Optimizing Renewable Energy Integration in GCPBBB Systems using Machine Learning

Nisal Jinadasa

December 4, 2025

Abstract

This report details the development and implementation of a Machine Learning-based optimization system for Grid-Connected Photo Sensor Based Battery Balancing (GCPBBB) architectures. The system integrates three distinct models: an Energy Predictor for demand forecasting, a Balancing Classifier for battery management, and a Grid Stability Monitor for real-time fault detection. Utilizing a hybrid dataset of real-world electrical measurements and calibrated synthetic operational data, the system achieves high predictive accuracy (MAE $\approx$ 42W) and robust classification performance (>99% accuracy). A modern React-based dashboard serves as the central control interface, providing real-time visualization and decision support.

1 Introduction

The integration of renewable energy sources, particularly solar PV, into the grid presents challenges related to intermittency and stability. Grid-Connected Photo Sensor Based Battery Balancing (GCPBBB) systems aim to mitigate these issues. This project enhances GCPBBB architectures by applying Machine Learning (ML) to optimize energy distribution, ensure battery health, and monitor grid stability in real-time.

2 Methodology

2.1 Data Acquisition and Preparation

To train robust models, a hybrid data strategy was employed, combining real-world fault data with high-fidelity synthetic operational data.

2.1.1 Real-World Calibration Data

- **Source:** Mendeley GPVS-Faults Dataset.
- **Composition:** 16 CSV files containing high-frequency electrical measurements (Voltage, Current) for various operating states (Stable F0, Faults F1-F7).

- **Calibration Statistics:**
  - Peak Power (95th percentile): 215.71 W
  - Mean PV Voltage: 88.10 V

  These statistics were used to calibrate the synthetic data generator to ensure realistic physical constraints.

2.1.2 Synthetic Training Data

A comprehensive dataset representing one month of minute-level operation (43,200 samples) was generated to simulate the GCPBBB environment.

- **Solar Irradiance:** Modeled using diurnal sine waves with stochastic cloud cover noise.
- **Grid Consumption:** Simulated based on typical residential load profiles with morning and evening peaks.
- **Battery SoC:** Calculated dynamically based on the net energy flow (Generation vs. Consumption).
- **Target Labels:** A "Balancing Signal" was derived based on economic and physical logic:
  - **0 (Discharge):** Deficit power or high demand.
  - **1 (Hold):** Balanced system or idle state.
  - **2 (Charge):** Surplus power or low demand.

2.2 Model Development

Three specialized non-deep learning models were developed to address specific system needs.

2.2.1 1. Grid Consumption Predictor (Planner Module)

- **Algorithm:** Random Forest Regressor.
- **Objective:** Forecast grid demand for the next time step $(t+1)$ to enable proactive battery scheduling.
- **Features:** Lagged consumption, generated power, time-of-day (cyclic), temperature, irradiance.

2.2.2 2. Balancing Signal Classifier (Planner Module)

- **Algorithm:** XGBoost Classifier.
- **Objective:** Determine the optimal battery action (Charge, Discharge, Hold) based on current system state and predicted demand.
- **Features:** Battery SoC, Net Energy, Predicted Consumption, Irradiance.

2.2.3 3. Grid Stability Monitor (Guardian Module)

- **Algorithm:** Random Forest Classifier.
- **Objective:** Real-time detection and classification of grid faults.
- **Features:** Statistical aggregates (Mean, Std, Max, Min) of high-frequency voltage and current signals over a rolling window.
- **Classes:** F0 (Stable), F1-F7 (Specific Fault Types like Line-Line, Open Circuit).

3 System Architecture and Dashboard

3.1 The "Control Room" Dashboard

The user interface is built using **React**, designed to function as a real-time "Control Room" for grid operators. It communicates with a **FastAPI** backend to integrate the three models into a cohesive workflow.

3.1.1 Workflow Integration

1. **Data Ingestion:** The backend ingests a stream of sensor data (simulated or real).
2. **Preprocessing:** Data is cleaned, scaled, and feature-engineered (e.g., lag generation) in real-time by the Python backend.
3. **Inference Pipeline:**
    - The *Stability Monitor* first checks the electrical signature. If a fault is detected (e.g., F1), a "Protection Relay" signal is triggered, isolating the system.
    - If stable, the *Energy Predictor* forecasts the next-step load.
    - The *Balancing Classifier* uses this forecast and current SoC to decide the optimal battery action.
4. **Visualization:** The React frontend polls the backend state to update live charts for Solar vs. Load, Battery SoC, and displays the current System Status and Predicted Action.

4 Performance Results

4.1 Grid Consumption Predictor

- **Metric:** Mean Absolute Error (MAE)
- **Result: 42.11 W**
- **Analysis:** The model accurately tracks diurnal patterns. An error of 42W is negligible compared to typical household loads (200-2000W), making it highly effective for short-term planning.

4.2 Balancing Signal Classifier

- **Overall Accuracy: 99.77%**
- **Key Insight:** The model achieved 100% precision and recall for the critical "Charge" and "Discharge" classes, ensuring that the battery is never charged during a deficit or discharged during a surplus unnecessarily.

Table 1: Balancing Classifier Performance

Class	Precision	Recall	F1-Score	Support
0 (Discharge)	1.00	1.00	1.00	4473
1 (Hold)	0.71	0.43	0.54	23
2 (Charge)	1.00	1.00	1.00	4141

4.3 Grid Stability Monitor

- **Overall Accuracy: 99.77%**
- **Analysis:** The model successfully distinguishes between stable operation and 7 distinct fault types. This high accuracy is critical for the "Guardian" module's safety function.

Table 2: Grid Stability Monitor Performance

Fault Type	Precision	Recall	F1-Score	Support
F0 (Stable)	1.00	1.00	1.00	47
F1 (Fault)	1.00	0.98	0.99	58
F2 (Fault)	1.00	1.00	1.00	54
F3-F7 (Faults)	1.00	1.00	1.00	274

5 System Implementation & DevOps

5.1 Codebase Structure

The project follows a modular microservices architecture, organized as follows:

- **backend/**: Contains the FastAPI application logic (`main.py`) and Docker configuration.
- **frontend/**: Hosts the React application source code, UI components, and build scripts.
- **src/**: The core Machine Learning module containing:
  - `train_model.py`: Orchestrates the training pipeline.
  - `model.py`: Defines the Random Forest and XGBoost architectures.

  - `data_loader.py` & `preprocessing.py`: Handles data ingestion and feature engineering.

- **models/**: Stores the serialized trained models (`.pkl` files).
- **.github/workflows/**: CI/CD configuration files.
- **deploy/**: Deployment configurations including `docker-compose.yml` and Nginx settings.

5.2 Backend Training & Communication

5.2.1 Training Pipeline

The backend training process is automated via `src/train_model.py`. It performs the following steps:

1. **Data Loading:** Ingests real-world fault data and generates synthetic operational data.
2. **Preprocessing:** Applies scaling and feature extraction (e.g., lag features).
3. **Model Training:** Sequentially trains the Energy Predictor, Balancing Classifier, and Stability Monitor.
4. **Serialization:** Saves the trained models and scalers using `joblib` for efficient inference.

5.2.2 Frontend-Backend Communication

The React frontend communicates with the FastAPI backend via a RESTful API:

- **Polling Mechanism:** The frontend polls the `/state` endpoint to retrieve the latest simulation status (SoC, Load, Generation).
- **Simulation Control:** User actions (e.g., Start/Stop, Reset) are sent via POST requests to `/step` and `/reset`.
- **Data Flow:** The backend processes these requests, runs the ML inference using the loaded models, and returns the updated system state as a JSON response.

5.3 Deployment & CI/CD

5.3.1 Deployment Architecture

The system is deployed on a **Google Cloud Platform (GCP) Compute Engine** instance using a containerized approach:

- **Docker Compose:** Orchestrates the `backend` and `frontend` containers, ensuring environment consistency.
- **Nginx Reverse Proxy:** Acts as the gateway, routing traffic to the appropriate container (Port 80 $\rightarrow$ React Frontend / FastAPI Backend) and handling SSL termination via Cloudflare.

5.3.2 CI/CD Pipeline

A robust CI/CD pipeline is implemented using **GitHub Actions** (`.github/workflows/main.yml`) to ensure code quality and reliability:

1. **Backend Testing:** Triggers on every push to `main`. Sets up Python environment, installs dependencies, and runs unit tests using `pytest`.
2. **Build Verification:** Builds both Backend and Frontend Docker images to verify that the application is deployable.
3. **Docker Compose Check:** Validates the composition configuration to prevent deployment failures.

6 Conclusion

The developed system demonstrates the efficacy of Machine Learning in optimizing GCPBBB systems. By combining accurate demand forecasting with intelligent battery management and robust fault detection, the solution ensures efficient renewable energy utilization and grid stability. The high performance metrics across all three models validate the hybrid data approach and the selected model architectures.