App Interface for Data Visualization and Fault Detection

Contents

List of Figures

Chapter 1

Introduction

1.1 Objective

The primary objective of this project was to develop an intuitive and **interactive web-based application** to **monitor** and **visualize** real time chemical plant data(which can be process measurements and manipulated variables) for **real time fault detection** in industrial process control systems.

Traditional approaches often fall short due to their reliance on predefined thresholds or expert knowledge, which may not generalize well across diverse environments. With the in- creasing complexity of dynamic systems, there is a pressing need for automated and robust fault detection mechanisms capable of processing large volumes of data and detecting anomalies with high accuracy.

The challenge was to create a **scalable**, **efficient**, and **user-friendly** interface that could handle continuous data streams from sensors, apply advanced machine learning techniques for fault classification, and present results in an accessible graphical format for engineers and analysts.

The problem originated from the need for automated fault detection in **Tennessee Eastman Process (TEP)** data, a standard dataset used for benchmarking fault diagnosis algorithms in industrial settings. Traditional approaches required manual inspection of sensor logs, leading to delayed responses, inefficiencies, and increased operational costs. Our solution aimed to automate this process, providing instant fault detection and comprehensive summary reports.

Chapter 2

Scope

2.1 Duration

The Project spanned over 3+ months with continuous enhancements and refinements.

2.2 Development Phases

2.2.1 Initialize Panel & Pygame

Task: Set up Panel and Pygame for user interaction and real-time image rendering.
Explanation: Panel provides an interactive dashboard with buttons and controls to manage the simulation and Pygame is used to generate the graphical interface where data will be displayed dynamically inside predefined boxes.

2.2.2 Define Box Coordinates

Task: Store the coordinates for 52 boxes where the data will be displayed.
Explanation: These coordinates determine the exact position of each box in the interface. Each box is represented as $(x_1, y_1), (x_2, y_2)$, where (x_1, y_1) is the top-left corner and (x_2, y_2) is the bottom-right corner.

2.2.3 Load Data from CSV File

Task: Read data from the `d00_te.dat` file and store it as a matrix.
Explanation: The function `read_csv_to_matrix()` reads the file line by line and stores each row as a list inside the matrix. This matrix is later used to display data dynamically inside the boxes.

2.2.4 Define Box Groups & Assign Colors

Task: Categorize boxes into groups and assign unique colors.

Explanation: Each group has a specific color, making it visually distinguishable. `BOX_GROUPS` defines groups, and `GROUP_COLORS` assigns colors to each group.

2.2.5 Image Generation

Task: Generate a Pygame surface with the background image and dynamically update it with data inside boxes.

Explanation:

- Loads a background image (`Dc2image.jpg`).
- Scales it to fit the window size (1920×1080).
- Draws boxes using predefined coordinates and fills them with the corresponding group color.
- Retrieves data from `matrix[row_index]` and displays the corresponding value inside each box.

2.2.6 Display Text Inside Boxes

Task: Render and position the data correctly inside each box.

Explanation:

- The function `font.render(row_data, True, (255, 0, 0))` creates a text object.
- $\texttt{center\_x} = \frac{\texttt{box}[0][0]+\texttt{box}[1][0]}{2}$ calculates the horizontal center of the box.
- $\texttt{center\_y} = \frac{\texttt{box}[0][1]+\texttt{box}[1][1]}{2}$ calculates the vertical center.
- Finally, `image.blit(box_text, (center_x, center_y))` places the text in the correct position.

2.2.7 User Controls (Enabled via Panel's Widgets & IPython Integration)

Start Simulation

Button: Triggers the `start_and_detect()` function.

Functionality:

- Fetches the next row of data.
- Updates the dynamically drawn boxes.
- Runs continuously in the background.

Stop Simulation

Button: Calls `stop_function()`.

Functionality:

- Immediately halts data updates.
- Preserves the current state so the user can review data.

Reset Simulation

Button: Calls `reset_function()`.

Functionality:

- Clears all displayed values.
- Resets the image to its initial state.
- Prepares the app for a fresh start.

Restart Simulation

Button: Calls `restart_function()`.

Functionality:

- Resets everything.
- Starts streaming data again.

2.2.8 Ensure Smooth Execution

Task: Use threading or event-based updates to avoid UI freezing.

Explanation:

- Running the update function in a separate thread prevents the main UI thread from freezing.
- This ensures that Pygame's display remains interactive.

2.2.9 Display the Final Image in Jupyter Notebook

Task: Convert the Pygame surface into a format that Panel can display.

Explanation:

- Use the `io.BytesIO()` object to store the image.
- Convert it to a PIL image using `Image.open(buffer)`.
- Display it in Panel using `pn.pane.PIL()`.

2.2.10 Implemented PCA

Dataset Selection

The dataset contains three different types of faults. However, for this implementation, only **Fault 1** is considered. The relevant portion of the dataset is extracted by selecting data corresponding to Fault 1.

Feature Selection and Splitting

The dataset is divided into input features and output labels. The input data consists of sensor readings, while the output label indicates the presence or absence of Fault 1. The dataset is further split into training and testing sets using an 80-20 split to ensure a fair evaluation.

Feature Normalization

To standardize the input features, normalization is applied using **StandardScaler**, which scales the data to have zero mean and unit variance. This step is crucial for ensuring that all features contribute equally to the PCA transformation.

Dimensionality Reduction using PCA

Principal Component Analysis (PCA) is employed to reduce the dimensionality of the dataset while preserving the most important variance in the data. The number of principal components is set to 20, capturing the majority of the variance and enabling efficient fault detection.

Fault Detection via Residual Analysis

To detect faults, the dataset is reconstructed using the principal components, and the reconstruction error is computed. The mean and standard deviation of this error are used to define **control limits**. Any sample with a reconstruction error beyond these limits is flagged as faulty.

2.2.11 Implemented Neural Networks (MLP)

Dataset Preparation

The dataset used for training the neural network is **TEP_Fault_dataset_modified.csv**. The dataset contains both normal and faulty conditions. For this implementation:

- **No Fault** data consists of rows **1 to 7201**.
- **Fault 1** data consists of rows **7202 to 14402**.

Feature Selection and Splitting

The dataset includes multiple fault types, but only **Fault 1** is considered. The columns corresponding to **Fault 1, Fault 2, and Fault 3** are dropped, and the target column for classification is set to **Fault 1**. The dataset is split into three parts:

- **Training Set:** 60% of the data
- **Validation Set:** 20% of the data
- **Testing Set:** 20% of the data

The test set is saved as a '.dat' file for evaluation purposes.

Feature Scaling

To ensure consistent feature representation, **StandardScaler** is applied to standardize the input features. The mean and variance of the training set are stored in `scaler.npy` and `scaler_var.npy` for later use.

Model Architecture

A **Multi-Layer Perceptron (MLP)** is used for fault classification. The network architecture consists of:

- **Input Layer:** Accepts **52 features**.
- **Hidden Layer 1: 128 neurons** with **ReLU** activation.
- **Dropout Layer:** 30% dropout to prevent overfitting.
- **Hidden Layer 2: 64 neurons** with **ReLU** activation.
- **Hidden Layer 3: 32 neurons** with **ReLU** activation.
- **Output Layer: 1 neuron** with **sigmoid** activation for binary classification.

Training Process

The model is trained using the **Adam optimizer** with a **binary cross-entropy loss function**. Training is performed for **30 epochs** with a batch size of **64**. An **Early Stopping** mechanism is used, monitoring validation loss and stopping training if no improvement is observed for 5 consecutive epochs.

Model Evaluation and Saving

After training, the model is evaluated on the test set. The final test accuracy is displayed, and the trained model is saved as `fault_detection_nn2.keras`. The scaler parameters are also saved for consistent preprocessing during inference.

2.2.12 Implemented Gramian Angular Field (GAF) Model

To further improve fault detection accuracy and leverage time-series patterns, we implemented the **Gramian Angular Field (GAF)** representation of sensor data. This approach converts time-series data into images, allowing a **Convolutional Neural Network (CNN)** to process them for classification.

Data Preprocessing

- The dataset was split into **"No Fault"** and **"Fault 1"** categories, maintaining a **60-20-20** train-validation-test split.
- Features were extracted by removing fault labels (`Fault_1`, `Fault_2`, `Fault_3`).
- Standardization was applied using **StandardScaler** to normalize the features.

Gramian Angular Field (GAF) Transformation

- The **pyts** library was used to transform the time-series data into **32×32 GAF images**.
- Each feature vector was reshaped and converted into an image, preserving temporal dependencies.
- The transformed images were then used as input for the CNN model.

CNN Model Architecture

A CNN was designed to process the GAF images, consisting of:

- **Conv2D** layers with ReLU activation for feature extraction.
- **MaxPooling2D** layers to reduce spatial dimensions.
- **Dropout** layers to prevent overfitting.
- **Fully connected** layers with a `sigmoid` activation function for binary classification.

Training and Evaluation

- The model was trained using **Adam optimizer** and **binary cross-entropy** loss.
- **Early stopping** was applied to prevent overfitting.

2.2.13 Fault Detection System Integration

To enhance user interaction and streamline fault detection, we implemented a dynamic selection and visualization interface. The following components were added:

Dropdown for Fault Detection Technology

A dropdown widget was introduced to allow users to select the fault detection technology. Since Neural Network(MLP) performed the best among the all, we decided to move ahead with this model only. The available options include:

- **None**: No fault detection is performed.
- **Neural Network**: A pre-trained neural network model is used for fault classification.

The selected technology determines the processing pipeline applied to the input data.

Result Panel

To provide a comprehensive visualization of the fault detection process, the following component was integrated:

- **Result Panel**: Updates in real-time to show the fault detection summary and classification results.

2.2.14 Fault Detection System Architecture

The fault detection system follows a structured approach to load data, apply preprocessing, and infer faults using a neural network model.

Model Loading and and Handling NaN values

- The trained neural network model is loaded from the `fault_detection_nn2.keras` file.
- The **StandardScaler** is applied to normalize input data, ensuring consistency with training data distribution.
- If missing values (`NaN`) appear post-scaling, they are replaced with the column mean to maintain prediction accuracy.

Continuous Fault Detection from Stream and Alarm Triggering

The fault detection system processes input data in a continuous manner:

1. The system reads data from a CSV file in a row-wise manner, simulating real-time inference.
2. Each row is preprocessed using standardization and missing value handling.
3. The neural network model makes predictions on each processed row.
4. A binary classification approach determines whether the current instance represents a fault or no fault.
5. If more than 10 fault instances are detected, an **alarm is triggered** to notify the user.

Fault Detection Summary

To provide an aggregated overview of detected faults, a summary is generated at the end of the detection cycle. The summary includes:

- The total number of inferences performed.
- The count of ”Fault” and ”No Fault” instances.
- A sequence-based fault summary indicating the range of data points associated with each fault status.

2.2.15 Results

The final fault summary is displayed in the result panel, offering a structured insight into the detected fault patterns.

This integration ensures a seamless user experience while enabling efficient fault classification and real-time monitoring.

2.2.16 Refactoring for Dynamic, Modular, and User-Friendly Codebase

To enhance flexibility and usability, the fault detection system has been redesigned to be fully dynamic. Users can now provide inputs interactively instead of modifying the code manually, making the application adaptable to different datasets and configurations.

Dynamic File Input Handling

Users can specify input files for both fault detection and image display at runtime. The system validates files for existence, format (.csv or .dat), and structure before proceeding. This ensures that incorrect or missing files do not disrupt the execution.

Flexible Box Coordinates Colors

Users can either choose default settings or manually input coordinates for image visualization. The system ensures correct input formatting and assigns appropriate colors dynamically, improving visualization and adaptability to different fault detection scenarios.

Custom Image Selection

Instead of relying on hardcoded images, users can now specify an image file dynamically at runtime. This enhancement allows flexibility in selecting different background images for visualization, ensuring compatibility with various datasets.

Dynamic Neural Network Model Configuration

The system now allows users to specify dataset filenames, column selections, and indices interactively. Validation functions ensure correct formats (e.g., integers for indices, single-column names). This adaptability ensures the model can be applied to different datasets without requiring code modifications.

Chapter 3

Technologies Used

3.1 Programming Languages

This project leverages multiple programming languages to ensure efficient implementation, seamless integration, and robust documentation. The primary languages used are:

3.1.1 Python

Python serves as the core language for this project due to its versatility, extensive libraries, and ease of integration.

3.1.2 LaTeX

LaTeX is used for structured documentation and technical report writing ensuring professional formatting and seamless integration of equations and figures.

3.1.3 Shell Scripting (Bash)

Bash scripting can be employed for automation and running scripts in deployment.

3.2 Frameworks

In this project, we utilized a structured framework to handle machine learning tasks efficiently.

3.2.1 TensorFlow

TensorFlow is an open-source machine learning framework developed by Google. It provides a flexible ecosystem for developing, training, and deploying machine learning models. Key reasons for using TensorFlow in this project:

- Supports deep learning architectures, including neural networks used for fault detection.

- Offers GPU acceleration for faster model training and inference.
- Provides Keras as a high-level API for easy model building and experimentation.
- Includes built-in tools for data preprocessing, model evaluation, and deployment.

In the development of this project, several tools were utilized to enhance coding efficiency, debugging, and execution management. These tools streamlined the development workflow and ensured smooth interaction with the codebase.

3.3 Tools

3.3.1 JupyterLab

Category: Interactive Development Environment (IDE)

Purpose:

- Used as the primary development environment for writing, running, and debugging Python code.
- Supports interactive coding, markdown documentation, and visualization.
- Allows execution of code cells independently, which is useful for testing individual components.

3.3.2 Pip

Category: Package Management Tool

Purpose:

- Used to install, upgrade, and manage Python libraries and dependencies.
- Ensures that required libraries like NumPy, Pandas etc are available in the environment.

3.3.3 Python Debugger (pdb)

Category: Debugging Tool

Purpose:

- Helps in step-by-step execution of the code to identify and fix errors.
- Allows setting breakpoints, inspecting variables, and controlling execution flow.

3.4 Libraries

The project incorporates multiple libraries to handle data processing, visualization, interactivity, and real-time fault detection. Each library serves a specific function within the implementation.

3.4.1 Panel

Purpose: The ‘panel‘ library enables the creation of an interactive GUI, allowing users to control the simulation through buttons, dropdowns, and real-time updates. It supports widgets like Button, Select, and Markdown, enhancing user interaction with the app. **Usage:**

- Provides the dropdown menu to select fault detection methods, user control buttons(start, stop, reset & restart) and result panel to visualize fault summary results.
- The layout made where every buttons are grouped in a coloumn and that arranged with the image in a row is provided by Panel `pn.row` and `pn.coloumn`.
- Integrates seamlessly with Pygame and create a single image pane using `pn.pane.PNG` to display images.

3.4.2 Pygame

Purpose: Handles real-time graphical rendering of images. **Usage:**

- Loads and scales the base image to fit the UI window (1920x1080 resolution) using `pn.image.load` and `pygame.transform.scale()` respectively.
- Draws rectangles around predefined box regions using `pygame.draw.rect`, categorizing them with different colors based on BOX˙GROUPS.
- Converts the Pygame surface into an array for further image processing using `pygame.surfarray.pixels3d(image)`
- The image processing function `generate_pygame_image()` coded is responsible for updating this visualization.

3.4.3 os

Purpose: Provides system-level operations, such as file path handling. **Usage:**

- Checks whether the dataset file (e.g., `TEP_Fault_dataset.csv`) exists before loading.
- Retrieves the correct file paths dynamically to avoid hardcoding absolute paths.

3.4.4 sys

Purpose: Manages system-related functions, including script termination and error handling. **Usage:**

- Allows controlled script exits if an error occurs while loading data.

3.4.5 csv

Purpose: Enables reading and writing CSV files. **Usage:**

- Reads real-time streaming data from `TEP_Fault_dataset.csv`.
- The function `read_csv_to_matrix()` converts CSV data into a matrix format.

3.4.6 time

Purpose: Implements time-based functions for delays and execution timing. **Usage:**

- Adds delays in UI updates to ensure smooth visualization. For example in this project, I have added time delay of *1 sec*, means the gap between the data of each row displaying inside the boxes will be 1 sec i.e after every 1 sec, new row data points will be displayed inside the boxes

3.4.7 numpy

Purpose: Performs mathematical and array-based operations, particularly for scaling and preprocessing. **Usage:**

- Used for normalizing input data before feeding it into the model.
- Loads and applies stored mean and variance values for preprocessing.
- `np.mean()` is used to fill NaN values after scaling to avoid misclassifications.

3.4.8 pandas

Purpose: Processes structured tabular data, including CSV file handling and manipulation. **Usage:**

- Loads `TEP_Fault_dataset.csv` into a structured DataFrame.
- Extracts the relevant 52 feature columns from the dataset.
- Converts raw CSV data into a matrix format suitable for model inference.

3.4.9 PIL (Pillow)

Purpose: Handles image processing and manipulation. **Usage:**

The ‘Pillow‘ (PIL) library processes the images generated by ‘pygame‘, performing transformations like rotation and flipping using `pil_image.rotate()` and `ImageOps.flip()` respectively to prepare visuals for display within the app interface.

3.4.10 IPython.display

Purpose: Enables interactive display of elements inside JupyterLab. **Usage:**

- Ensures smooth rendering of images and UI elements.
- Displays outputs interactively without requiring external windows.

3.4.11 threading

Purpose: Enables multi-threaded execution of functions to keep UI responsive during real-time fault detection. **Usage:**

- Runs the fault detection process in a separate thread, ensuring the UI does not freeze.
- Allows continuous data streaming while letting the user interact with controls.

3.4.12 io

Purpose: Manages input/output operations, particularly for handling streamed data. **Usage:**

- Used for reading CSV data dynamically as it is streamed in.

3.4.13 winsound

Purpose: Generates sound notifications for fault detection alerts. **Usage:**

- Plays an alert sound whenever fault count becomes greater than 10.
- The sound is triggered inside the `detect_fault()` function to immediately notify the user of issues.

3.4.14 sklearn (scikit-learn)

Purpose: Provides machine learning utilities, primarily for preprocessing and model handling. **Usage:**

- Uses `StandardScaler` for feature scaling.
- Loads pre-trained models for fault classification.
- Ensures that the same mean and variance from training are applied during inference.

3.4.15 ipywidgets

Purpose: Enhances interactivity inside JupyterLab with input widgets. **Usage:**

- Provides UI components like dropdown menus and buttons for method selection and control the simulation respectively.

3.4.16 ast

Purpose: Parses and validates user input values when working with interactive selections. **Usage:**

- Used for safely converting user input strings into structured Python objects.
- Ensures correct data types when processing dynamically entered values.

3.5 Fault Detection Technolgies

3.5.1 PCA

PCA is a widely used statistical technique for dimensionality reduction and feature extraction. It transforms high-dimensional datasets into a smaller set of uncorrelated variables called principal components, which capture the most variance in the data. In the context of fault detection and process monitoring, PCA has been effective in reducing the computational complexity of large-scale datasets while retaining critical information for fault classification. However, its reliance on linear relationships limits its ability to capture complex, nonlinear dynamics in industrial systems. Additionally, PCA does not inherently address temporal information, making it less effective for time-series data analysis.

3.5.2 Neural Network (MLP) Model

A Multi-Layer Perceptron (MLP) is a type of artificial neural network (ANN) that consists of multiple layers of interconnected neurons. It is a feedforward neural network where data moves from the input layer to the output layer without looping back.

Structure of MLP

- **Input Layer:** Receives input features (e.g., sensor readings in this project).
- **Hidden Layers:** Intermediate layers with neurons that apply activation functions (like ReLU) to learn complex patterns.
- **Output Layer:** Produces the final classification or regression result (e.g., fault detected or not).

How It Works

- Each neuron processes input data using weights, biases, and an activation function.
- The network is trained using backpropagation, which adjusts weights based on the error (loss function) to improve accuracy.
- MLP is commonly used for classification and regression tasks, making it suitable for fault detection in this project.

3.5.3 GAF

Gramian Angular Field (GAF) is a method that converts time-series data into images. Instead of passing raw numerical values to a model, GAF encodes fault data as visual representations, allowing a CNN (Convolutional Neural Network) to detect patterns.

Chapter 4

Impact

4.1 Scalibility

- **Deployment on Mobile and Cloud Platforms:** The system can be deployed across multiple industrial setups, allowing real-time analysis of high-dimensional sensor data. Mobile deployment would allow on-site fault detection in remote locations, while cloud deployment could enable centralized monitoring and analysis for distributed systems. This expansion would provide end-users with flexible, powerful tools for fault management.

4.2 Future Scope

- **Multi label Classification**: The model can be expanded from binary fault classification to multi-label fault classification to further increase its utility.

4.3 SOLID Principles

- The CodeBase can be re-architectured to meet the SOLID(Single Responsibilty Principle, Open Closed Principle, Liskov's Substitution Principle, Interface Segregation Principle, & Dependency Inversion Principle) Principles for maintainability.

4.4 Results

We have achieved 99.97% accurate results overall n identifying faulty vs. non-faulty conditions, significantly reducing false alarms.

4.4.1 Neural Network-Based Approach

The neural network model achieved high accuracy across training, validation, and testing datasets:

- **Training Accuracy:** Approached 1.0, demonstrating effective learning of fault patterns from the training data.
- **Validation Accuracy:** Reached 1.0, indicating strong generalization on unseen validation data during training.
- **Testing Accuracy:** Achieved 0.9997, confirming excellent model reliability when deployed on new data. The results reflect minimal overfitting due to early stopping and dropout layers in the model.

4.4.2 Gramian Angular Field (GAF)-Based Approach

The GAF-based model showed strong improvements over training epochs:

- **Training Accuracy:** Increased from an initial 60% to 95% by the 30th epoch, showing that the model effectively adapted to identifying fault patterns as training progressed.
- **Validation Accuracy:** Improved from 60% to 80%, indicating the model's improved ability to generalize to validation data but with some signs of overfitting.
- **Testing Accuracy:** Achieved 80.78%, which is reasonable but lower than the neural network approach, suggesting potential for further optimization.

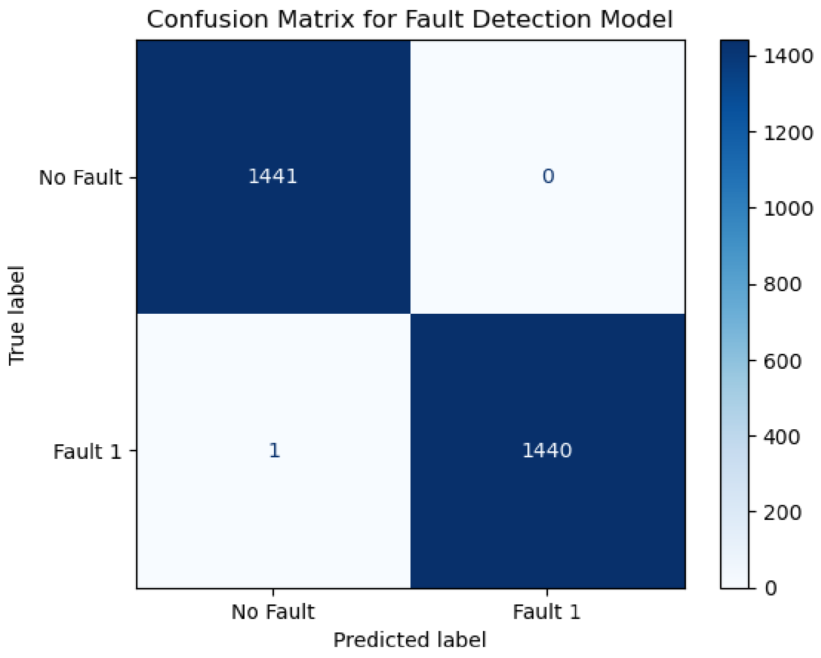
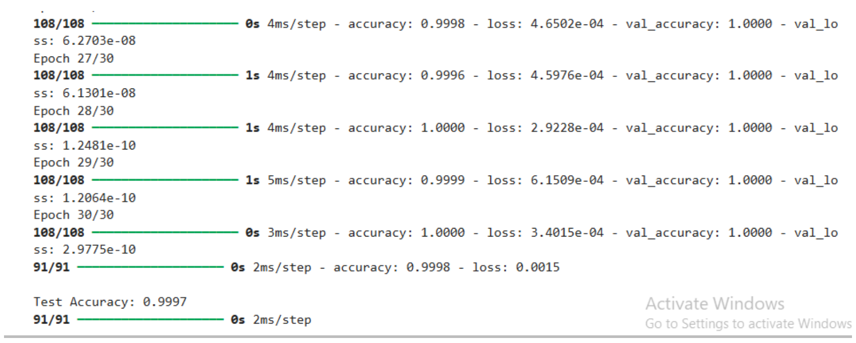


Figure 4.4.1: Results of the Neural Network-Based Approach.



```
108/108 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9998 - loss: 4.6502e-04 - val_accuracy: 1.0000 - val_lo
ss: 6.2703e-08
Epoch 27/30
108/108 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9996 - loss: 4.5976e-04 - val_accuracy: 1.0000 - val_lo
ss: 6.1301e-08
Epoch 28/30
108/108 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 1.0000 - loss: 2.9228e-04 - val_accuracy: 1.0000 - val_lo
ss: 1.2481e-10
Epoch 29/30
108/108 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9999 - loss: 6.1509e-04 - val_accuracy: 1.0000 - val_lo
ss: 1.2064e-10
Epoch 30/30
108/108 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 1.0000 - loss: 3.4015e-04 - val_accuracy: 1.0000 - val_lo
ss: 2.9775e-10
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.9998 - loss: 0.0015

Test Accuracy: 0.9997
91/91 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
```

Figure 4.4.2: Results of the Neural Network-Based Approach.

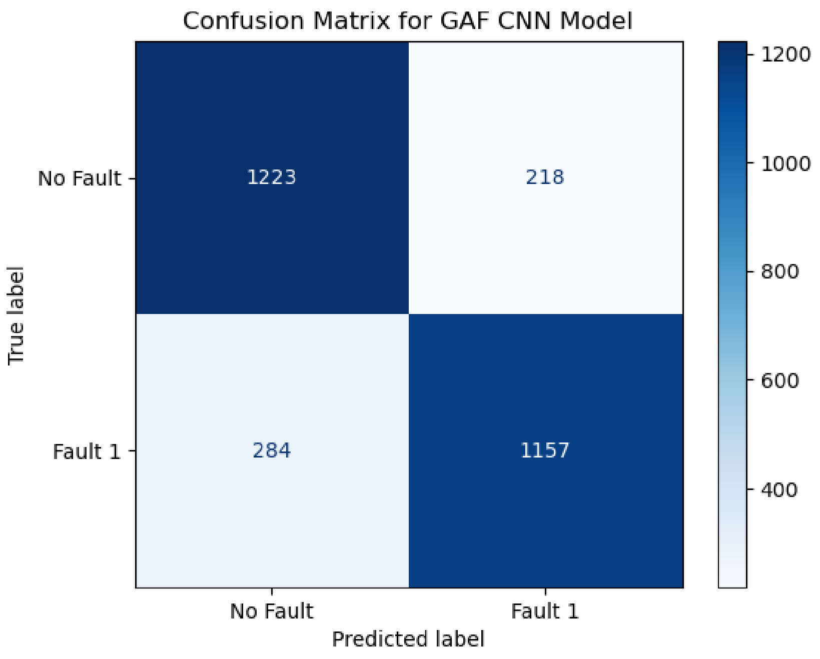
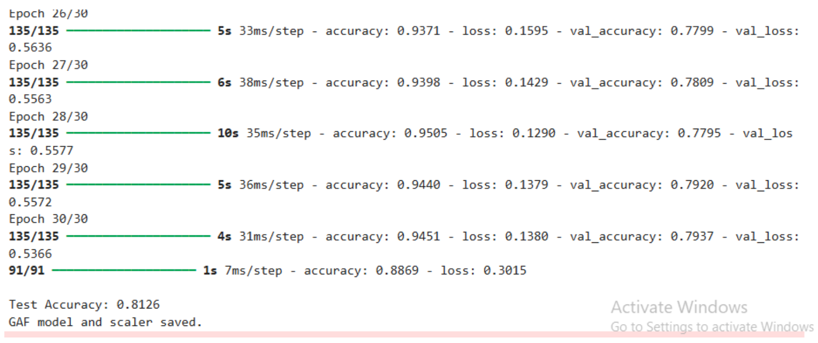


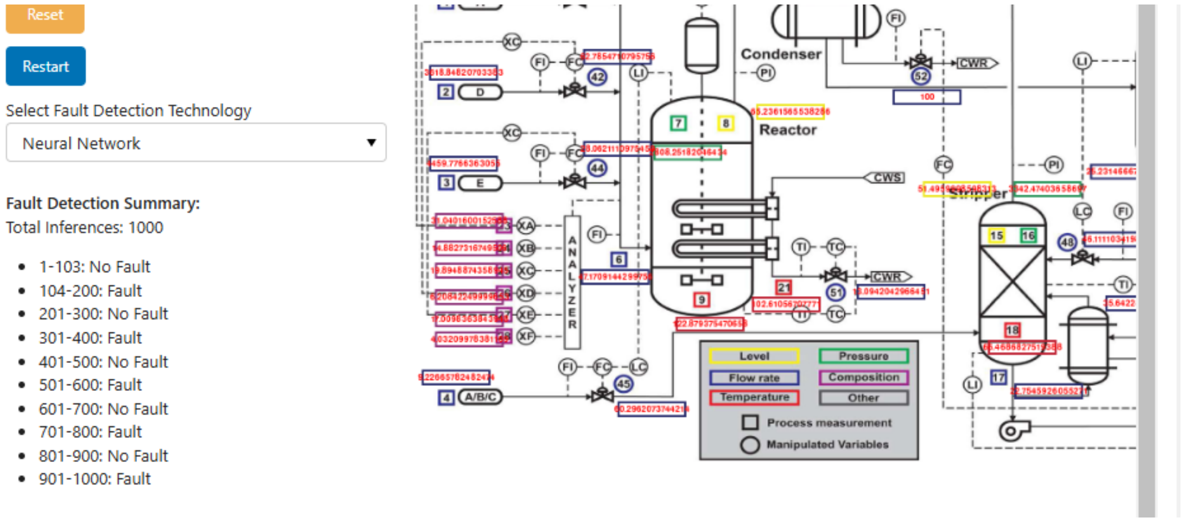
Figure 4.4.3: Results of the GAF-Based Approach.



```
Epoch 26/30
135/135 ━━━━━━━━━━━━━━━━━━━━ 5s 33ms/step - accuracy: 0.9371 - loss: 0.1595 - val_accuracy: 0.7799 - val_loss:
0.5636
Epoch 27/30
135/135 ━━━━━━━━━━━━━━━━━━━━ 6s 38ms/step - accuracy: 0.9398 - loss: 0.1429 - val_accuracy: 0.7809 - val_loss:
0.5563
Epoch 28/30
135/135 ━━━━━━━━━━━━━━━━━━━━ 10s 35ms/step - accuracy: 0.9505 - loss: 0.1290 - val_accuracy: 0.7795 - val_los
s: 0.5577
Epoch 29/30
135/135 ━━━━━━━━━━━━━━━━━━━━ 5s 36ms/step - accuracy: 0.9440 - loss: 0.1379 - val_accuracy: 0.7920 - val_loss:
0.5572
Epoch 30/30
135/135 ━━━━━━━━━━━━━━━━━━━━ 4s 31ms/step - accuracy: 0.9451 - loss: 0.1380 - val_accuracy: 0.7937 - val_loss:
0.5366
91/91 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - accuracy: 0.8869 - loss: 0.3015

Test Accuracy: 0.8126
GAF model and scaler saved.
```

Figure 4.4.4: Results of the GAF-Based Approach.



Select Fault Detection Technology

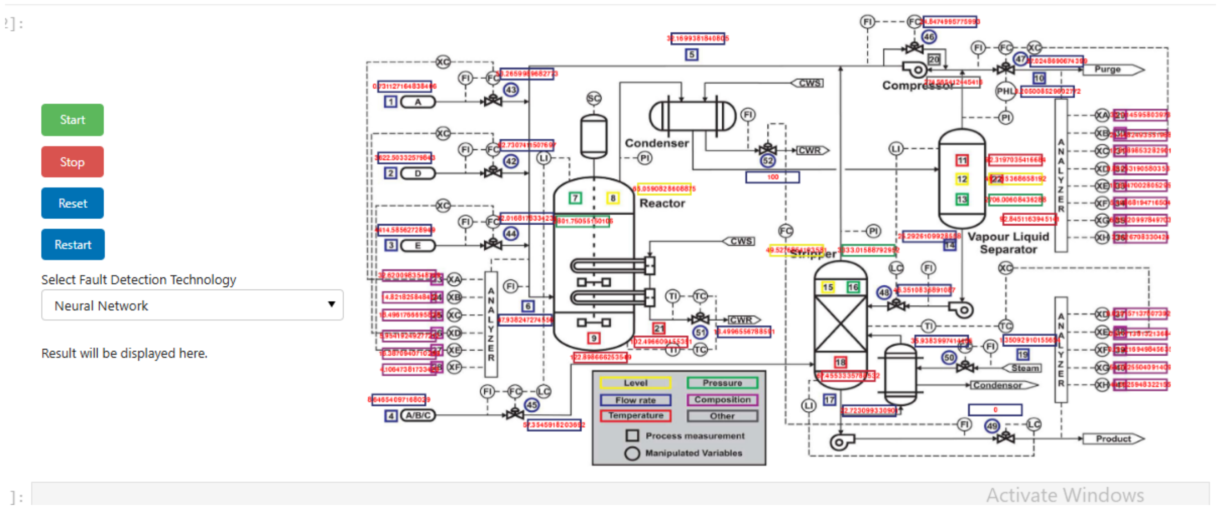
Neural Network

Fault Detection Summary:

Total Inferences: 1000

- 1-103: No Fault
- 104-200: Fault
- 201-300: No Fault
- 301-400: Fault
- 401-500: No Fault
- 501-600: Fault
- 601-700: No Fault
- 701-800: Fault
- 801-900: No Fault
- 901-1000: Fault

Figure 4.4.5: Result Panel Fault Summary



Select Fault Detection Technology

Neural Network

Result will be displayed here.

Figure 4.4.6: Interface of the App

Chapter 5

My Role & Challenges

As this was an individual project, everything required for this project was done by me under the supervision of my professor.

5.1 Challenges

The implementation includes several core functions for data streaming, fault detection, and real-time visualization:

5.1.1 Image Generation

- Initially, the project required dynamically drawing and updating boxes on an image while continuously displaying changing data. The challenge was that Panel, which is primarily designed for interactive dashboards, did not provide an easy way to modify images in real-time.

  This was the main challenge as told by my professor also but I wasted many days trying this with Panel and unfortunately unable.

  At that time, my friend told me that he has made Tic Tac Toe game using Pygame library and instantly it striked me that tic tac toe game also requires dynamic image updating using tick and cross operations. This led to the introduction of Pygame, which is well-suited for real-time graphics rendering. However, Pygame opened a separate window which was lacking in terms of user experience and tha was not ideal since the goal was to keep the visualization within the Panel-based interface.

  To solve this, Panel and Pygame were integrated, allowing Pygame to handle the dynamic rendering while ensuring the final visualization remained inside the original Panel dashboard. This integration combined the strengths of both libraries.

5.1.2 Asynchronous Processing(Multithreading/Concurrency)

- Actually, initially what I was doing was when the data was coming inside the boxes, I was extracting the same row data and immediately passing to the model for fault detection. However, this approach caused UI freezing, as the processing time of the model could sometimes exceed the time gap between two consecutive data updates.

  To solve this problem, I used asynchronous programming to handle the fault detection in the background. In this way, fault detection can proceed without being blocked by the time it takes to process the data.

5.1.3 Data Preparation

Mismatch in Column Count

- The raw dataset used for displaying values in the plant image contained $n \times 52$ data points (where 52 represents sensor readings and n represents different time instances).
- However, the labeled dataset provided for training had **56 columns**, including three output columns: `"Fault 1"`, `"Fault 2"`, and `"Fault 3"`.
- To ensure proper training, I had to **drop one less important column**, reducing the input features accordingly.

String-to-Float Conversion Issue

- The `csv.reader` function reads all data as **strings** by default.
- Since machine learning techniques like **PCA and Neural Networks** require **numeric (float) data**, I had to convert the values.
- Before conversion, I encountered **unusual spaces** in the dataset, preventing direct conversion to float.
- To resolve this, I **cleaned the dataset** before applying the conversion.

Handling Data Format for Pygame

- The function `generate_pygame_image` expected dataset values in **string format** because it used the `split()` method to extract and display values inside boxes.
- However, since the model required **float values** for computation, I had to **handle both scenarios separately**, ensuring numerical data for fault detection and string format for visualization.

Binary Classification Data Selection

- The dataset (`TEP_Fault_dataset.csv`) contained **28,805 rows**, with:
  - **Rows 1-7201** → No Fault
  - **Rows 7202-14403** → Fault 1
  - **Rows 14404-21604** → Fault 2
  - **Rows 21605-28805** → Fault 3
- Since the project focused on **binary classification (No Fault vs. Fault 1)**:
  - I extracted **14,402 rows** (excluding the first header row).
  - The data was split into:
    - **Training:** 60% from each No Fault and Fault 1 data
    - **Validation:** 20% from each No Fault and Fault 1 data
    - **Testing:** 20% from each No Fault and Fault 1 data

Handling NaN Issues After Scaling

- Model predictions were inconsistent due to **NaN values appearing after scaling**.
- Adjusted the **StandardScaler** implementation, ensuring NaN values were **replaced with the mean of each feature** before inference.
- This improved **stability and consistency** in predictions.