layer**: A fully connected layer maps the hidden output to the original number of features.

Listing 6: GRU Decoder Architecture

```
class GRUDecoder(nn.Module):
    def __init__(self, seq_len, input_dim=64, n_features=1):
        ...
        self.gru1 = nn.GRU(input_size=input_dim, hidden_size=input_dim)
        self.gru2 = nn.GRU(input_size=input_dim, hidden_size=2*input_dim)
        self.output_layer = nn.Linear(2*input_dim, n_features)

    def forward(self, x):
        x = x.repeat(self.seq_len, self.n_features)
        x = x.reshape((self.n_features, self.seq_len, self.input_dim))
        x, _ = self.gru1(x)
        x, _ = self.gru2(x)
        x = x.reshape((self.seq_len, self.hidden_dim))
        return self.output_layer(x)
```

3.2.3 Autoencoder Integration

The GRU autoencoder connects the encoder and decoder into a single end-to-end architecture for reconstructing time-series input.

Listing 7: GRU Autoencoder Integration

```
class GRUAutoencoder(nn.Module):
    def __init__(self, seq_len, n_features, embedding_dim=64):
        ...
        self.encoder = GRUEncoder(seq_len, n_features, embedding_dim)
        self.decoder = GRUDecoder(seq_len, embedding_dim, n_features)

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

Like the LSTM model, the GRU autoencoder is trained to reconstruct normal ECG signals, and high reconstruction errors indicate potential anomalies in the input data.

4 Training Process

We employed two distinct training procedures to train the GRU-based autoencoder, each with different techniques and hyperparameter configurations.

4.1 Train 1:

This training method adopts a traditional epoch-based approach where each sequence from the dataset is processed individually. The model updates its parameters after every single sequence (i.e., sample-wise training). Performance metrics are computed for both the training and validation datasets at the end of each epoch.

Techniques used in Train 1:

- **Optimizer:** Adam optimizer with a fixed learning rate of 1×10^{-3} and a weight decay of 1×10^{-4}.
- **Loss Function:** Mean Absolute Error (L1 Loss), summed over the sequence.
- **Learning Rate Scheduler:** ReduceLROnPlateau, which decreases the learning rate by a factor of 0.5 if the validation loss does not improve for 5 consecutive epochs.
- **Accuracy Metric:** Defined as the percentage of sequence elements where the absolute error is below a tolerance threshold of 0.1.
- **Early Stopping:** Training stops if no improvement in validation loss is observed over 10 epochs.
- **Model Checkpointing:** The best-performing model (based on validation loss) is saved during training.

4.2 Train 2:

The second training approach enhances computational efficiency by using mini-batches and PyTorch's `DataLoader`. While batches are used, each sequence within the batch is still processed individually, and gradients are updated per sample.

Techniques used in Train 2:

- **Optimizer:** Adam optimizer with a lower learning rate of 1×10^{-4} and a weight decay of 1×10^{-5}.
- **Loss Function:** Mean Squared Error (MSE Loss), which penalizes larger errors more strongly.
- **Learning Rate Scheduler:** CosineAnnealingLR with $T_{max} = 10$, providing smooth, periodic decay of the learning rate.
- **Gradient Clipping:** Applied with a maximum norm of 1.0 to prevent exploding gradients in RNNs.
- **Accuracy Metric:** Defined similarly to Train 1, but with a higher tolerance threshold of 0.2.
- **Batch Size:** A mini-batch size of 32 is used.
- **Early Stopping and Checkpointing:** Same logic as Train 1, saving the model with the best validation loss.

5 Model Training Results

5.1 LSTM Autoencoder with Train 1

Convergence Plots

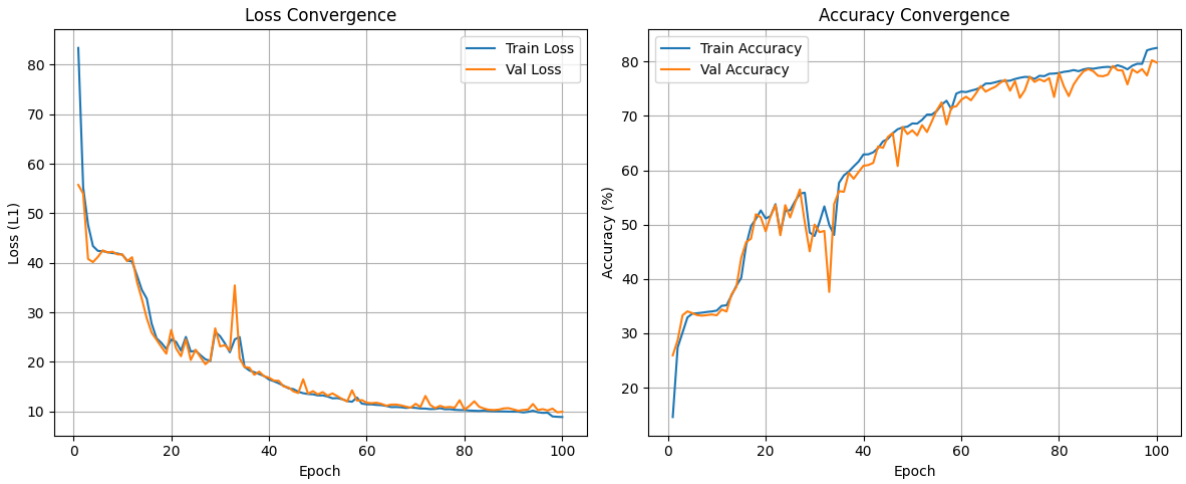

Figure 2: Convergence behavior of the LSTM Autoencoder trained with Train 1. Left: L1 Loss for training and validation sets across epochs. Right: Accuracy evolution on training and validation sets.

Observation. The convergence plots in Figure 2 show that the model exhibits stable learning dynamics. The training and validation losses (L1) decrease consistently over epochs, with minimal overfitting. The validation accuracy follows a similar trajectory to the training accuracy, reaching over 75% by the end of training. These results suggest that the LSTM autoencoder is effectively learning to reconstruct the sequences with generalization to unseen validation data.

5.2 LSTM Autoencoder with Train 2

Convergence Plots

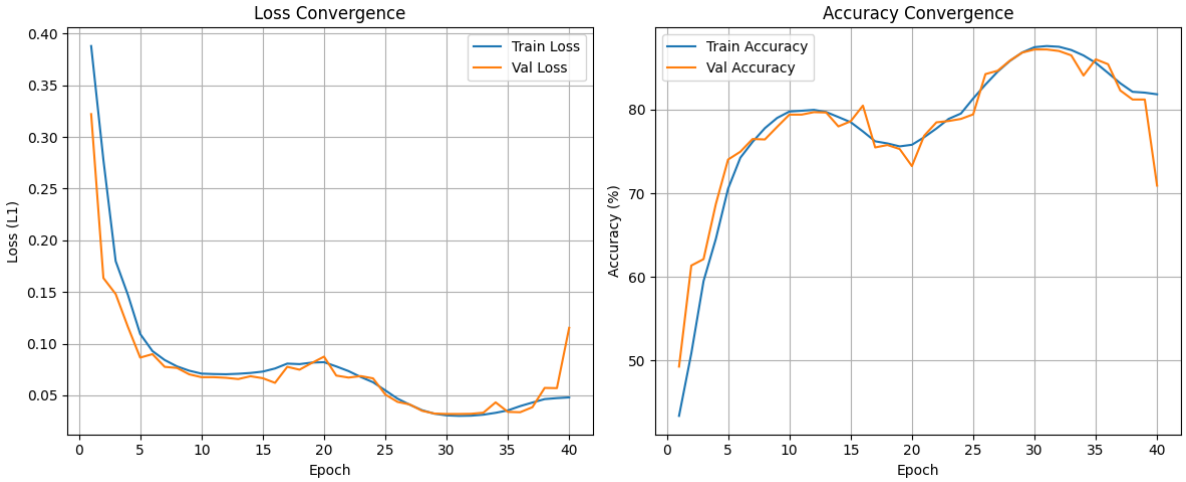

Figure 3: Convergence behavior of the LSTM Autoencoder trained with Train 2. Left: L1 Loss on training and validation datasets. Right: Accuracy over epochs.

Observation. As shown in Figure 3, the model achieves rapid convergence in both training and validation loss within the first few epochs. However, after epoch 30, there is a slight increase in validation loss and a dip in validation accuracy, suggesting early signs of overfitting. Despite this, the model achieves an accuracy peak above 85% before the decline, indicating strong initial generalization performance.

5.3 GRU Autoencoder with Train 1

Convergence Plots

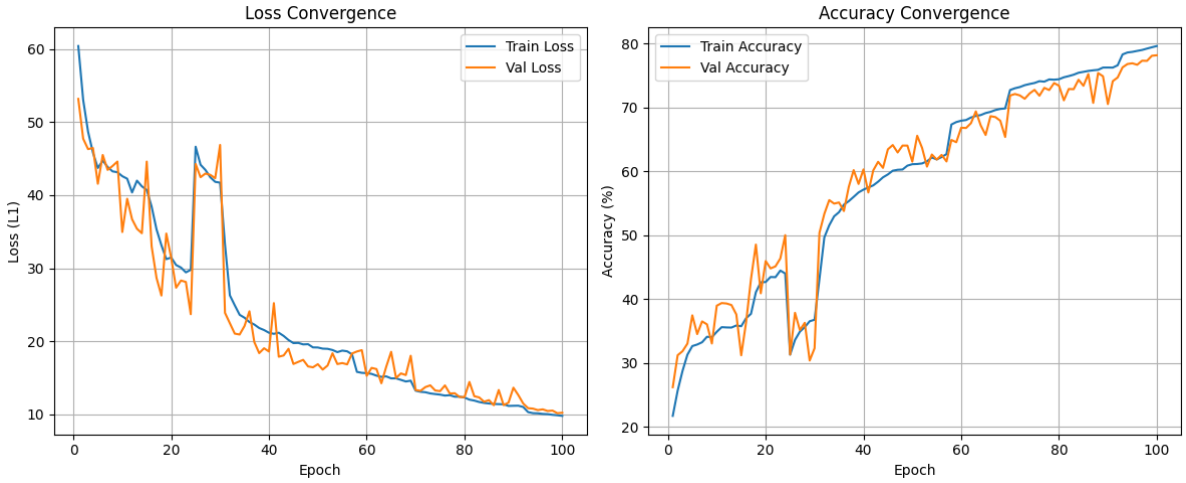

Figure 4: Convergence behavior of the GRU Autoencoder trained with Train 1. Left: L1 Loss on training and validation datasets. Right: Accuracy over epochs.

Observation. As shown in Figure 4, the model achieves rapid convergence in both training and validation loss within the first few epochs. However, after epoch 30, there is a slight increase in validation loss and a dip in validation accuracy, suggesting early signs of overfitting. Despite this, the model achieves an accuracy peak above 85% before the decline, indicating strong initial generalization performance.

5.4 GRU Autoencoder with Train 2

Convergence Plots

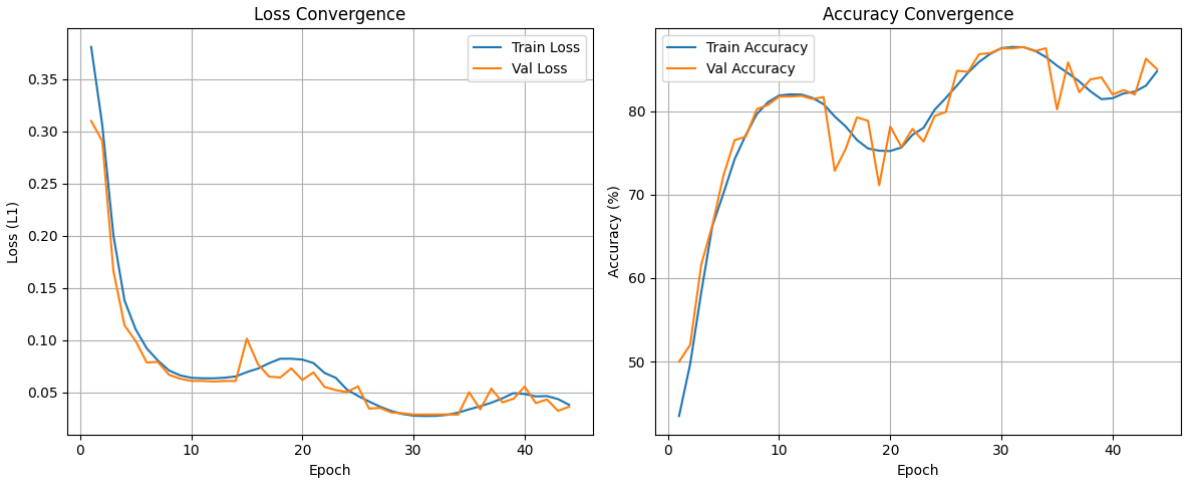

Figure 5: Convergence behavior of the GRU Autoencoder trained with Train 2. Left: L1 Loss on training and validation datasets. Right: Accuracy over epochs.

Observation. As shown in Figure 5, the model achieves rapid convergence in both training and validation loss within the first few epochs, stabilizing around 0.10 L1 loss. The accuracy rises steadily, peaking above 80

6 Validation Results

The table below summarizes the validation performance of each model in terms of loss, accuracy, and reconstruction error statistics.

Model (Training)	**Val. Loss**	**Val. Accuracy (%)**	**Mean Recon. Error**	**Std Recon. Error**
LSTM (Train 1)	9.3420	80.93	0.0667	0.0793
LSTM (Train 2)	15.0456	64.46	0.1075	0.1409
GRU (Train 1)	10.4513	77.62	0.0747	0.0910
GRU (Train 2)	13.7101	68.16	0.0979	0.1272

Table 1: Validation loss, accuracy, and reconstruction error statistics for each model configuration.

7 Test Results

In this phase, the trained models were evaluated on a mixed test set containing both **normal** and **abnormal** data. The evaluation focused on key classification metrics such as precision, recall, F1-score, and overall accuracy. These metrics provide insight into each model's ability to correctly detect anomalies while minimizing false positives and false negatives.

Model	Precision	Recall	F1-Score	Accuracy
LSTM Train 1	0.9965	0.5792	0.7325	0.6024
LSTM Train 2	0.9884	0.5797	0.7307	0.5983
GRU Train 1	0.9942	0.5318	0.6929	0.5568
GRU Train 2	0.9907	**0.6157**	**0.7595**	**0.6332**

Table 2: Performance comparison of all models on the mixed test set (normal and abnormal samples).

8 Discussion

The test results presented in Table 2 allow for a comparative evaluation of the four models: two based on LSTM and two based on GRU, each trained with different data configurations.

- **Precision:** All models demonstrated exceptionally high precision values (above 0.98), indicating a strong ability to correctly classify anomalies with minimal false positives. LSTM Train 1 achieved the highest precision (0.9965), closely followed by GRU Train 1 (0.9942).
- **Recall:** The recall values show greater variability, representing each model's ability to correctly identify actual anomalies. GRU Train 2 obtained the highest recall (0.6157), which is significant as it reflects its capability to detect a larger portion of the anomalies compared to other models.
- **F1-Score:** The F1-score balances precision and recall. GRU Train 2 also achieved the highest F1-score (0.7595), indicating the best trade-off between correctly identifying anomalies while minimizing false alarms.
- **Accuracy:** GRU Train 2 outperformed the others in terms of overall classification accuracy (0.6332), reinforcing its superiority across multiple metrics.

Conclusion: Based on the analysis of precision, recall, F1-score, and accuracy, **GRU Train 2** stands out as the most effective model for anomaly detection in the test set. It offers a balanced performance and demonstrates robustness in detecting anomalies within both normal and abnormal sequences.

9 Conclusion

In this work, we explored the effectiveness of recurrent neural network architectures—LSTM and GRU—for anomaly detection using autoencoders. We trained and evaluated four models under two different training conditions for each architecture, and assessed their convergence, validation performance, and final testing metrics.

Through a comprehensive analysis of validation and test results, we found that all models showed high precision, with significant variation in recall and F1-score. Among them, the GRU-based model trained under the second training configuration (GRU Train 2) achieved the best overall performance. It exhibited the highest recall (0.6157), F1-score (0.7595), and accuracy (0.6332), making it the most suitable candidate for detecting anomalies effectively.

This study demonstrates the advantage of GRU networks in this anomaly detection context, particularly when paired with appropriate training data. Future work can focus on optimizing the reconstruction threshold dynamically and incorporating hybrid or attention-based architectures to further enhance detection accuracy and robustness.