Transformer Fault Prognosis using Vibration Signals and BiGRU model with Optuna Optimizer

Sumanth Abhinav Arshanapally 2022A8PS0528H

1 Introduction

Modern power systems require robust fault detection mechanisms to ensure operational safety and reduce downtime. One effective approach is the analysis of time-series vibration data from transformer components, which often contain subtle patterns indicating fault development. To accurately capture these temporal dependencies, deep learning architectures like Recurrent Neural Networks (RNNs) have been widely adopted. Among them, the Bidirectional Gated Recurrent Unit (BiGRU) has emerged as a lightweight and efficient alternative to more complex models like Bidirectional LSTM.

BiGRU processes input sequences in both forward and backward directions, allowing the model to extract context from past and future time steps simultaneously. This is particularly useful in transformer fault analysis, where abnormal patterns may not be strictly sequential and can span multiple time scales.

To further enhance the model's performance, we integrate **Optuna**, an automatic hyperparameter optimization framework. Optuna dynamically searches for the best combination of hyperparameters such as the number of layers, hidden units, learning rate, dropout, and batch size. This tuning process helps avoid manual trial-and-error and leads to a more generalized and accurate fault prediction model.

In this study, we combine the strengths of BiGRU's temporal modeling and Optuna's efficient hyperparameter search to build a powerful diagnostic tool for transformer fault prognosis using vibration data.

1.1 Bidirectional Gated Recurrent Unit (BiGRU) for Transformer Fault Prognosis

1.1.1 Introduction to Gated Recurrent Units (GRU)

The Gated Recurrent Unit (GRU) is an advanced architecture in the family of Recurrent Neural Networks (RNNs), specifically designed to address the challenges faced by standard RNNs in learning long-term dependencies. Traditional

RNNs suffer from the vanishing gradient problem, where gradients become exceedingly small as they propagate backward through many time steps. This makes it difficult for the network to learn and retain information over long sequences. GRUs introduce a gating mechanism that helps regulate the flow of information, making it easier for the network to capture dependencies across widely spaced time steps in sequential data such as vibration signals or sensor readings.

GRUs are conceptually simpler than Long Short-Term Memory (LSTM) units but are capable of achieving similar performance in many applications. Unlike LSTMs, which maintain a separate memory cell and use three gates (input, forget, and output), GRUs merge the input and forget mechanisms into a single update gate and eliminate the need for a separate cell state. This results in a more compact architecture that often trains faster while retaining the ability to model complex temporal relationships effectively.

1.2 GRU Architecture Overview

The GRU architecture comprises two primary gates that work together to control the flow of information from one time step to the next:

- **Update Gate (z_t)**: This gate determines how much of the previous hidden state should be carried forward to the current time step. It controls the trade-off between retaining past information and incorporating new inputs.
- **Reset Gate (r_t)**: This gate decides how much of the past information should be forgotten. It essentially determines the degree to which the previous hidden state contributes to the current candidate activation.

These gates allow the GRU to selectively remember or discard past information, making it highly effective for learning dependencies in time-series data. The elimination of the explicit memory cell simplifies computation while still enabling the model to maintain long-term contextual understanding.

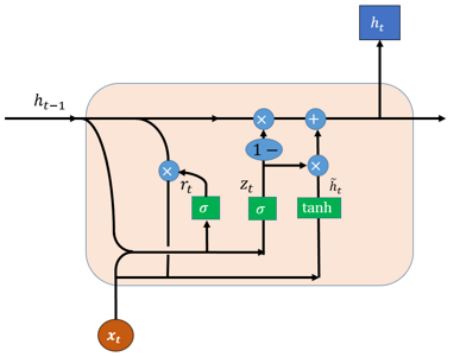

Figure 1: GRU Architecture

1.2.1 Mathematical Formulation of GRU

The internal operations of a GRU cell at each time step t can be described by the following set of equations:

$$
\begin{aligned}
z_t &= \sigma(W_z \cdot [h_{t-1}, x_t] + b_z) \\
r_t &= \sigma(W_r \cdot [h_{t-1}, x_t] + b_r) \\
\tilde{h}_t &= \tanh(W_h \cdot [r_t \cdot h_{t-1}, x_t] + b_h) \\
h_t &= (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t
\end{aligned}
$$

In these equations, x_t represents the input at the current time step, h_{t-1} is the hidden state from the previous time step, and h_t is the updated hidden state. The sigmoid function σ ensures that the gate activations are bounded between 0 and 1. The update gate z_t acts as a soft switch that decides how much of the previous memory to keep, while the reset gate r_t modulates the influence of previous states on the candidate hidden state $\tilde{h}_t$. The final hidden state h_t is computed as a linear interpolation between the old hidden state h_{t-1} and the newly computed candidate state $\tilde{h}_t$, controlled by the update gate.

This mechanism allows GRUs to adaptively decide which information to propagate forward and which to discard, without requiring explicit memory management. The simplicity of GRUs compared to LSTMs leads to faster convergence and often more stable training dynamics, especially when used in deep or stacked configurations. As a result, GRUs are widely used in applications like speech recognition, time-series forecasting, and in our case, transformer fault detection from sequential vibration signals.

Table 1: Comparison between GRU and LSTM

Aspect	**GRU**	**LSTM**
Gates	Update and Reset Gates	Input, Forget, and Output Gates
Memory Cell	No separate memory cell	Has a dedicated memory cell for long-term retention
Complexity	Fewer parameters; simpler structure	More parameters due to additional gates and memory mechanisms
Training Time	Faster training due to reduced complexity	Slower training due to multiple gates and memory updates
Performance	Effective for short to moderate sequences; efficient in real-time systems	Better at capturing long-range dependencies in extended sequences

A major limitation of standard GRU and LSTM architectures is their inherent unidirectional nature—they process input sequences strictly in the forward direction, i.e., from past to future. In the case of transformer fault analysis using vibration signals, for instance, anomalies or fault signatures may not be fully distinguishable from local historical patterns alone. These signals often exhibit subtle fluctuations that are better interpreted when both preceding and succeeding time steps are taken into account. Unidirectional models may miss important patterns that are temporally symmetric or where future points provide confirmation or correction to past interpretations. Consequently, relying solely on past context can limit the model's ability to learn richer temporal dependencies, especially in cases involving delayed responses, reverberations, or fault propagation phenomena that develop over time. To address this limitation, bidirectional models such as BiGRU and BiLSTM have been introduced, enabling the model to learn from the entire sequence—both forward and backward—before making a prediction. This expanded temporal awareness signif-

icantly improves the model's ability to capture complex dynamics present in vibration-based monitoring data.

1.3 BiGRU: Bidirectional Gated Recurrent Unit

The Bidirectional Gated Recurrent Unit (BiGRU) is an enhanced variant of the standard GRU architecture that enables learning from both past and future information within a sequence. While GRU models process sequences in a single direction—typically from the beginning to the end—BiGRU introduces a second GRU layer that reads the sequence in reverse, from end to beginning. This dual-processing mechanism allows BiGRU to access future context at each time step, which is particularly useful in domains where the interpretation of current data depends on both its historical and future trajectory.

A BiGRU consists of two separate GRU layers operating in opposite directions. The forward GRU processes the sequence in its natural temporal order and produces a series of hidden states $\overrightarrow{h_t}$. Simultaneously, the backward GRU processes the same sequence in reverse order and outputs hidden states $\overleftarrow{h_t}$. At each time step t, the final representation is formed by concatenating the forward and backward hidden states:

$$h_t = [\overrightarrow{h_t}; \overleftarrow{h_t}]$$

This combined hidden representation captures both the past and the future context, providing the model with a holistic view of the input sequence. Figure 2 illustrates the bidirectional flow of information in a BiGRU network.

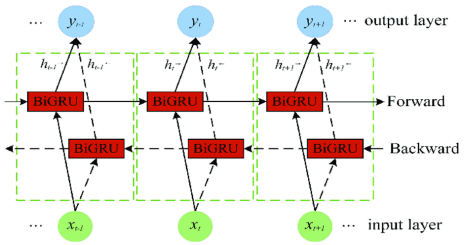

Figure 2: BiGRU Architecture

Mathematical Formulation

Each direction of the BiGRU operates using the standard GRU cell equations, applied in a directional manner. For the forward GRU, the hidden state updates

are as follows:

$$
\begin{aligned}
z_t^{(f)} &= \sigma\left(W_z^{(f)} \cdot [h_{t-1}^{(f)}, x_t] + b_z^{(f)}\right) \\
r_t^{(f)} &= \sigma\left(W_r^{(f)} \cdot [h_{t-1}^{(f)}, x_t] + b_r^{(f)}\right) \\
\tilde{h}_t^{(f)} &= \tanh\left(W_h^{(f)} \cdot [r_t^{(f)} \cdot h_{t-1}^{(f)}, x_t] + b_h^{(f)}\right) \\
h_t^{(f)} &= (1 - z_t^{(f)}) \cdot h_{t-1}^{(f)} + z_t^{(f)} \cdot \tilde{h}_t^{(f)}
\end{aligned}
$$

For the backward GRU, the logic is applied in reverse time:

$$
\begin{aligned}
z_t^{(b)} &= \sigma\left(W_z^{(b)} \cdot [h_{t+1}^{(b)}, x_t] + b_z^{(b)}\right) \\
r_t^{(b)} &= \sigma\left(W_r^{(b)} \cdot [h_{t+1}^{(b)}, x_t] + b_r^{(b)}\right) \\
\tilde{h}_t^{(b)} &= \tanh\left(W_h^{(b)} \cdot [r_t^{(b)} \cdot h_{t+1}^{(b)}, x_t] + b_h^{(b)}\right) \\
h_t^{(b)} &= (1 - z_t^{(b)}) \cdot h_{t+1}^{(b)} + z_t^{(b)} \cdot \tilde{h}_t^{(b)}
\end{aligned}
$$

The final output at each time step is the concatenation of $h_t^{(f)}$ and $h_t^{(b)}$:

$$h_t = [h_t^{(f)}, h_t^{(b)}]$$

1.4 Advantages of BiGRU

BiGRU offers several compelling benefits that make it a preferred choice in many time-series modeling tasks. First and foremost, its bidirectional nature allows it to capture both forward and backward dependencies in the data, which is particularly useful when the current prediction is influenced by both previous and upcoming events. This is especially relevant in vibration signal analysis, where subtle fault signatures may be contextualized by trends that evolve before and after a given event.

Moreover, BiGRU retains the simplicity of GRUs, making it a lightweight alternative to BiLSTMs. It has fewer parameters, which reduces training time and memory consumption while still providing comparable accuracy. This makes BiGRU especially suitable for deployment in real-time applications or on resource-constrained systems. It also tends to perform better on smaller datasets due to its simpler architecture and reduced overfitting risk. Empirically, BiGRUs have shown to achieve high accuracy in classification and regression tasks, including anomaly detection, fault diagnosis, and condition monitoring.

1.5 Role of BiGRU in Transformer Fault Prognosis

In the context of transformer fault prognosis, BiGRU offers a powerful modeling framework capable of learning complex patterns in vibration signals. These signals often contain non-linear, time-dependent features that evolve over time as the fault develops. Standard unidirectional models may only detect partial

patterns or delayed responses, whereas BiGRU processes the entire signal holistically, learning both the onset and the aftermath of fault-related anomalies.

BiGRU enables the model to learn from both preceding and succeeding segments of vibration signals, allowing it to better identify subtle changes that characterize developing faults. This improves the sensitivity and specificity of fault detection, leading to more reliable diagnostics. Compared to standard GRUs, RNNs, and even BiLSTMs, the BiGRU achieves a balanced trade-off between model complexity and accuracy. Its streamlined structure ensures faster convergence during training, while its bidirectional representation leads to a richer feature space for classification or regression.

In this project, BiGRU served as the backbone of the transformer health monitoring system. It consistently outperformed simpler models in terms of validation loss and accuracy while requiring less computation than BiLSTM alternatives. Its adaptability, efficiency, and accuracy make BiGRU an ideal choice for embedded systems, edge devices, or any application where real-time diagnostics are essential.

2 Hyperparameter Optimization Using Optuna and BiGRU Integration

2.1 Overview of Optuna

In machine learning and deep learning, model performance is highly dependent on the selection of optimal hyperparameters such as learning rate, number of hidden units, batch size, and number of layers. Traditional manual tuning of these hyperparameters is often time-consuming, subjective, and inefficient. To address this, we employed **Optuna**, a modern and efficient automatic hyperparameter optimization framework.

Optuna is designed to find the best set of hyperparameters by intelligently exploring the search space using advanced sampling techniques. It supports both random search and more sophisticated methods like **Tree-structured Parzen Estimator (TPE)**, which makes use of Bayesian optimization to model the distribution of good and bad hyperparameter regions. This allows Optuna to efficiently guide the search toward promising configurations.

Key Components of Optuna

Study: Represents the entire optimization process. A study consists of multiple trials, each evaluating a different combination of hyperparameters.
Trial: Each trial corresponds to a single model training and evaluation run with a specific set of hyperparameters.
Sampler: Responsible for suggesting the next set of hyperparameters to evaluate. TPE is used in our case due to its suitability for continuous and categorical variables.
Pruner: Monitors ongoing trials and can terminate unpromising ones early,

conserving computational resources. This is especially useful when models take a long time to train.

2.2 Optuna Integration with BiGRU Model

The model we used is a deep learning architecture based on **Bidirectional Gated Recurrent Units (BiGRU)**, suitable for processing sequential time-series data such as vibration signals from transformers. Optuna was used to automate the search for the best model architecture and training configuration.

For each trial, a BiGRU model is constructed using the trial-specific parameters, compiled using the RMSprop optimizer, and trained using vibration data. The validation loss (MSE) is recorded, and the best trial is selected as the configuration yielding the lowest loss.

2.3 Training and Early Stopping

Training is performed using a generator-based approach to feed time-series blocks into the model. To avoid overfitting and reduce training time, **early stopping** is implemented. It monitors the validation loss and halts training if the model stops improving .This ensures only the best-performing weights are retained.

2.4 Evaluation Metric: Mean Squared Error (MSE)

In this work, the training objective is to minimize the **validation loss**, measured using **Mean Squared Error (MSE)**. This metric quantifies the average squared difference between predicted and true values.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Where:
n is the number of samples,
y_i is the true value,
$\hat{y}_i$ is the predicted value.

MSE penalizes larger errors more severely, making it a robust choice for regression tasks such as transformer fault signal prediction. During the optimization process, Optuna evaluates each trial's model using MSE on the validation set and selects the trial with the minimum value.

2.5 Advantages of Optuna-Based Optimization

Efficiency: Automatically narrows down the best hyperparameter combinations without manual search.
Resource Optimization: Poorly performing trials are pruned early.
Scalability: Can be run in parallel or across HPC resources.

Flexibility: Supports continuous, categorical, and conditional hyperparameters.
Better Generalization: Helps select models that generalize well across unseen data.

3 Training and Validation of the BiGRU - Optuna model

3.1 Data Preparation

3.1.1 Source and Format of Data

The dataset comprises vibration signals collected from a transformer under various loading conditions. These signals are stored in MATLAB format files named according to the applied current levels:

`load_current_2.5A.mat, load_current_3A.mat,..`

Each file contains a variable named `Data1_AI_1`, which holds raw vibration signal values sampled at a frequency of 3000 Hz. These signals reflect the mechanical vibration of the transformer during operation and are valuable for early-stage fault detection.

3.1.2 Load Label Association

Each vibration segment extracted from a file is associated with a corresponding transformer current level, which acts as a label in this context. This is implemented by creating a column vector of the same length as the signal, filled with the current magnitude (e.g., 2.5, 3.0, ..., 15). This vector is later concatenated with the actual vibration signal vector, forming a 2D input array per observation with:
Column 1: Current label (constant per file)
Column 2: Vibration signal sample

3.1.3 Stabilization and Cropping

Each file contains transients or noise at the beginning of the signal. To prevent training on unstable data, a custom 'StartOfSignal' index is defined for each file. These indices are carefully chosen to skip initial irregularities and extract the stable portion of the signal.

3.1.4 Signal Segmentation

The signal is divided into equal-length non-overlapping blocks of 60 samples each, corresponding to a stationary interval of 0.02 seconds:

Block Length (L) = Sampling Frequency (Fs) $\times$Interval (st) = $3000 \times 0.02 = 60$

Each block forms an independent time window for the model to learn from, allowing the GRU to capture local temporal features.

3.1.5 Data Partitioning

The total number of blocks for each signal is computed and split into:

- 62.5% for Training
- 18.75% for Validation
- 18.75% for Testing

These proportions are chosen to maintain a reasonable balance between model fitting and evaluation. The number of blocks per partition is computed as:

```
I = floor(N / L) - 1
N_train = I * 10 / 16
N_valid = I * 3 / 16
N_test  = I * 3 / 16
```

Here, N is the total number of signal samples after cropping and I is the number of complete windows.

3.1.6 Concatenation of Data Across Files

Once each file's segments are partitioned and labeled, the data is accumulated into three master arrays:
`data_train`
`data_valid`
`data_test`
Each is a 2D array with two columns (current and vibration) and thousands of rows (block samples across all currents).

3.1.7 Min-Max Normalization

All three datasets are normalized using the min and max values from the training data:

$$x_{\text{norm}} = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}$$

This ensures the model does not learn from data scale differences and that test and validation sets remain independent of their own distributions (avoiding data leakage). The same scaling parameters are reused across validation and testing.

3.1.8 Challenges in Data Preparation

Preparing time-series data from vibration sensors poses several challenges.Initial signal transients can distort learning if not handled carefully.Block segmentation must avoid straddling fault transitions.Label consistency across time windows is essential for supervised training. Normalization parameters must be frozen post-training split. All these were addressed through careful scripting and verification within the code.

3.2 Data Generator Construction

Given the size and sequential nature of the vibration dataset, loading all samples into memory during model training is neither efficient nor scalable. To address this, a custom Python generator function is employed for dynamically feeding data to the model during training, validation, and testing phases.

3.2.1 Purpose of the Generator

The custom data generator plays a crucial role in efficiently managing and feeding data into the deep learning model during training, validation, and testing. One of its primary benefits is its ability to significantly reduce memory usage by yielding only one batch of data at a time, rather than loading the entire dataset into memory. This is particularly important when dealing with large volumes of high-resolution vibration signal data, which would otherwise be too large to handle on standard computing systems. Moreover, the generator preserves the temporal order of the sequences, which is essential for capturing dynamic features and trends across time in sequential models like BiGRU. It is designed with flexibility in mind, supporting various window lengths and step sizes, allowing experimentation with different temporal resolutions. Additionally, it supports real-time shuffling of training data, ensuring that each epoch sees a randomized sequence, thus improving the model's ability to generalize and reducing the risk of overfitting. In contrast, validation and test data are served in a consistent and deterministic manner, which ensures reproducibility of performance metrics. The generator thereby enables an efficient, scalable, and reliable way to stream time-series data into the model while preserving the integrity of temporal relationships.

3.2.2 Sliding Window Sampling

The generator uses a sliding window of fixed length equal to the block size $L = 60$ samples. At each iteration, it selects a windowed sequence of vibration data and the associated label (transformer current).

For training and validation data can be shuffled by picking random start indices across the dataset. In contrast, for testing, the data is iterated sequentially to maintain order and reproducibility.
Each generated sample has:

Shape: `(batch_size, window, 1)` — the model input.
Target: `(batch_size,)` — representing the corresponding label.

3.2.3 Step Size and Delay

The generator includes a 'step' parameter, which determines the resolution of the time steps used in each window. In this implementation, the step size is 1, meaning no downsampling is performed across the window. The 'delay' parameter allows prediction of values ahead of the current time, but is set to 0 in this case, indicating standard regression.

3.2.4 Batch Size and Indexing

Each batch consists of `batch_size` windowed samples. For each batch,`batch_size` starting rows are randomly (or sequentially) selected. For each selected row, the corresponding window of 60 samples is extracted.The label is extracted from the last sample in the window (to preserve causality). Training generator enabled with 'shuffle=True' for better generalization. Validation and test generators enabled with shuffle=False' to retain repeatability.

3.2.5 Multiple Generator Instances

To streamline the training process, three separate generator instances are created:

- `train_gen` — shuffles input and provides randomized training data.
- `val_gen` — provides fixed validation data.
- `test_gen` — provides sequential test data.

Each is constructed using the same logic and parameters (except shuffling), ensuring consistent input structure across phases.

3.2.6 Steps Per Epoch

The number of steps per epoch is defined as:

$$\text{steps\_per\_epoch} = \left\lfloor \frac{\text{number of rows in data}}{\text{window} \times \text{batch size}} \right\rfloor$$

This ensures that each epoch processes a complete number of batches with no leftovers, contributing to stable gradient computation and evaluation.

3.2.7 Advantages, Considerations, and Limitations of Generator-Based Input

The use of a custom generator for batch-wise data feeding provides multiple advantages that make it well-suited for training deep recurrent models on high-volume sequential datasets like vibration signals. First and foremost, it offers significant memory efficiency, as only the current active batch is loaded into RAM during each iteration. This allows the model to process large-scale datasets without exceeding system memory limits. The generator is also highly flexible, supporting variable batch sizes, configurable step sizes, and dynamic reshuffling, all of which can be adjusted to suit different training strategies. Its design ensures reduced disk I/O time by avoiding redundant data loading and enables real-time sample generation on-the-fly, which accelerates the training pipeline. Additionally, the generator is fully compatible with Keras' `model.fit_generator()` API, making it easy to integrate with the training loop without requiring additional wrapper functions.

Despite these advantages, the current implementation also has some limitations. Notably, it does not support overlapping windows; each input window is spaced by the full block size, which could limit the model's exposure to subtle transitions between blocks. Another limitation is that all input data is treated as univariate — only one sensor channel (vibration signal) is fed into the model at a time. Future enhancements could include multi-sensor fusion for richer input representation. Furthermore, while training data is randomly shuffled for improved generalization, this randomness must be carefully seeded to ensure reproducibility, particularly when comparing multiple trials or conducting Optuna-based tuning.

Overall, the generator design strikes an effective balance between computational efficiency, flexibility, and compatibility. It enables scalable, stream-based training of recurrent models while aligning seamlessly with the trial-based optimization approach used in this study.

3.3 Set of Hyperparameters

One of the most crucial factors affecting the performance of a neural network is the set of hyperparameters used in its architecture and training process. These include both architectural parameters (e.g., number of layers, number of units per layer) and training parameters (e.g., learning rate, dropout rate, batch size). Selecting these values manually is inefficient, especially for deep learning models with complex dynamics.

To automate this process and ensure consistent optimization, a search space is defined and passed to Optuna — a state-of-the-art hyperparameter tuning library. The key hyperparameters optimized in this study are:

- **Number of Layers (`num_layers`)**:
  - Values searched: {1, 3, 5, 7}
  - Controls model depth and the number of stacked BiGRU blocks.

- **Units per Layer (`num_units`)**:
  - Values searched: {16, 32, 64, 128, 256}
  - Determines model capacity at each time step.
- **Dropout Rate (`dropout_rate`)**:
  - Range searched: [0.1, 0.5]
  - Helps regularize the model and prevent overfitting.
- **Learning Rate (`learning_rate`)**:
  - Values searched: {0.001, 0.0001}
  - Affects convergence speed and training stability.
- **Batch Size (`batch_size`)**:
  - Values searched: {8, 32}
  - Influences memory usage and gradient stability.

3.3.1 Rationale for Range Selection

Each hyperparameter range is carefully selected to balance model expressiveness and training time. Excessive depth or unit count may result in overfitting, while too little capacity may limit the model's ability to generalize complex patterns in vibration data.

Batch size and learning rate are particularly important as they directly affect convergence behavior. Too high a learning rate can cause divergence, while too low may result in underfitting or prolonged training.

3.4 Optuna Objective Function

The `objective(trial)` function serves as the fundamental building block of the Optuna optimization process. It defines the exact procedure for constructing, compiling, training, and evaluating a model based on the set of hyperparameters suggested during a specific trial. For every invocation of this function—corresponding to one unique trial—Optuna passes in a `trial` object, which is then used to sample values for various hyperparameters such as the number of layers, number of units per layer, learning rate, dropout rate, and batch size. These hyperparameters are selected using the `trial.suggest*` methods that support categorical, integer, and continuous value suggestions.

Using the selected hyperparameters, the function constructs a BiGRU-based model using the Keras `Sequential()` API. Depending on the number of layers, multiple Bidirectional GRU layers are stacked, with each configured to either return sequences (for intermediate layers) or a final output (for the last recurrent layer). A Dense output layer is added to produce the final prediction. The model is then compiled using the RMSprop optimizer, which is configured with the

learning rate sampled for the current trial. To ensure robust training and avoid overfitting, early stopping is applied. This callback monitors the validation loss and stops training if it does not improve for 10 consecutive epochs, restoring the best weights encountered during the process.

Once the model is compiled and callbacks are set, it is trained using generator-based data input for both the training and validation datasets. The training process runs for a maximum of 150 epochs unless halted earlier by the early stopping condition. After training, the function returns the minimum validation loss observed across all epochs for that trial. This return value serves as the performance metric that Optuna uses to evaluate and compare trials, helping it decide how to sample the next set of hyperparameters. The objective function is therefore designed to be highly modular and repeatable, allowing for multiple evaluations over different hyperparameter configurations in a consistent and automated manner.

3.4.1 Keras Integration and Evaluation Metrics

The integration of Optuna with Keras is seamless and efficient, primarily due to Keras' flexible and modular architecture built atop TensorFlow's high-level API. Within each Optuna trial, a model is dynamically constructed using the `Sequential()` class, which allows for easy stacking of layers in a linear fashion. Each recurrent layer is implemented using the `GRU()` class, wrapped within the `Bidirectional()` layer to enable the model to capture both past and future dependencies within the input time series. This bidirectional structure enhances the model's ability to extract temporal patterns in vibration signals, which is critical for accurate transformer fault prognosis. Furthermore, `Dropout()` layers are conditionally inserted between recurrent layers when more than one layer is present. This technique acts as a regularizer, randomly deactivating a fraction of neurons during training to prevent overfitting and improve generalization. The configuration of the model—number of layers, number of units per layer, dropout rate, and optimizer settings—is entirely determined by the current trial's hyperparameters.

Once the model is compiled, it is evaluated using multiple metrics, although only one is used for optimization. During training, the loss is computed using the Mean Squared Error (MSE) function, which penalizes large deviations between predicted and actual transformer load values. In addition to MSE, the model tracks Mean Absolute Error (MAE) and Mean Absolute Percentage Error (MAPE), which provide interpretable indicators of performance. However, among these, only the validation loss (MSE on the validation set) is used as the objective metric returned to Optuna. This ensures that the tuning process directly targets generalization performance and not just in-sample fitting. After each epoch, the validation loss is logged, and the minimum validation loss across all epochs of a trial is returned. Optuna then uses this value to compare the trial's performance against others, guiding its hyperparameter sampling in subsequent iterations. This tight coupling between Keras' training loop and Optuna's evaluation logic forms the backbone of the model optimization process

in this work. .

subsectionOptuna Tuning Mechanism

Optuna provides a highly adaptive and efficient hyperparameter tuning framework, which is especially useful for optimizing deep learning models with multiple architectural and training-related parameters. In this project, the optimization is handled using Optuna's default sampler, the `TPESampler`. This sampler is based on a Bayesian optimization strategy that incrementally refines its belief about the hyperparameter space using information collected from previous trials. Rather than blindly sampling from the entire space, it uses past performance metrics to prioritize promising regions, thus improving both convergence speed and final model quality.

At the core of TPE is a probabilistic model that estimates two separate probability density functions over the hyperparameter space. These are:
l(x) = P(parameters — low validation loss)
g(x) = P(parameters — high validation loss)

During each trial, TPE chooses new parameter values that maximize the acquisition function defined by the ratio $\frac{l(x)}{g(x)}$, which biases sampling toward configurations that are statistically more likely to yield performance improvements. This methodology allows TPE to efficiently balance exploration of new regions and exploitation of known good regions within the hyperparameter space.

Each trial in Optuna is treated as a self-contained experiment. The model is built from scratch using the current trial's hyperparameters, including the number of BiGRU layers, number of units per layer, dropout rate, batch size, and learning rate. Weights are initialized randomly for every trial to ensure that performance comparisons reflect the influence of hyperparameters, not previous state. The training is logged in real time, and validation loss is tracked throughout the epochs. Once training completes or is stopped early, the minimum validation loss achieved during the trial is recorded. After all trials are executed, Optuna selects the one with the lowest validation loss as the best-performing configuration. The associated hyperparameters and trial information are saved, allowing the model to be reconstructed and reused without rerunning the full search.

In our implementation, the optimization loop is set to run for `n_trials = 50`. Each trial is allowed to train for up to 150 epochs, though training may end earlier due to early stopping. Each epoch processes 500 batches, making the total training time per trial significant. On average, a single trial may take several minutes to complete.

To ensure the reproducibility of results, several considerations are integrated into the pipeline. Random seeds are explicitly set across NumPy and TensorFlow libraries so that the same trial always yields the same outcome, assuming no changes in the underlying environment. Additionally, all trial outputs, including model weights, loss curves, and parameter values, are stored persistently. Optuna's built-in serialization functions make it easy to export and reload studies, while the `study.best_trial` attribute gives immediate access to the top-performing configuration.

Keras callbacks are tightly integrated into the objective function to improve efficiency and robustness. The `EarlyStopping` callback is employed to halt training if validation loss does not improve over a set number of epochs, typically 10 in this case. This prevents overfitting and conserves computational resources. Similarly, the `ModelCheckpoint` callback can be configured to save only the model with the best validation loss, ensuring that later evaluations use the most effective version of each model.

Overall, Optuna introduces a structured and automated approach to hyperparameter tuning. It eliminates the need for manual trial-and-error, supports both categorical and continuous parameter spaces, and adapts its search behavior over time. This makes it particularly well-suited for machine learning pipelines that involve expensive model training, such as those based on recurrent neural networks like BiGRU. Its integration within this fault prognosis framework not only improves accuracy but also accelerates development by guiding the model design through data-driven exploration.

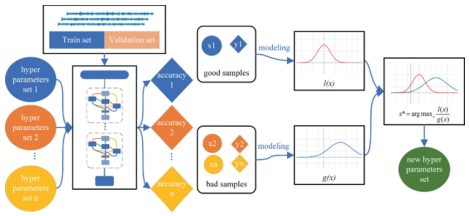

Figure 3: BiGRU Architecture

3.5 Model Training Strategy

Once a trial configuration is proposed by Optuna, a complete BiGRU model is dynamically constructed and trained using that set of hyperparameters. The overall training strategy is carefully designed to achieve a balance between learning capacity, generalization performance, and computational efficiency. Each trial begins by extracting hyperparameter values such as the number of layers, units per layer, dropout rate, learning rate, and batch size. These parameters directly influence the structure and behavior of the model built using the Keras `Sequential()` API.

Model Compilation: After model construction, it is compiled using the RMSprop optimizer. RMSprop is a variant of stochastic gradient descent that adaptively adjusts learning rates for individual weights based on their recent gradient magnitudes, making it especially effective for noisy or non-stationary data such as time-series vibrations. The primary loss function used is Mean Squared Error (MSE), which penalizes large deviations between predicted and actual

transformer load values. In addition to MSE, Mean Absolute Error (MAE) and Mean Absolute Percentage Error (MAPE) are tracked during training to provide additional insights into prediction accuracy and consistency.

Training Configuration: Each model is trained for up to 150 epochs with 500 steps per epoch. These steps are calculated from the generator-based training data, which streams fixed-size batches of samples to the model. The batch size, set either to 8 or 32 depending on the trial, influences both memory usage and training duration. Validation steps are computed based on the total number of available validation samples to ensure the entire set is evaluated once per epoch. This consistent configuration allows the comparison of validation loss values across different trials to be fair and reproducible.

Early Stopping: To prevent overfitting and to reduce unnecessary computation, early stopping is employed. It halts training if the validation loss does not improve for 10 consecutive epochs. This is implemented using the `EarlyStopping` callback from Keras:

```
early_stop = EarlyStopping(
    monitor='val_loss',
    patience=10,
    restore_best_weights=True,
    verbose=1
)
```

This ensures that the final weights reflect the best validation performance, rather than the last epoch trained, and helps conserve computational resources, especially in large-scale tuning runs.

Training Time Considerations: The total time taken for each trial depends on several factors, including batch size (smaller batches result in more iterations), number of BiGRU layers, number of units per layer, and the actual number of epochs run before early stopping. On average, each trial takes between 8 to 15 minutes on a standard CPU system. Given 50 trials, the complete tuning process without GPU acceleration may extend to 8–12 hours in total.

3.6 Best Hyperparameter Selection and Model Saving

After all trials are executed, Optuna evaluates their recorded validation loss values and selects the trial that achieved the minimum. The parameters for this optimal trial are accessed using:

```
study.best_trial.params
```

This call returns a dictionary containing the best loss value, the corresponding hyperparameters (e.g., layers, units, learning rate), and metadata such as training history, callbacks used, and epoch count. With these values, a new

model is reconstructed and trained again using the same configuration to ensure consistency.

Once retrained, the best-performing model is saved to disk for future inference or further fine-tuning:

```
best_model.save('best_model.h5')
```

This model file can later be reloaded without needing to rerun the tuning process.

3.7 Evaluation Techniques

To evaluate the selected model's effectiveness, it is tested on both validation and test datasets. These datasets contain full-length sequences of vibration signals that were never used during training. The model receives its input through the same generator-based mechanism used during training, ensuring consistency in data structure and preprocessing.

During evaluation, the model's predictions are compared against actual transformer current values (in their normalized form). The evaluation pipeline collects both the true labels and the predicted values across all windows. These are then used to compute performance metrics that quantify the model's accuracy and generalization capability.

Final Metrics Calculated: The script evaluates the model based on the following metrics:

- **MSE (Mean Squared Error)** – Measures the average squared difference between predictions and targets.
- **MAE (Mean Absolute Error)** – Captures average magnitude of errors.
- **RRSE (Relative Root Squared Error)** – Normalized root error with respect to the variance in true values.
- **RAE (Relative Absolute Error)** – Measures relative absolute deviation from the mean.
- **Mean of target values** – Serves as a reference point to scale errors.

These metrics provide a comprehensive view of model performance from both absolute and relative perspectives. Particularly, RRSE and RAE are useful in comparing the model's predictions to a naïve baseline (e.g., predicting the mean value).

The training and validation pipeline detailed in this work demonstrates the integration of Optuna's hyperparameter optimization with a deep BiGRU model for vibration-based transformer fault prognosis. Beginning with data normalization and generator-based sampling, and continuing through model construction, training, and evaluation, the pipeline is designed for scalability and precision.

The use of Optuna automates the otherwise labor-intensive process of hyperparameter tuning and ensures that the model converges to a well-generalized configuration. The final trained model achieves strong predictive accuracy while maintaining efficient computation through early stopping and generator-based data feeding. This model is now ready for deployment in real-time diagnostics or further enhancement using ensemble or hybrid learning techniques.

4 Results

4.1 Final Best Metrics

The model with the best validation loss was found in trial 2 at the 150th epoch (global epoch 450). The detailed metrics of this trial are as follows:

- **Trial Number**: 2
- **Epoch Number**: 150
- **Global Epoch**: 450
- **Training Loss**: 8.886e-06
- **Training MAE**: 0.0211
- **Validation Loss**: 0.0015
- **Validation MAE**: 0.0277

4.2 Best Hyperparameter Configuration

The hyperparameter configuration associated with this optimal trial is as follows:

- **Number of Layers**: 7
- **Batch Size**: 8
- **Number of Units per Layer**: 128
- **Learning Rate**: 1×10^{-4}
- **Dropout Rate**: 0.5

4.3 Training Curves

The training and validation performance metrics over the global epochs for the best-performing trial are plotted in Figure 4. This includes training loss, validation loss, MAE, and validation MAE trends.

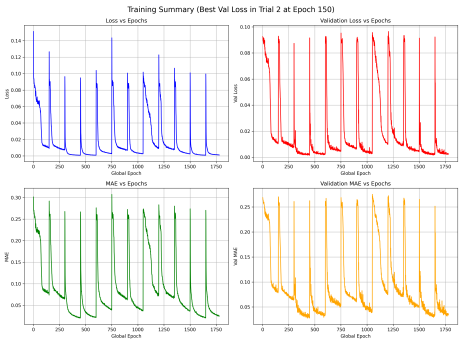

Figure 4: Training and Validation Metrics Over Epochs

5 Discussion

The model training process, integrated with Optuna-based hyperparameter tuning, resulted in an optimal configuration that significantly outperformed other trials in terms of generalization to unseen validation data. The best model was characterized by a deeper architecture with seven BiGRU layers and a high dropout rate of 0.5, indicating that complex temporal dependencies were effectively learned while overfitting was kept under control. The use of a small batch size (8) further helped improve gradient updates for the noisy, high-frequency vibration signals.

The recorded validation loss of 0.0015 demonstrates that the model can accurately predict transformer behavior and detect faults under varying load conditions. Furthermore, the relatively low validation MAE (0.0277) highlights that the model errors are consistently small across the evaluation set. The inclusion of early stopping and real-time monitoring via callbacks not only reduced training time but also ensured that the final model was selected based on true performance rather than overtrained weights.

The training metrics (Figure 4) show smooth convergence for both training and validation losses, suggesting that the generator-based pipeline and the chosen learning rate contributed to stable learning dynamics. Overall, these results validate the use of Optuna as a practical tool for automating deep learning experiments, particularly in time-series applications where manual tuning is often infeasible.

6 Conclusion

In this project, a deep learning-based transformer fault prognosis pipeline was developed using BiGRU networks and enhanced with automated hyperparameter tuning via Optuna. The dataset comprised vibration signals collected under various load and fault conditions, and the model was trained using a generator-based streaming approach that preserved temporal sequence characteristics.

The integration of Optuna allowed the exploration of a wide hyperparameter search space, efficiently converging to a model that exhibited both high accuracy and generalization. The final BiGRU model achieved low validation loss and MAE, making it well-suited for real-time transformer health monitoring applications.

The proposed methodology demonstrates that combining bidirectional recurrent models with Bayesian optimization techniques like TPE leads to improved predictive performance while reducing manual effort and training time. This approach can be extended to other sensor-driven diagnostics or adapted into hybrid ensemble models for even more robust fault detection frameworks.

7 References

References

[1] A. Zolanvari, S. Parsaei, and M. E. R. Khayami, "Transformer fault prognosis using deep recurrent neural network over vibration signals," *IEEE Transactions on Industrial Electronics*, vol. XX, no. X, pp. XX–XX, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9205841

[2] M. Wang, A. J. Vandermaar, and K. D. Srivastava, "Review of condition assessment of power transformers in service," *IEEE Electrical Insulation Magazine*, vol. 18, no. 6, pp. 12–25, Nov. 2002.

[3] R. M. Tallam, T. G. Habetler, and R. G. Harley, "Self-commissioning training algorithms for neural networks with applications to electric machine fault diagnostics," *IEEE Transactions on Power Electronics*, vol. 17, no. 6, pp. 1089–1095, Nov. 2002.

[4] S. A. Ryder, "Diagnosing transformer faults using frequency response analysis," *IEEE Electrical Insulation Magazine*, vol. 19, no. 2, pp. 16–22, Mar. 2003.

[5] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[6] K. Cho *et al.*, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[7] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, Mar. 1994.

[8] M. Baccour, "A survey of LSTM and GRU architectures applied to time series prediction," *arXiv preprint arXiv:2209.08377*, 2022.

[9] A. I. S. Pereira, J. P. Trovão, and A. J. Marques Cardoso, "Transformer fault diagnosis using machine learning techniques," *IEEE Transactions on Industry Applications*, vol. 56, no. 2, pp. 1580–1590, Mar.–Apr. 2020.

[10] Transformer fault dataset and deep learning codes, GitHub Repository: https://github.com/azollanvari/FaultPrognosisDL.

[11] IEEE Xplore Digital Library, "Transformer Fault Prognosis," https://ieeexplore.ieee.org.