

# Saved Parameters and Contribution

---

- This section contains the saved model parameters and the CSV comparison results: [https://drive.google.com/drive/folders/199xvOAdl1gVRwKMzwrmmLvRwp9Wzye3X?usp=drive\\_link](https://drive.google.com/drive/folders/199xvOAdl1gVRwKMzwrmmLvRwp9Wzye3X?usp=drive_link)

- ## Project Scope

- This project delivers an end-to-end electricity price prediction and trading workflow for the electricity market. The system covers data ingestion and cleaning, feature engineering, sequence construction, deep learning model training and comparison, rule-based strategy backtesting, and an interactive Streamlit dashboard for experimentation and visualization. My contributions span the full pipeline from backend logic to user-facing analytics.

- ## 1. End-to-End Pipeline Design and Modularization

- I designed the overall pipeline as five decoupled modules with standardized inputs/outputs so each stage can be tested independently and extended without modifying the entire codebase:
- Data processing ( `data_pipeline_v2.py` )
- Model implementations (LSTM)
- Unified evaluation ( `metrics.py` )
- Interactive dashboard and strategies ( `app.py` , `strategies.py` )
- This modular structure improves maintainability, enables controlled experiments, and supports reproducibility across team members.

- ## 2. Data Pipeline V2: Data Quality, Label Integrity, and Leakage-Safe Preprocessing

- **Problem diagnosis**
  - The raw dataset contained substantial missing values. Naive imputation (forward/backward fill) can create artificial constant-price segments and produce misleading return-based labels.
  - Invalid or abnormal prices (e.g., non-positive values) and near-zero price changes add noise to both modeling and strategy evaluation.
  - For the classification task, class imbalance can inflate accuracy and distort comparisons between models.
- **Implemented solution**
  - Built a revised “Data Pipeline V2” focused on label integrity and reproducible splits:
  - Removed rows with missing values in critical columns used for model inputs and label construction.
  - Filtered invalid price records (e.g., price  $\leq 0$ ).
  - Removed zero or near-zero returns using an epsilon threshold to avoid constant-price artifacts.

- Created fixed-length sequences using a 14-day sliding window for time-series modeling.
- Performed standardization using `StandardScaler` fit only on the training split, then applied to validation/test splits to prevent information leakage.
- Balanced the binary classification dataset via undersampling to achieve an approximately 50/50 class distribution.

- **Outcome**

- Produced a clean, leakage-safe, and balanced dataset suitable for fair classification benchmarking.
- Reduced the risk of inflated metrics caused by label artifacts from missing-value handling.

- **3. Deep Learning Modeling: LSTM Variant Implementations and Controlled Comparisons**

- I implemented and compared four LSTM-based model variants under the same data pipeline and training protocol to ensure fair evaluation:

- LSTM baseline (stacked LSTM)
- Bidirectional LSTM (BiLSTM)
- LSTM with multi-head self-attention
- BiLSTM with multi-head self-attention

- **Key technical elements**

- Integrated multi-head self-attention using `nn.MultiheadAttention` with residual connections and layer normalization to stabilize training.
- Applied regularization (dropout, batch normalization where appropriate) and early stopping to reduce overfitting.
- Created a configuration-driven naming/logging mechanism to make each run auditable and traceable to its architecture settings.

- **4. Unified Training and Experiment Automation**

- To avoid inconsistent training procedures across models, I implemented a unified training interface and batch experiment runner:
  - Standardized training loop with consistent optimizer, loss function, early stopping logic, checkpointing, and metric reporting.
  - Batch script to run all LSTM variants sequentially and export aggregated results (CSV) for direct comparison.
  - Ensured consistent evaluation metrics across models (accuracy, precision, recall, F1), with clear separation of train/validation/test usage.
- This work enabled reproducible benchmarking and reduced manual experimentation errors.

- **5. Rule-Based Strategies and Backtesting Module**

## ( **strategies.py** )

- In addition to ML models, I implemented rule-based strategies as a pure backend module with no Streamlit dependencies. The goal was to provide comparable baselines and a consistent evaluation interface.
- Strategy implementations
  - Percentile Channel Breakout:
    - Computes rolling percentile channels (e.g., 20th/80th percentile) over a fixed window and generates long/short signals based on channel breaks.
    - Transforms signals into persistent positions using forward-filled position logic.
- Break of Structure (BOS):
  - Detects local trend regimes and key recent highs/lows over a lookback window.
  - Generates trading actions when price breaks structural levels, supporting position transitions among long/short/neutral states.
- Unified return computation
  - Implemented a standardized function to compute strategy returns from positions:
    - Aligns position at time  $t$  with realized return from  $t$  to  $t+1$
    - Supports long/short trading (position in  $\{-1, 0, +1\}$ )
  - This design allows rule-based strategies and ML outputs to be evaluated using the same metric functions (e.g., ROI, Sharpe, win rate, drawdown), enabling consistent comparisons.

## • 6. Streamlit Interactive Dashboard (app.py): UI-Backend Separation and Experiment Workflow

- I built an interactive Streamlit dashboard that exposes both ML training and strategy backtesting through a unified user workflow, while maintaining a strict separation between UI and backend logic.
- **UI-Backend separation**
  - app.py implements the presentation layer: parameter inputs, chart rendering, and results reporting.
  - Backend logic (data pipeline, strategies, training utilities) remains independent and testable without Streamlit.
- **Core dashboard functionality**
  - Cached data loading to avoid repeated I/O and to improve responsiveness during user interactions.
  - Parameterized strategy backtesting:
    - Users can adjust window sizes, percentile thresholds, date ranges, and initial capital.
    - Dashboard displays price series with annotated trade signals and cumulative return curves.

- ML training interface:
  - Users select task type and model type, run training, and inspect training curves and evaluation plots.
  - Outputs include diagnostic plots such as confusion matrix, ROC/PR curves (classification), and prediction-vs-actual plots (regression), depending on the selected task.

- **Engineering value**

- Provides a reproducible experiment environment where team members can run comparable evaluations from a single UI entry point.
  - Improves interpretability by visualizing signals, training dynamics, and error patterns.

- **7. Key Findings and Practical Recommendations**

- Attention mechanisms provided consistent improvements over the baseline LSTM by learning time-step importance rather than relying solely on the final hidden state.
  - BiLSTM variants improved balance-sensitive metrics (F1) but increased computational cost, highlighting a trade-off between model capacity and efficiency.

-