Summary

The increasing adoption of electric vehicles, driven by international sustainability goals, has led to a rapid expansion of electric vehicle charging stations. While this transition supports carbon neutrality, it potentially can introduce substantial challenges in the future to the existing power infrastructure, particularly the risk of transformer overload, degradation, and voltage instability due to rising and volatile electricity demand.
To combat these challenges, the study proposed an ensemble model that aggregates unique model predictions, possibly enhancing the reliability of EVCS electricity consumption forecasting and aid in effective overload capturing. The time series are aggregated at an hourly resolution with a 24-hour forecast horizon. The dataset is split into training, validation, and test sets using a 60/20/20 ratio.
To support the analysis, three datasets are used:

- Colorado Dataset: Real-world data from Boulder, Colorado with data from 2018 to 2023, covering public level 2 EVCS usage, aggregated to an hourly level. This dataset was filtered to remove COVID-impacted periods and is used for both forecasting and overload classification.
- SDU Synthetic Dataset: A simulated dataset based on Danish residential home charging from 2022 to 2032, modeled via a multi-agent system to reflect long-term grid load development, including detailed overload durations.
- Colorado Baseload Dataset: A separate dataset from the U.S. Energy Information Administration, reporting hourly electricity demand in the Public Service Company of Colorado balancing area. While it does not provide charger-level data, it is used exclusively to approximate the background baseload in Boulder and evaluate overload risks. This dataset is critical for defining overload thresholds during the classification task.

The ensemble integrates multiple diverse base learners, including Random Forest, Gradient Boosting, AdaBoost, LSTM, GRU, xPatch, and PatchMixer, trained on bootstrapped datasets. To ensure diversity and avoid overfitting, each type of base learner is included in the ensemble only once. The hypothesis is that aggregating predictions from these models will enhance the robustness of EVCS electricity forecasts and improve overload detection accuracy. This, however has the prerequire that models components must each generate good results for the ensemble to work. The experimental design is divided into two primary experiments:

1. Electricity Consumption Forecasting: This experiment evaluates how well different models can predict hourly energy consumption for EVCS. The primary performance metric used is Mean Absolute Error (MAE) or Huber Loss, which reflects prediction accuracy.

2. Overload Detection Classification: This experiment assesses each model's ability to detect potential transformer overloads based on predicted energy consumption and background baseload. Both recall and MAE are used as evaluation metrics, where recall is emphasized to ensure true overload events are captured, while MAE is also considered to avoid extreme over-predictions.

Deep learning models, particularly PatchMixer and xPatch, consistently outperformed both ensemble and standalone models across all experiments. While they achieved the highest overall scores, a common limitation was their inability to fully capture extreme upper consumption values, revealing challenges in modeling peak EVCS demand. Despite this, they demonstrated strong overload detection on the Colorado dataset, with recall scores of 0.88 (PatchMixer) and 0.71 (xPatch). Performance dropped on the SDU dataset, with recall scores of 0.54 and 0.64, respectively. Ensemble models showed mixed and less reliable results, largely due to unstable configurations and inconsistent performance among base learners, which hindered their generalization across tasks and datasets.

Capturing Transformer Overloads through Ensemble-Based Forecasting of EV Charging Station Consumption

An Evaluation of Model Diversity and Its Impact on Detection Performance

Casper Søgård and Sebastian Truong

Software, cs-25-dt-10-02, 2025-05

Master Thesis

AALBORG UNIVERSITY

STUDENT REPORT

Department of Computer Science
Software
Selma Lagerløfs Vej 300
9220 Aalborg Ø
www.aau.dk

Title:
Capturing Transformer Overloads through Ensemble-Based Forecasting of EV Charging Station Consumption

Theme:
Master Thesis in Software
Specialisation: Database Technology (DT)

Project Period:
Autumn Semester 2025

Project Group:
cs-25-dt-10-02

Participants:
Casper Søgård
Sebastian Truong

Supervisor:
Torben Bach Pedersen
Martin Moesmann

Page Numbers: 100

Date of Completion:
June 6, 2025

Abstract:

The rapid adoption of electric vehicles (EVs), driven by global sustainability goals, has led to widespread deployment of electric vehicle charging stations (EVCS), raising concerns about the stability of existing power infrastructure. This study addresses the challenge of forecasting EVCS electricity consumption to help mitigate transformer overload risks. Using time series data aggregated at an hourly resolution, we analyze a real-world dataset from Boulder, Colorado and a synthetic Danish residential dataset.

Two main tasks are explored: (1) consumption forecasting using MAE and Huber loss, and (2) overload detection using recall and MAE/Huber Loss. An ensemble model is proposed, combining diverse base learners (e.g. xPatch, PatchMixer, GRU, LSTM, AdaBoost, Random Forest, Gradient Boosting) trained on bootstrap samples. Results show that deep learning models, particularly PatchMixer and xPatch, outperform ensemble and standalone models in both forecasting and overload detection, though all models struggled with peak consumption values. While ensemble performance was mixed, issues were largely attributed to unstable configurations and underperforming base learners.

Acknowledgements

We would like to express our sincere gratitude to our supervisors, Torben and Martin, for their invaluable support throughout the course of this project. Your quick feedback and constructive suggestions have significantly shaped both the direction and quality of our work. From refining our research focus to improving the clarity of our analysis, your expertise has been instrumental at every stage. We truly appreciate your dedication, availability, and encouragement during this process.

Preface

Casper Søgård
csagar19@student.aau.dk

Sebastian Truong
struon20@student.aau.dk

Aalborg University, June 6, 2025

Contents

The use of Generative AI

Generative AI was used in various aspects of this study to facilitate the processes. ChatGPT was used to get an overview of topics, while aiding in refactoring sentences with Overleaf's Writefull. Furthermore, Copilot was used together with ChatGPT to aid in understanding error messages and to provide suggestions on how to fix them.

1 — Introduction

The global shift toward sustainable energy has accelerated the adoption of electric vehicles (EVs) worldwide, with international efforts of the United Nations Goal 7.4 promoting innovation and investment in clean energy infrastructure. As governments and industries push for carbon neutrality, EVs have become a key element in reducing emissions in the transportation sector. As a result, the number of electric vehicle charging stations (EVCS) is expanding rapidly to meet the increasing demand for EVs. Although this transition provides clear environmental benefits, it also poses growing challenges to both the current and future energy infrastructure and power grid.

Amidst these challenges, the increasing and often unpredictable electricity demand from EVCS introduces new risks to grid stability, including transformer overload, accelerated transformer degradation, and voltage instability. Transformer overload occurs when the local and regional demand for electricity exceeds the capacity of the transformer, leading to overheating, power outages, and accelerated transformer aging. The transformer sustains heavier loads during peak hours, which can effectively reduce the transformer's lifespan by inducing thermal stress and causing an insulation breakdown. Similarly, voltage instability can occur due to sudden surges or drops in the EV charging load, resulting in power quality issues such as voltage drops or spikes.

Understanding and managing EVCS energy consumption is therefore crucial for the longevity and operational reliability of the transformer. By analyzing historical data to predict future electricity demand, forecasting can help anticipate peak loads, identify potential overload scenarios, and determine how these scenarios should be handled.

Consequently, this report investigates the role of forecasting EVCS electricity consumption and its ability to capture potential overload scenarios. A key focus of the project is the proposal of an ensemble model, which combines the predictions of unique models to enhance the stability and robustness. By evaluating the proposed ensemble model against other models, the study seeks to determine whether the aggregation of multiple predictions offers more reliable and interpretable insights into EVCS demand and effectively captures transformer overloads.

The contributions of this study are as follows:

1. **Development of a Bagging Architecture:** A proposed ensemble model is introduced, incorporating a diverse set of base learners, including ensemble models (Random Forest, Gradient Boosting Trees, AdaBoost), deep learning architectures (LSTM, GRU), and well-performing general forecasting models (xPatch and Patch-

Mixer). These models are trained on bootstrapped training datasets, and their predictions are aggregated to leverage the unique contributions of each base learner. However, results showed that the ensemble's performance was mixed and often limited by unstable configurations and inconsistent base learners, reducing its effectiveness in generalization across datasets.

2. **Comprehensive Model Comparison:** The ensemble model is rigorously assessed against its standalone base models to evaluate whether model aggregation enhances the reliability and robustness of EVCS consumption forecasting. The evaluation is conducted over a 24-hour forecasting horizon with an hourly granularity. Experimental findings showed that deep learning models, especially PatchMixer and xPatch, consistently outperformed all other models in forecasting accuracy, achieving the lowest errors but occasionally struggling to capture peak consumption values.

3. **Assessment of Transformer Overload Detection Capabilities:** The study evaluates the ability of the ensemble model and its individual base learners to capture transformer overload events by classifying the predictions. The efficacy of the models is assessed utilizing the recall metric as a key indicator, highlighting their effectiveness in identifying transformer overloads. PatchMixer and xPatch achieved strong recall scores on the Colorado dataset (0.88 and 0.71, respectively), but their performance dropped on the more challenging SDU dataset (0.54 and 0.64).

2 — Background

This chapter presents two different categories of forecasting models: Deep Learning and Ensemble Learning. Each section presents the core methodologies of the models, providing a comprehensive overview.

2.1 Deep Learning

Deep Learning is a subfield within Machine Learning that utilize neural networks to model complex patterns in data from raw input data. These models have been applied in a wide range of applications, including computer vision, natural language processing, and time-series forecasting. This study will focus on these four deep learning architectures: Multilayer Perceptron (MLP), Convolutional Neural Network (CNN), Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU)

2.2 Multilayer Perception (MLP)

MLP is a commonly used multi-layered feed-forward neural network. [1] It contains one input layer U_{in} and an output layer U_{out} with one or more hidden layers $U^{(i)}_{hidden}$. Each neuron in the hidden and output layers computes a weighted sum of its input, defined as:

$$f^{(u)}_{\text{net}}(\mathbf{in}_u, \mathbf{w}_u) = \sum_{v \in \text{pred}(u)} w_{uv} \, \text{out}_v.$$

where u is the current neuron in the network, in_u is the input to the neuron of the previous layer, $pred(u)$ are predecessor neurons of the current neuron, w is the weight matrix, and out_v is the output of the current neuron.

2.3 Convolutional Neural Network (CNN)

CNN [2] is a feedforward neural network originally developed for image processing tasks. It has since been adapted for time series forecasting due to its strength in capturing local dependencies and patterns over time. In time series applications, 1D convolutions are employed, where filters slide over one dimension rather than two spatial dimensions, as is the case in image processing. The core idea of a CNN involves convolutional kernels, which are small, learnable filters that slide over the time axis of the input sequence, see Figure 2.1.

These kernels are designed to detect short-term patterns such as trends or repeating signals within a fixed window of previous time steps. Each convolutional operation is followed by a non-linear activation function and a pooling layer to reduce the dimensionality of the embedding and help detect more detailed patterns as more convolutional layers are passed through. Lastly, this is passed onto the fully connected layers to convert the temporal features into the output prediction dimensions. Two specific variants of 1D convolutional layers are particularly effective in time series tasks.
In standard 1D convolutions for time series, a kernel typically processes all input features (channels) together at each time step, capturing both temporal and cross-feature interactions simultaneously. However, this can be computationally expensive. Depthwise convolutions [3] simplify this by applying a separate 1D filter to each feature channel independently over time. Instead of learning combined representations across all features, each channel is processed in isolation. This greatly reduces the number of parameters and computational load, making the architecture more efficient. Depthwise convolutions focus on extracting temporal features within individual input channels without mixing information across them.
Pointwise convolutions [4] use a 1×1 kernel where each output channel is computed as a linear combination of all input channels in a single time step, allowing it to learn new combinations or projections of the features without looking at neighboring time steps.
Depthwise convolutions are often combined with Pointwise convolutions, enabling the models to capture both the temporal dependencies and inter-channel while being computationally lightweight.

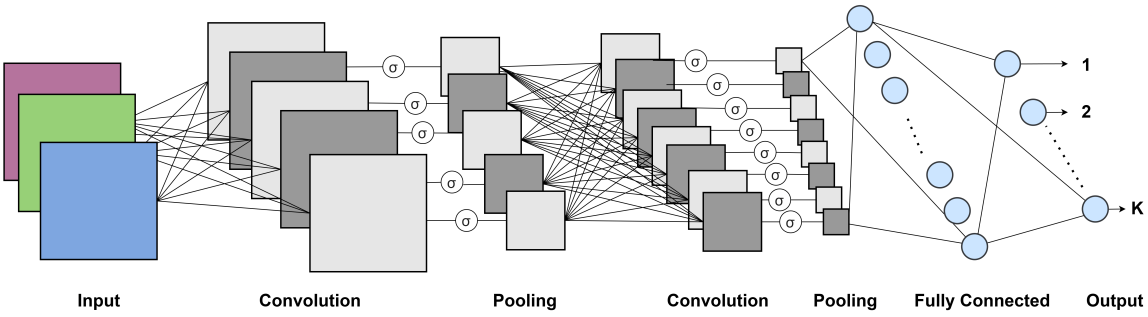

Figure 2.1: Shows the CNN architecture where σ denotes a loss function, inspired by this source [5]

2.3.1 Long Short-Term Memory (LSTM)

LSTM [6] is a type of Recurrent Neural Network (RNN) widely used to capture long-term dependencies of sequential data, e.g. time series. The intuition of RNNs is their ability to retain memory from previous inputs and use it to process subsequent inputs. To achieve this, LSTM uses a gating mechanism that controls the information flow throughout a memory cell [7]. LSTM cells use three different gates: an input gate, a forget gate, and an output gate. The gating mechanism makes LSTM effective for tasks requiring the modeling of long-term

dependencies and is responsible for controlling the information flow in the memory cell, and remembering the previous time steps. The memory cell state c_t is defined as:

$$c_t = f_T \odot c_{t-1} + i_t \odot g_t$$

where f_T is the forget gate, $\odot$ is the element-wise product, c_{t-1} is the previous cell memory, i_t input gate and g_t is the candidate cell.
The first gate is the forget gate f_t that controls which parts of the memory cell can be forgotten.

$$f_t = \sigma\Big(W_{xf}x_t + W_{hf}h_{t-1} + b_f\Big)$$

where σ is an activation function, W is a weight matrix, h is the hidden state, and b_f is the bias.
Afterwards, the input gate i_t handles which part of the new memory content is added to the memory cell.

$$i_t = \sigma\Big(W_{xi}x_t + W_{hi}h_{t-1} + b_i\Big)$$

Then, the newly proposed candidate memory cell g_t is passed to the memory cell c_t.

$$g_t = \tanh\Big(W_{xg}x_t + W_{hg}h_{t-1} + b_g\Big)$$

where $tanh$ is the hyperbolic tangent function.
Lastly, the output gate o_t controls how much of the memory cell c_t is used to compute the hidden state h_t. [8]

$$o_t = \sigma\Big(W_{xo}x_t + W_{ho}h_{t-1} + b_o\Big)$$
$$h_t = o_t \odot \tanh(c_t)$$

2.3.2 Gated Recurrent Unit (GRU)

Gated Recurrent Unit (GRU) is another variant of the RNN. GRU aims to capture the dependencies of different time scales with recurrent units and consists of two gates: an update gate and a reset gate. [8] Compared to LSTM, GRU combines the forget and input gates into a single update gate u_t. It also merges the cell state and hidden state into one hidden state h_t [7], defined as:

$$h_t = (1 - z_t) \odot ht_1 + z_t \odot h'_t$$

where h' is the candidate hidden state.
The update gate u is responsible for the amount of cell memory c to be retained, which is:

$$z_t = \sigma\Big(W_{xz}x_t + W_{hz}h_{t-1} + b_z\Big)$$

The reset gate r_t is given as:

$$r_t = \sigma\Big(W_{xr}x_t + W_{hr}h_{t-1} + b_r\Big)$$

Then, GRU determines how much of the hidden state h_t to forget when computing the new candidate memory h'_t, defined

$$h'_t = \tanh\Big(W_{xh}x_t + r_t \odot (W_{hh}h_{t_1}) + b_h\Big)$$

2.4 Ensemble Learning

Ensemble learning methods aim to improve predictive performance and generalization ability by combining the output of multiple base learners. [9] These techniques leverage the diversity among individual models to reduce variance, bias and improve the predictive power compared to an individual learner. Variance refers to how much the model's predictions change when trained on different subsets of the training data, where a high variance means the model is complex and overfits on noise and other unrelated patterns. Bias refers to how well the model can forecast on unseen data, where high bias means the model might be too simple to capture the underlying patterns, resulting in underfitting.
An ideal ensemble model should have a low variance and low bias to achieve an exceptional performing model, but this requires a good balance of variance and bias. This is required since complex models reduce bias but increase variance, and vice versa. Three main ensemble learning methods exist, which are the following: Bagging, Boosting, and Stacking.

Bagging, also called Bootstrap aggregating, involves splitting the training data for each base learning using random sampling with replacement, i.e. elements may be selected multiple times, to generate b different subsets used to train b base learners. The b base learners' predictions are then combined, usually by taking the mean of the predictions. One of the most common bagging models is Random Forest.

Boosting learners builds base learners sequentially. It uses the input data to train a weak learning model, computing the predictions from the weak learner, selecting the poorly predicted samples, and then training the next weak base learner with an adjusted training set comprising of the poorly predicted samples from the previous training round. This interactive process continues until a predefined number of base learners are completed. Common boosting models are AdaBoost and Gradient Boosting.

Stacking, or stacked generalization, is an ensemble learning technique that, unlike bagging and boosting, which typically use the same model, leverages the diversity of different models to capture various patterns in the data. Stacking combines multiple base models to improve predictive performance, where each base learner is trained on the same training dataset and generates predictions for a validation set. These predictions are then used as input features for a meta-learner, which is trained to combine them and find an optimal way to combine

the output of the base models. Finally, the meta-model produces the overall prediction on the unseen data.

2.4.1 Random Forest

Random Forest [10] is a widely used ensemble learning method for classification and regression tasks. The main concept of the model is to construct a multitude of decision trees during training time and output the average of the predictions over the trees in regression tasks and apply bootstrap sampling as seen in Bagging.
The Random Forest regressor learns a function f that maps historical input sequences to future predictions. Each tree in the ensemble learns a subset of this mapping, and the final prediction is computed by averaging the outputs of all T trees:

$$\hat{X} = f_{\mathrm{RF}}(X) = \frac{1}{N} \sum_{i=1}^{N} h_i(X),$$

where $h_i(\cdot)$ is the i-th regression tree trained on a bootstrap sample of the data, N is the total number of trees in the forest and X is the training set.
Having introduced Random Forest as a classical ensemble approach, the following sections will explore more prevalent ensemble models. These include boosting-based methods such as Gradient Boosting and AdaBoost.

2.4.2 AdaBoost

Adaptive Boosting (AdaBoost) [11] is a foundational model in ensemble learning. Unlike bagging-based methods such as Random Forests, which aim to reduce variance through averaging uncorrelated models, AdaBoost seeks to reduce bias by sequentially combining weak learners into a strong predictive model.
The model builds an additive model by fitting a sequence of weak regressors (or classifiers), typically decision stumps (one-level decision trees), in which each successive regressor focuses more on examples misclassified by previous ones. AdaBoost builds a regressor f_{AB} by iteratively adding T weak learners h_t with associated weights α_t:

$$\hat{X} = \sum_{t=1}^{T} \alpha_t h_t(X)$$

The weights are adjusted according to the error of the predictions by increasing the error prone learners. The subsequent regressors then focus on improving the mistakes of its predecessors.

2.4.3 Gradient Boosting

Gradient Boosting [12] is an ensemble learning method that generalizes boosting to arbitrary differentiable loss functions. It constructs a predictive model by sequentially adding weak learners, typically decision trees, where each learner is trained to correct the errors made previously in the ensemble. $f(x)$ denotes the gradient boosting model trained to approximate the mapping from past observations to future values. The prediction is thus given by:

$$\hat{X} = f(X) = \sum_{m=1}^{M} \gamma_m h_m(X),$$

where: $h_m(X)$ is the m-th weak learner, $\gamma_m \in \mathbb{R}$ is the weight assigned to the m-th learner, typically learned by minimizing a loss function, and M is the total number of boosting iterations.

3 — Problem Analysis

The large-scale adoption of EVs pushes the global number of charging points from almost 4 million in 2023 to almost 25 million by 2035, following the Announced Pledges Scenario (APS), including climate pledges and targets of the world governments. [13] In the USA, this trend is also seen with a goal of going from 206,000 to 500,000 charging ports by 2030. This goal is supported by a 635 million dollar grant from the U.S. Department of Transportation's Federal Highway Administration (FHWA) to continue building out EV charging and alternative fueling infrastructure.

3.1 EV Charging and User Charging Behavior

EV charging is generally categorized into three major levels: Level 1, Level 2, and DC Fast Charging. These levels differ significantly in terms of power delivery, charging speed, and typical use case. Table 3.1 summarizes key characteristics of each level, including voltage, power range, estimated range per hour of charge, and their prevalence among EVCS.

Charger Level	Voltage	Power Range	Range per Hour Charged	% of Public Chargers	Typical Use Case
Level 1 (AC)	120 V	1–1.9 kW	2–5 miles	$< 1\%$	Home
Level 2 (AC)	208–240 V	2.9–19.2 kW	10–20 miles	80%	Home, Workplace, Public
Fast Charger (DC)	400–1000 V	50–500 kW	180–240 miles	20%	Public

Table 3.1: Overview of charging levels in USA [14] [15]

EV charging patterns vary significantly over time and are heavily influenced by user behavior and the type of charging infrastructure utilized. Understanding these temporal load patterns is critical, as they directly affect how and when electricity demand is introduced to the power grid.

Residential charging, typically facilitated through Level 1 or Level 2 chargers, is the most common form of EV charging. It generally occurs in the evening hours when drivers return home from work. As a result, a substantial portion of residential charging demand is concentrated between 5:00 PM and 9:00 PM. This coincides with the traditional peak period for household electricity consumption due to lighting, cooking, heating or cooling, and appliance use. The convergence of residential EV charging and conventional household demand places a compounding strain on the local distribution grid.

In contrast, public charging behavior, particularly at workplaces and fast-charging stations, tends to exhibit different load profiles. Workplace charging, often using Level 2 infrastructure, typically peaks during regular working hours (e.g., 8:00 AM to 4:00 PM), distributing

demand more evenly across the daytime. Public fast chargers, however, experience more variable and unpredictable usage patterns. These chargers may see higher demand during weekends or holidays and can create short-term, high-intensity loads on the distribution grid. This divergence in charging behavior between residential and public settings creates distinct challenges for power system operators. While public charging stations can shift demand away from traditional peaks, their high power ratings and stochastic usage patterns can result in localized stress and voltage instability.
Understanding these behavioral dynamics is essential for the effective planning and integration of EV charging infrastructure into the power grid.

3.2 The Power Grid of the USA

The electricity supply chain consists of three primary segments. The first is generation, where electricity is produced. Generation can occur at a variety of sources, including fossil fuel power plants (e.g., coal, natural gas), nuclear power stations, and renewable energy facilities such as solar farms, wind parks, hydropower plants, biomass facilities, and geothermal stations.
Then, the transmission grid transports electricity over long distances through high-voltage power lines, where the voltage typically is between 115 kV and 765 kV. This larger current is sent to the distribution grid, which moves power over shorter distance to end users (households, businesses and smaller industrial sites) with voltage typical below rated below 34 kV. From this, a distribution substation takes the incoming voltage, typically around 34kV, and steps it down to a low-voltage power grid, usually at 120V, that connects to end users. [16] Sometimes, this voltage power is increased to 240V for heavier appliances such as electric stoves, washing machines, and clothes dryers. [17] Substations play a critical role in the connection of the generation, transmission, and distribution grid. Substations can be used for numerous distinct system functions, although most substations utilize transformers to either step up or step down the voltage to match the required voltage. The combination of the transmission and distribution grid with substations is also called the power grid. [18] This deployment of the power grid is fundamentally used universally worldwide.

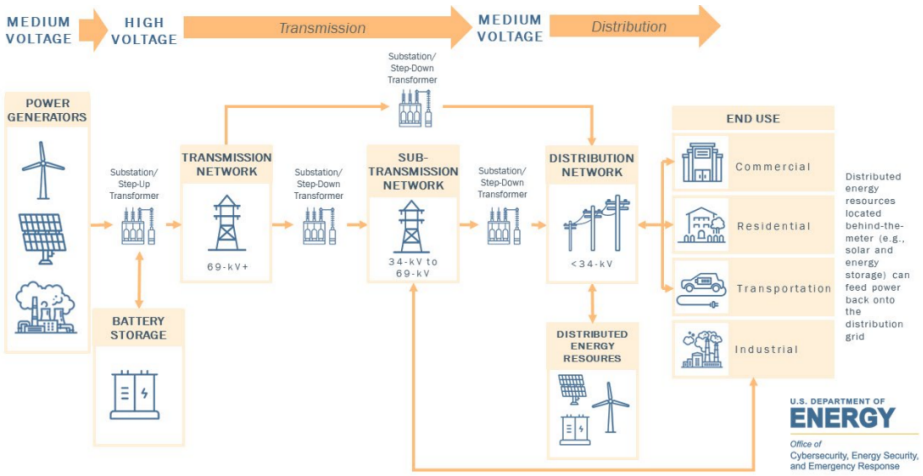

Figure 3.1: Overview of the USA Electric System from U.S. Department of Energy from [18]

3.3 EVCS Effect on the Power Grid

The rising number of EVCS introduces new charging loads that can overload the grid. Especially the distribution grid can encounter limitations due to the increasing power demand of EV charging, which primarily occurs in the distribution network. [19] These limitations can be manifested in several ways, including power quality and power transformer degradation. [20]

3.4 Power Quality

Power quality degradation and voltage excursions occur in two directions being: a voltage drop during high demand, and overvoltage during high demand and sudden load drops before regulating equipment can adjust. Existing voltage control devices can maintain line voltage within specifications for some load fluctuations, but large and rapid fluctuations cause wider excursions beyond the range of traditional voltage control devices. The problem then occurs when the charging loads are not aligned with the distribution generation peaks. This mismatch in voltage results in differences between the net load during charging hours. [20]

3.5 Transformer Overloading

Transformer overloading occurs when the power drawn on a line exceeds the rated limits of the component. Managing the loads, therefore could prevent transformer overloading, and not only power outages, but to mitigate transformer degradation. Furthermore, with the introduction of residential charging, the traditional cool-down periods at night are subsided

with a now more constant flow of energy consumption, and thus more transformer aging. [20]
The current transformers in the distribution may also be installed decades ago, potentially leading to more power outages, since they were not built to support the increasing loads from EV charging.
The amount of electrical demand a distribution transformer can safely carry is specified in a rating, called the transformer rating. The transformer is rated given its size, which depends on its load demand, voltage levels, power factor, efficiency, and overload capacity. The transformer rating is measured in Volt-ampere (VA). [21]

Transformer Overload Example

To illustrate how a transformer overload occurs in the power grid, a simplified model is made in Figure 3.2, where this source [22] is used for standard reference equations and values for the transformer. The example has the distribution network transporting electricity to the distribution step-down transformer. This transformer then distributes electricity throughout households and industries contributing to a collective baseload and three EVCS. Now, a prerequisite for a transformer overload is that the baseload combined with the EVCS consumption exceeds the limits of the transformer. To examine this, the study will calculate the utilization of the transformer in a simplified setup.

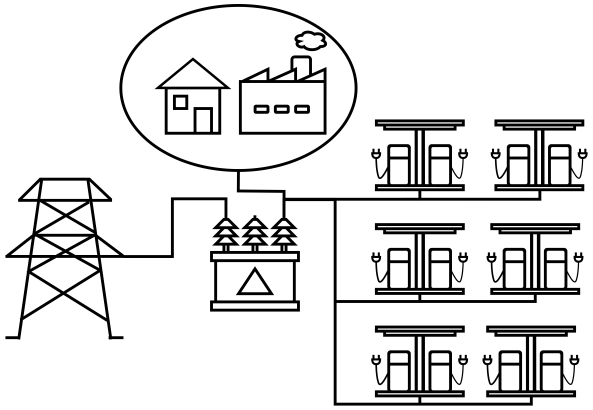

Figure 3.2: A simplified visualization of a transformer overload scenario

In this example, a $150kVA$ distribution transformer handles the load. To calculate the transformer utilization, the transformer rating should be converted to kW, which is done by:

$$kW = kVA \times PowerFactor$$

where kVA is the transformer rating and power factor is the effectively with which electrical power is converted.

For simplicity, the power factor = 1, meaning the transformer rating is $150kW$. Now, the example is set for calculating the transformer utilization at an arbitrary peak hour. Here, the baseload from residential houses and industry is 145 kW, and six EVCS with level 2 charging ports each utilizing the maximum charging power of 7.2 kW, resulting in $7.2kW \times 18 = 129.6kW$. The transformer loading ratio K is then calculated as:

$$K = \frac{S_L}{S_R} = \frac{145 + 129.6}{150} = 1.831$$

where S_L is the total load and S_R is the transformer rating. This indicates that the transformer is operating at 183.1% of its rated capacity, which means that it is significantly overloaded during this peak hour.

Transformer overloads result in excess heat being produced, which can thermally reduce the effectiveness and lifetime of the transformer. To figure out the effect this overload has on the transformer, the temperature of the oil within the transformer is calculated, which is referred to as the hottest-spot temperature. The hottest-spot temperature θ_H of the transformer can be calculated with the equation:

$$\theta_H = \theta_A + \Delta\theta_{TO} + \Delta\theta_H$$

where, θ_A is the ambient temperature, $\Delta\theta_{TO}$ is the top-oil temperature rise and $\Delta\theta_H$ is the change in the hot-spot temperature.

The top-oil temperature rise $\Delta\theta_{TO}$ describes the change in the transformer's rated top-oil rise, where the standard reference value of older transformers typically uses 55 °C, and newer transformers use 65°C. Given the previously found transformer loading ratio of 1.831, the top-oil temperature rise can be calculated as:

$$\Delta\theta_{TO} = \Delta\theta_{TO,R} \cdot K^1 = 55 \cdot 1.831 = 100.7°\text{C}$$

where $\Delta\theta_{TO,R}$ is the transformer's rated top-oil rise. K^1 refers to the average heating in the entire transformer, which scales linearly with increasing load and temperature.

The hot-spot temperature rise $\Delta\theta_H$ is then calculated to find the hottest spot in the winding insulation within the transformer. This is given by:

$$\Delta\theta_H = \Delta\theta_{H,R} \cdot K^{1.6} = 30 \cdot 1.831^{1.6} = 79.2°\text{C}$$

where $\Delta\theta_{H,R}$ is the rated hot-spot temperature rise of 30 °C as a standard reference value, and the $K^{1.6}$ exponent comes from the localized heating, which scales non-linearly with load due to increased current density and cooling limitations.

Given these formulas, the hottest-spot temperature can be calculated:

$$\theta_H = \theta_A + \Delta\theta_{TO} + \Delta\theta_H = 30 + 100.7 + 79.2 = 209.9°\text{C}$$

Looking at the table below, it can be seen that with a hottest-spot temperature of $\theta_H = 209°\text{C}$, the transformer will have a loss of life slightly above 0.5% after just 30 minutes.

Time (h) / % loss of life	0.05	0.10	0.25	0.50	1.00	2.00	4.00
0.5	171	180	193	204			
1	161	171	183	193	204		
2	153	161	174	183	193	204	
4	144	153	164	174	183	193	204
8	136	144	155	164	174	183	193
16	128	136	147	155	164	174	183
24	124	131	142	150	159	168	178

Table 3.2: Loss of Life Expectancy by Hottest-spot Temperature (°C), reproduced from [22]

3.6 EVCS Electricity Consumption Forecasting

Accurate forecasting of EVCS consumption is essential for addressing several technical challenges associated with large-scale EV adoption, including transformer overloading, voltage instability, and increased stress on distribution infrastructure. As the number of EVCS installations rises, so too does the strain they impose on local grids, potentially leading to transformer overload scenarios.

The task of forecasting EVCS consumption is inherently a time series problem, where the goal is to predict short-term electricity consumption patterns. These forecasts can inform grid operators about expected demand surges and help in grid load balancing, infrastructure planning, and preventive maintenance. Accurate short-term forecasts, in particular, are crucial for mitigating immediate overload risks.

Accurate modeling of EVCS demand is challenging due to several sources of variability, including user behavior, temporal charging habits, and contextual factors such as weather, holidays, and special events. Capturing the complex, often non-linear, dependencies in time series facilitates the usage of advanced modeling techniques capable of learning both temporal trends and sudden demand shifts.

To address this, the study proposes a bagging model, which focus on aggregating multiple unique base models, which will be referred to as the ensemble throughout the report. In doing so, the goal is to utilize different predictions from unique models to leverage the strengths of diverse model architectures, further described in Section 5.3. This is hypothesized to improve the predictive reliability of the models. However, this assumption has the prerequisite that the standalone models each are stable and can generate reliable and accurate results, since failing that could decrease its effectiveness due to its reliance on multiple model predictions.

Although electricity demand forecasting is generally framed as a regression task, identifying and mitigating transformer overloads requires an additional classification perspective. Classification metrics serve as additional tools to evaluate how effectively a model captures peak EVCS consumption, since critical periods pose a high risk of overloading transformers over

an extended period of time.
In this study, both regression and classification tasks are integrated to provide a comprehensive evaluation of model performance. Regression assesses the accuracy of consumption forecasts, while classification measures the model's ability to capture critical overload events. Together, these perspectives aim to support proactive grid management and ensure the reliability of EVCS infrastructure under increasing demand. The following section outlines the scope of the study and the methodological approach used to evaluate and compare forecasting models.

3.7 Scope of the Study

This study focuses on forecasting the electricity consumption of EVCS by analyzing time series with the objective of providing information on demand and identifying transformer overloads. The time series are aggregated at an hourly resolution with a forecast horizon of 24 hours. The data is divided into a training, validation, and test set using a 60/20/20 split. By analyzing EVCS charging sessions, the study aims to identify peak consumption periods, potential overload scenarios, and temporal trends in charging behavior. These insights are intended to address the challenges previously described by transformer overloading.
To this end, the study evaluates a range of forecasting models, with a particular emphasis on creating the ensemble. By aggregating predictions from multiple unique models, the ensemble is hypothesized to provide improved reliability and robustness of the predictions, which may prove useful in counteracting transformer overloads. The effectiveness of the ensemble is assessed against standalone models to determine their value in accurately forecasting EVCS demand and capturing overload events.
To detect potential overloads, the study formulates the problem as a classification task. Specifically, it determines whether a transformer is expected to be overloaded based on the combined forecast of EVCS electricity consumption, the local baseload, and the local transformer rating. The classification can result in either a true positive (TP), true negative (TN), false positive (FP) or a false negative (FN).

- TP: an overload is correctly predicted
- TN: a non-overload is correctly predicted
- FP: an overload is falsely predicted, and no overload actually occurs
- FN: an actual overload occurs, but is not predicted

Of these four outcomes, an FN means an overload event was not captured, which can have very severe effects and lead to transformer overloading. Since it was not anticipated, it

gives less time to mitigate overloads and adjust EVCS consumption. This particular scenario therefore poses a significant risk, which is important to handle properly to ensure infrastructure reliability.

3.7.1 Formal Definition of the Forecasting Problem

To formalize the forecasting task, consider an EVCS electricity consumption as a multivariate time series with hourly granularity. The input sequence consists of a lookback window length T, representing the previous historical observations. This is mathematically expressed as:

$$X = (x_{t-T+1}, x_{t_T+2}, \ldots, x_t)$$

where $x_t \in \mathbb{R}^M$ represents the multivariate observation at time t, and M denotes the number of features (e.g. electricity consumption, day of the week, season etc.). The objective is to predict the electricity consumption for the next 24 hours, i.e. a forecasting horizon $H = 24$. The prediction sequence is defined as:

$$\hat{X} = (\hat{x}_{t+1}, \hat{x}_{t+2}, \ldots, \hat{x}_{t+H})$$

.

where each predicted value $\hat{x}_{t+i} \in \mathbb{R}^M$ corresponds to the univariate forecast at hour $t + i$. Thus, the input lookback sequence is $x \in \mathbb{R}^{M \times T}$ and the predicted output sequence $x \in \mathbb{R}^{M \times H}$.

The total load on the transformer is then given as: $\hat{L}_{t+H} = \hat{X}_{ev,t+H} + X_{base,t+H}$ where $\hat{X}_{evcs}$ is the total load forecast and X_{base} is the baseload.

To evaluate the performance of predictive models, several error metrics are commonly used, each offering unique insights into forecasting accuracy.

One of the most intuitive metrics is the Mean Absolute Error (MAE), which calculates the average magnitude of the errors between predicted and actual values, without considering their direction. This gives a straightforward measure of overall prediction accuracy being [23]:

$$MAE = \frac{1}{H} \sum_{t=1}^{H} ||x_t - \hat{x}_t||$$

where x_t is the actual value, $\hat{x}_t$ is the predicted value at time t, and H is the forecast horizon. Since all errors contribute equally, MAE provides a clear, unbiased estimate of typical forecast error.

In contrast, the Mean Squared Error (MSE) emphasizes larger errors by squaring the differences. MSE is defined as [23]:

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

This makes MSE particularly sensitive to outliers, as large deviations have a disproportionately large impact on the result. While this can be advantageous when large errors are especially undesirable, it may also skew evaluation in the presence of occasional anomalies. For situations where interpretability is key, the Mean Absolute Percentage Error (MAPE) is often preferred. By expressing errors as percentages of the actual values, MAPE offers a scale-independent perspective defined as [23]:

$$MAPE = \frac{1}{H}\sum_{t=1}^{H} || \frac{x_t - \hat{x}_t}{x_t} ||$$

This makes it easy to understand the typical error in relative terms, though it can become unstable when actual values approach zero.
When comparing performance across different datasets, the Normalized Root Mean Square Error (NRMSE) is useful. It adjusts the RMSE by a normalization factor, such as the range of the actual values seen in the equation [24]:

$$NRMSE = \frac{\sqrt{\frac{1}{H}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}}{y_{max} - y_{min}}$$

This normalization allows NRMSE to reflect error in proportion to the variability of the data, making it suitable for cross-dataset comparisons.
Lastly, Huber loss offers a balanced alternative that combines the strengths of MAE and MSE. It behaves like MSE for small errors, encouraging precision, but transitions to a linear MAE-like form for large errors, which helps reduce sensitivity to outliers. Huber loss is defined as [25]:

$$L_\delta(x_t, \hat{x}_t) = \begin{cases} \frac{1}{2}(x_t - \hat{x}_t)^2 & \text{if } |x_t - \hat{x}_t| \leq \delta \\ \delta\left(|x_t - \hat{x}_t| - \frac{1}{2}\delta\right) & \text{otherwise} \end{cases}$$

where $\delta > 0$ is a tunable threshold that controls the transition between quadratic and linear behavior. The lower values of δ emphasize the linear behavior of the loss function, increasing the robustness to outliers, while the higher values shift the focus to quadratic behavior, increasing the sensitivity to smaller deviations.

3.7.2 Formal Definition of the Overload Classification Problem

The overload can formally be defined as a binary classification problem where an overload occurs when the electricity consumption exceeds a certain threshold T, which is defined as:

$$\hat{Y} = \begin{cases} 1 & \text{if } \hat{L}_{t+H} > T(overload) \\ 0 & \text{otherwise} \end{cases}$$

To evaluate the model's ability to correctly identify transformer overloads, the study has implemented three classification metrics: accuracy, precision, and recall. [26]
Accuracy measures the overall correctness of the predictions:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

This metric provides a general overview of predictive performance, but this can be misleading. To illustrate that, assume 100 total predictions were made, where 10 of them are actual overloads. Of these, only 7 overloads (TP) are predicted, whereas 5 are missed (FN), and 85 out of 90 are correctly classified as non-overloads (TN), and 3 were wrongly classified as overloads (FP). The accuracy would then be:

$$\text{Accuracy} = \frac{7 + 85}{100} = 0.92$$

Although it can give a good understanding of the accuracy of the predictions, it hides the fact that 5 of the overloads were missed, which means that other metrics are needed.
Precision evaluates the reliability of overload predictions by measuring the proportion of predicted overloads that are genuinely overload events:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Considering the previously introduced example, where 7 correctly classified overloads and 3 non-overloads were incorrectly classified as overloads, the precision is calculated as:

$$Precision = \frac{7}{7 + 3} = 0.7$$

This indicates that 70% of the predicted overloads were correct, while 30% were false alarms. Although a precision of 0.7 suggests moderate reliability in overload predictions, it only tells part of the story, since it does not account for the missed overloads, which are most critical, as the recall metric.
Recall measures the model's ability to detect all overloads and whether the overloads were correctly identified:

$$\text{Recall} = \frac{TP}{TP + FN}$$

Continuing from the previous example, where 7 overloads were correctly identified and 5 overload events were missed, the recall is:

$$\text{Recall} = \frac{7}{7 + 5} = 0.58$$

This illustrates that only 58% of the actual overloads were detected successfully by the model, while 42% were missed, representing a more accurate picture of the model's ability to capture overloads. In scenarios where failing to detect an overload can have serious consequences, recall becomes a particularly important metric.

Although accuracy, precision, and recall will all be used to evaluate model performance, this study prioritizes recall due to its critical importance in effective overload detection. The other metrics will serve to support and validate the reliability of the recall metric.
Based on the considerations above, the following problem definition is proposed:

3.8 Problem Definition

How can an ensemble model, aggregating unique model predictions, enhance the reliability of EVCS electricity consumption forecasting and aid in effective transformer overload capturing?

4 — Related Work

This chapter reviews existing forecasting models, highlighting their approaches and scope limitations. The chapter introduces recent state-of-the-art forecasting models that have been examined in the EVCS domain. Furthermore, the chapter explores powerful general forecasting models that have the potential to enhance the robustness of aggregated forecasting models and improve their effectiveness in detecting transformer overloads.

4.1 State-of-the-Art Models for EVCS Consumption Forecasting

A wide range of forecasting strategies have been developed to predict electricity consumption at EVCS. This study focuses on two major categories: deep learning models, particularly Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks, and ensemble methods, such as Gradient Boosting, Random Forest, and AdaBoost. Each has shown varying degrees of success in modeling EVCS consumption patterns.

Among deep learning models, LSTM and GRU are among the most frequently used for EVCS forecasting due to their effectiveness in modeling temporal dependencies across a variety of domains [27, 28, 29, 30, 7]. A 2022 study [28] evaluated the performance of LSTM, GRU, Artificial Neural Networks (ANN), and Recurrent Neural Networks (RNN) using data from two public EVCS in Morocco, comprising of 2,000 observations. The models were tested with one to three hidden layers, where GRU demonstrated superior performance, achieving a MAPE of 2.90% with a single hidden layer, compared to LSTM's best result of 3.42% with three hidden layers.

The ensemble learning models: Gradient Boosting, Random Forest, Ada Boosting, have also been used widely for EVCS electricity consumption forecasting. [31, 32, 33] A notable study from 2024 [31] compared various ensemble and deep learning models on a comprehensive dataset from Germany consisting of over 500 charging locations. The models examined include Ada Boosting, Random Forest, LSTM, and Gradient Boosting. The study employed a forecasting horizon of 24 hours with a 15-minute granularity and used a rolling window setup spanning one week. Here, AdaBoost and Random Forest had the best results with normalized Root Mean Squared Error (nRMSE) scores of 0.418 and 0.411 respectively. Comparatively, LSTM and Gradient Boosting achieved worse scores of 0.610 and 0.508 showing the relative robustness of some of the traditional ensemble methods.

Some recent studies aim to combine linear and non-linear modeling approaches to capture

both trend and complexity in consumption behavior. A notable example is a 2022 study [34] that proposed a hybrid model combining Seasonal AutoRegressive Integrated Moving Average (SARIMA) with LSTM. SARIMA first models the linear, seasonal components of the data. The residuals, representing the non-linear portion, are then predicted using an LSTM network. The final forecast is constructed by summing the SARIMA and LSTM outputs.
This hybrid model was tested on an EVCS dataset from Spain with hourly resolution and an 80/20 train-test split. The SARIMA–LSTM model outperformed several alternatives, including ARIMA, LSTM, Extreme Learning Machine, Support Vector Regression, and ARIMA–LSTM. It achieved a MAPE of 4.61%, outperforming the next-best model (ARIMA–LSTM) which had a MAPE of 5.86%.

4.2 State-of-the-Art General Time Series Forecasting Models

This section presents three recent general time series forecasting models that can be added to the aggregate of multiple individual models due to their strong predictive performance in a variety of datasets, which can improve the reliability of predictions. Specifically, the models considered are the Deep-shallow multi-frequency Pattern Disentangling neural network (D-PAD), Exponential Patch (xPatch), and PatchMixer. These models have demonstrated exceptional performance across seven widely benchmarked real-world datasets, according to the rankings provided by Papers with Code [35] (accessed on 05/03/2025). The seven datasets are:

- ETT captures the electricity transformer temperature with an hourly granularity for four datasets: ETTh1, ETTh2, ETTm1, and ETTm2. [36]
- An hourly electricity consumption data of 321 clients from 2012 to 2014. [37]
- Traffic with hourly data describing the road occupancy rates measured by 862 sensors on San Francisco Bay area freeways. [37]
- Weather data with 21 weather sensors with a 10-minute granularity at the Weather Station of the Max Planck Biogeochemistry Institute in 2020. [38]

4.2.1 Deep-Shallow Multi-Frequency Patterns Disentangling Neural Network (D-PAD)

D-PAD [39] is a model that consists of two modules, see Figure 4.1. The decomposition-reconstruction-decomposition (D-R-D) module is responsible for decomposing the signals into simpler signals to leverage the various local characteristics of the signals and learn information scattered and mixed among different components, whereas the interaction and

fusion (IF) module represents these different frequency patterns in graphs to model their potential interactions between the patterns.
The decomposition of the signals, see Figure 4.1(a), is found in the multi-component decomposing (MCD) block within the D-R-D module, which utilizes mathematical morphology, a field based on set theory, to transform the signals in the time series. In detail, the morphology operations: dilation and erosion are applied to calculate and generate an upper and lower envelope curve of the time series. The upper envelope, obtained by dilation, extends the signal by emphasizing the local maxima, expanding the function's range upward. Similarly, the lower envelope, found by erosion, follows the local minima and shrinks the signal downward. By morphing the different frequency signals with these operations, the local characteristics of the frequency are captured. These MCD blocks are layered on top of tree-like structured D-R blocks, see the lower part of Figure 4.1(a), that integrate a branch guidance generator (BGG), see Figure 4.1(b), to guide the selection of the components obtained by the MCD block, together with numerous MLPs.
As a result, the best selection of components is then selected among a tree-like structure of decomposition-reconstruction (D-R) blocks, where each component is passed on to the IF module.
The information passed to the IF module is then decomposed into a graph representation, see the upper part of Figure 4.1(a), to capture the different frequency patterns and to model the potential interactions among them. These potential interactions are mapped onto a self-adaptive adjacency matrix, where different interactions between components of different signal frequency patterns are mapped onto multiple graphs.
These features are then summed up and passed to an MLP for a final prediction:

$$\hat{X} = MLP(Z_{out})$$

where Z_{out} is the output of the IF module.

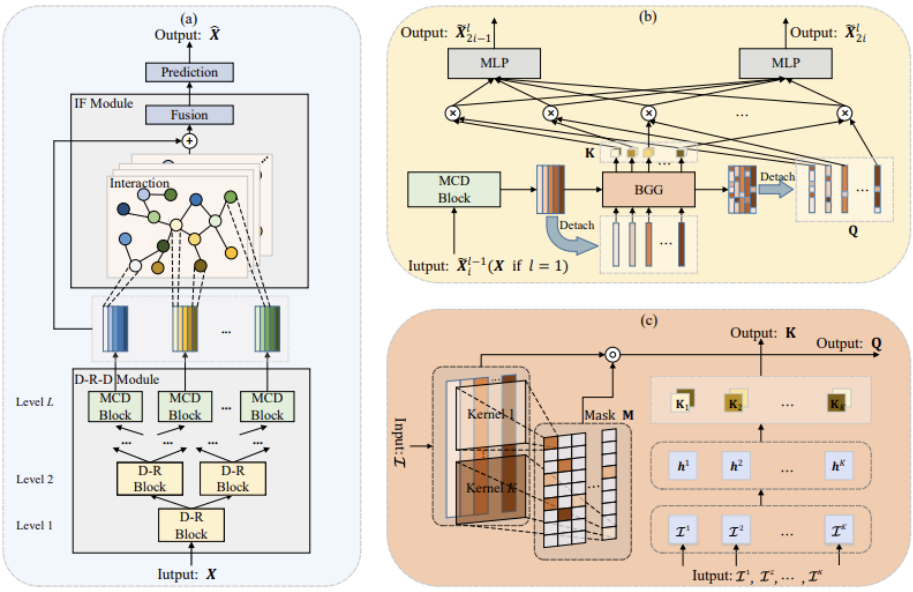

Figure 4.1: Architecture of D-PAD from the original paper. [39] (a) D-PAD consisting of two parts, the D-R-D module and interaction and fusion (IF) module. (b) The D-R block decomposes a series into multiple components and reconstructs them to two new series. (c) BGG combines convolutions and projections generating **Q** and **K** to guide the branch selection for each component.

4.2.2 Exponential Patch (xPatch)

xPatch [40] is a model composed of two layers, a linear layer and a non-linear layer, which uses an Exponential Moving Average (EMA) scheme to decompose the time series into a seasonal component and a trend component. These components differ in their focus on the dataset, where the trend component captures the long-term direction of the data, and the seasonal component captures the repeating patterns found at e.g. a daily, weekly, and monthly intervals, resulting in non-linearity.

The trend component is handled with a linear block that is based on an MLP network, see the lower part of Figure 4.2. The first part of the block consists of two consecutive blocks consisting of a fully connected layer, average pooling, and layer normalization. Here, the linear layer and average pooling operations compress the dimensions of the feature to retain only the most significant features of the smoothened trend, thus fitting the available space effectively. After the two blocks, the result is upscaled to merge with the result from the non-linear block for a final prediction.

The seasonal component is handled with a non-linear block based on a patching scheme and a CNN-based model, see the upper part of Figure 4.2. In this scheme, the time series is divided into smaller, fixed-length segments called patches. These patches are extracted by sliding a window across the sequence using a specified stride. This approach reduces computational load while preserving important temporal patterns within each patch. [41]

Consequently, the patching unfolds the time series using a sliding window, enabling the block to focus on the repetitive seasonal features and the dependencies between the patterns. Following that, the CNN-based stream, consisting of a depthwise convolution and pointwise convolution, is used to capture the spatio-temporal patterns and inter-patch correlations, focusing on the non-linear features of the seasonal data. The depthwise convolution captures temporal features of the patches, whereas the pointwise convolution captures the inter-patch feature correlations. Both of the CNN-blocks are followed by the Gaussian Error Linear Units (GELU) activation function to smoothen transitions around zero normalized via BatchNorm. For the depthwise convolution, a residual stream is utilized together with the embedding from the depthwise convolution to retain the patching embeddings further into the stream. These predictions are then processed to merge with the linear block for the final prediction.
The final prediction merges the blocks to a final prediction, resulting in:

$$\hat{X} = Linear(concat(\hat{X}^i_{lin}, \hat{X}^i_{nonlin}))$$

where $\hat{X}^i_{lin}$ is the prediction in the linear block and $\hat{X}^i_{nonlin}$ is the prediction in the non-linear block.

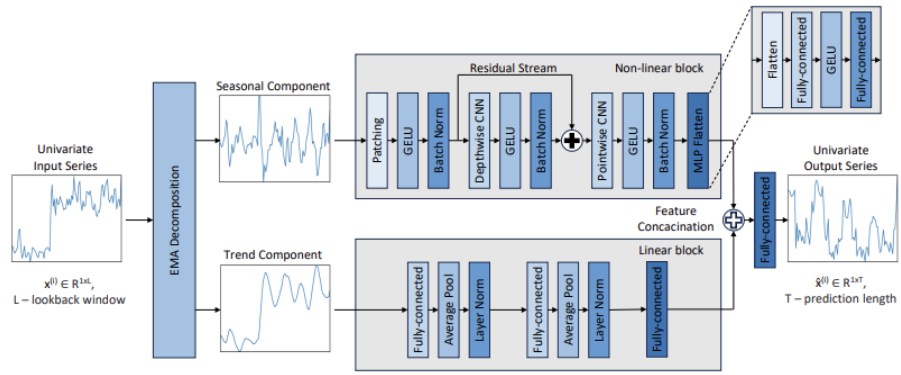

Figure 4.2: The architecture of xPatch from the original paper. [40] The time series is decomposed into a seasonal and trend component and processed through a linear and non-linear block.

4.2.3 PatchMixer

PatchMixer [42], see Figure 4.3, employs a single stream block based on a CNN network, named the PatchMixer block. The model utilizes mean and standard deviation to normalize the time series. Then, similar to xPatch, Patch Embedding is used to split the time series into patches to capture local temporal patterns. Here, a residual connection, called Linear Flatten Head, is used to preserve the global trends in the time series without further processing and merge it into the final prediction.
Afterwards, the PatchMixer block consisting of a depthwise and pointwise convolution is used followed by the GELU and BatchNorm scheme, also seen in xPatch.

Lastly, an MLP consisting of a flatten, fully-connected, GELU, and fully-connected layer is deployed to capture the nonlinear and broad dynamics of the temporal patterns, and merged with the Linear Flatten Head residual connection for a final prediction.

$$\hat{X} = concat(\hat{X}_{MLP}, \hat{X}_{resid})$$

where $\hat{X}_{MLP}$ is the prediction after the processing of the PatchMixer block and the MLP flatten head, and $\hat{X}_{resid}$ is the residual output from the learned projection matrix from the Patch Embedding.

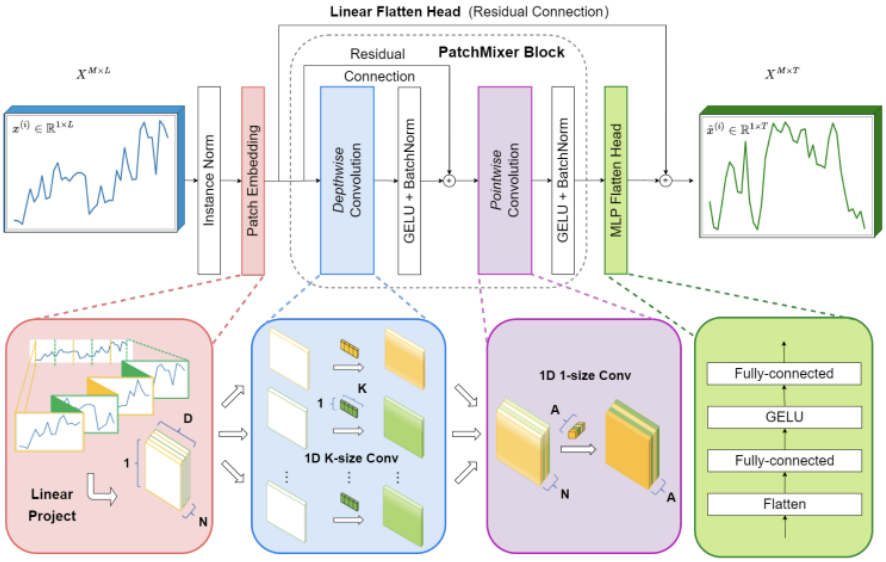

Figure 4.3: The architecture of PatchMixer from the original paper. [42] It employs a patch-based depthwise convolution module called Patch-Mixer Block together with an MLP based model to predict on the time series.

5 — Methodology

This chapter begins by introducing the datasets utilized in the study, along with the preprocessing techniques applied to prepare them for analysis. It then outlines the models selected for evaluation and the technologies used for their implementation. Lastly, the experimental design is described in detail, including dedicated sections on the hyperparameter tuning process and the core experimental procedures.

5.1 Analysis of Data

Accurate forecasting of EV energy consumption relies on access to high-quality data from charging stations. However, such data has been difficult to obtain in Denmark, as many municipalities and private companies do not make their datasets publicly available. To address this limitation, this study draws on two distinct data sources, each offering valuable but different insights into EV charging behavior.
The first is a publicly accessible dataset from Colorado, USA, which contains real-world data from public EV charging stations. This dataset provides a practical foundation for understanding large-scale, real-world usage patterns in public charging infrastructure. This data is supplemented by a baseload dataset that describes the overall consumption of Colorado. The second is a synthetic dataset based on residential charging behavior in Denmark, developed as part of a Ph.D. thesis. Although synthetic, it is tailored to emulate Danish residential charging and allows for analysis of residential charging patterns.
By incorporating both datasets, with one reflecting actual public charging behavior and the other simulating residential use, this study is able to explore a broader range of forecasting scenarios and model the diverse nature of EV energy consumption more comprehensively.

5.1.1 Colorado Dataset

The Colorado dataset is a real-world dataset for the City of Boulder in Colorado, USA. The dataset contains charging sessions from city-owned level 2 EVCS, spanning from 01-01-2018 to 30-11-2023. It initially includes 19 EVCS, but the rapid expansion of charging stations increases the number to 46 by the end of the dataset. The dataset, see Table 5.1, contains sessions with information on *energy consumption*, *charging time duration*, location, *station name*, and more. [43]
To enable time series analysis, the charging session data was converted to an hourly format. The aggregation method distributes each session's total electricity consumption proportion-

Station Name	Address	State Province	Zip/Postal Code	Start DateTime	End DateTime	Total Duration	Charging Time	Energy Consumption
BOULDER / JUNCTION ST1	2280 Junction Pl	Colorado	80301	2018-01-01 17:49:00	2018-01-01 19:52:00	0 days 02:03:02	0 days 02:02:44	6.504000
BOULDER / JUNCTION ST1	2280 Junction Pl	Colorado	80301	2018-01-02 08:52:00	2018-01-02 09:16:00	0 days 00:24:34	0 days 00:24:19	2.481000
BOULDER / JUNCTION ST1	2280 Junction Pl	Colorado	80301	2018-01-02 21:11:00	2018-01-03 06:23:00	0 days 09:12:21	0 days 03:40:52	15.046000
BOULDER / ALPINE ST1	1275 Alpine Ave	Colorado	80304	2018-01-03 09:19:00	2018-01-03 11:14:00	0 days 01:54:51	0 days 01:54:29	6.947000
BOULDER / BASELINE ST1	900 Baseline Rd	Colorado	80302	2018-01-03 14:13:00	2018-01-03 14:30:00	0 days 00:16:58	0 days 00:16:44	1.800000

Table 5.1: First five entries of the original Colorado dataset

ally throughout the session duration. This results in an hourly electricity consumption data presented in Figure 5.1.

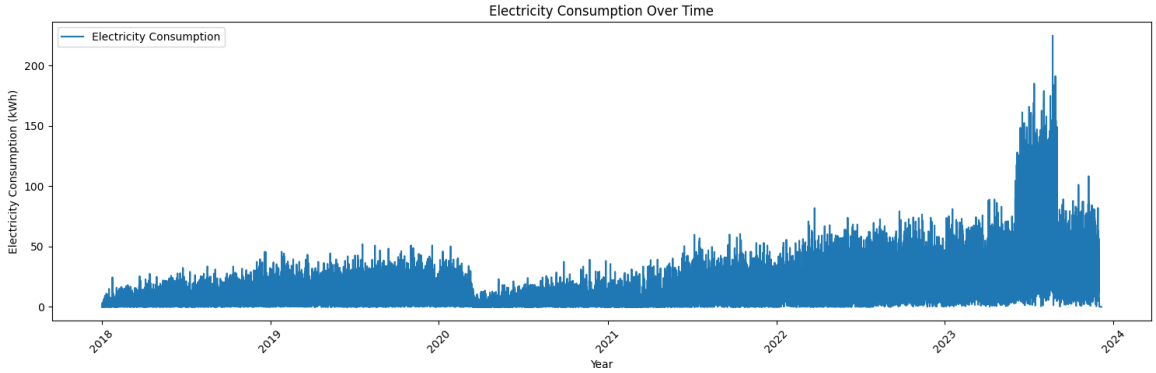

Figure 5.1: Charging session split into corresponding hours

Since the energy consumption was negatively impacted by COVID-19, seen in the year 2020, the dataset was filtered to only contain data from 30-05-2021 to 30-05-2023. The dataset is divided into a 60/20/20 split, consisting of a 60% training set, 20% validation set, and 20% test set, as seen in Figure 5.2.

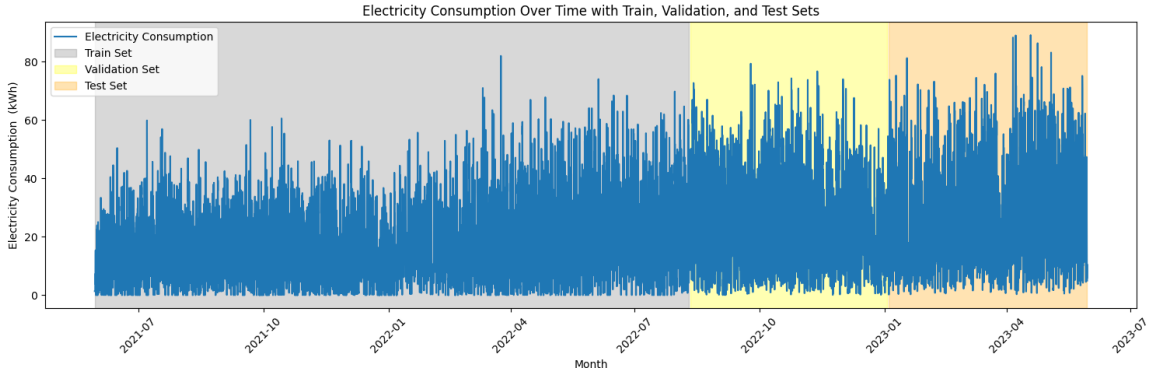

Figure 5.2: The filtered dataset with a train, validation, and test 60/20/20 split

The average daily energy consumption for a week, see Figure 5.3, remains relatively consistent, where consumption peaks on Fridays and Saturdays, and drops around Sundays and

Mondays. This pattern can be attributed to the fact that EVCS are generally located in public spaces, where more traffic is expected during peak hours.

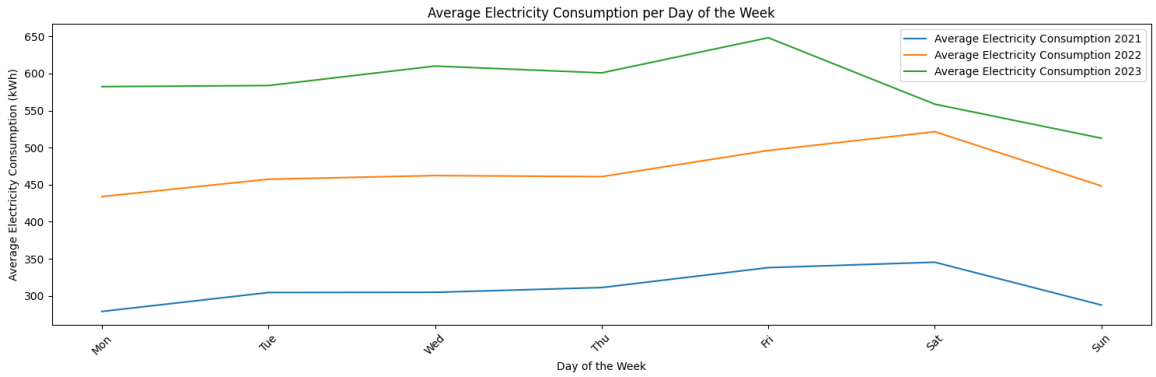

Figure 5.3: Average energy consumption per day of the week

As shown in Figure 5.4, EV charging occurs between 06:00 and 21:00 primarily, with peak consumption observed between 10:00 and 11:00. This trend may be due to the charger's location, often located in commercial and public areas where users typically charge their vehicles during daytime hours while at work, shopping, or running errands.

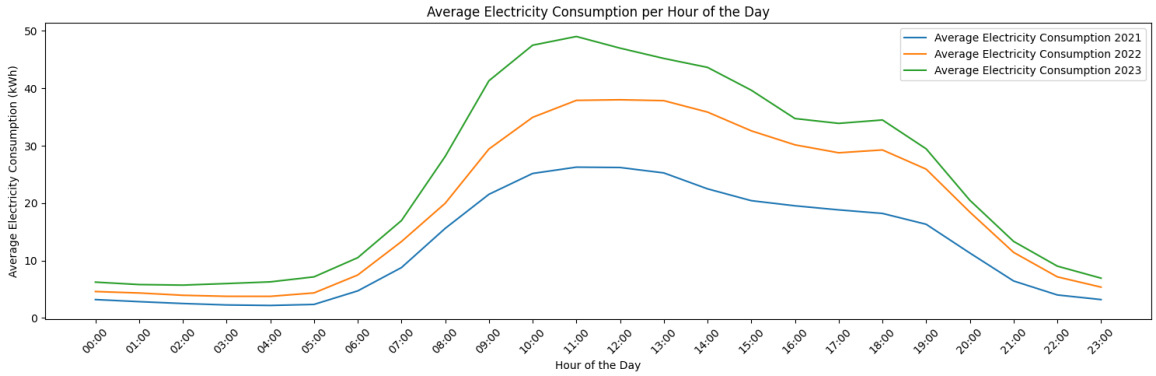

Figure 5.4: The average hourly energy consumption

5.1.2 SDU Dataset

The SDU dataset is a synthetic dataset based on a published PhD thesis [44] by Kristoffer Christensen. The publication uses a multi-agent-based simulation framework to generate the synthetic dataset simulating home charging to evaluate the development of the grid load in the future. This dataset is significantly different from the Colorado dataset, since it is a synthetic dataset modeled after residential charging to simulate the yearly residential charging increase and how grid overloads can occur in the future. The simulation is based on the Danish radial distribution network with 126 residents, each with their respective

residential charger. The table of the dataset can be seen in Table 5.2. The original dataset spans from 01-01-2020 to 31-12-2032 with an hourly granularity and contains information about the *total number of EVs*, *total number of charging EVs*, *total grid load*, *aggregated baseload*, *aggregated charging load*, and *overload duration*.

	Timestamp	Total number of EVs	Number of charging EVs	Number of driving EVs	Aggregated base load	Aggregated charging load	Overload duration [min]
0	2020-01-01 00:00:07	1.00	0.00	0.00	62.94	0.00	0.00
1	2020-01-01 01:00:07	1.00	0.00	0.00	53.19	0.00	0.00
2	2020-01-01 02:00:07	1.00	0.00	0.00	49.46	0.00	0.00
3	2020-01-01 03:00:07	1.00	0.00	0.00	45.42	0.00	0.00
4	2020-01-01 04:00:07	1.00	0.00	0.00	39.57	0.00	0.00

Table 5.2: Overview of the SDU Dataset, with some features excluded

The simulation initially consists of one EVCS in 2024, with the number of EVs gradually increasing yearly. This study focus on the period from December 31, 2029 to December 31, 2030 and has divided the dataset into a training, validation, and test sets using a 60/20/20 split, as illustrated in Figure 5.5. The slight upward trend of consumption, seen in Figure in Figure 5.5 can be attributed to the increasing number of EVs over time, where the number of EVs goes from 49 to 69 during the specified period.

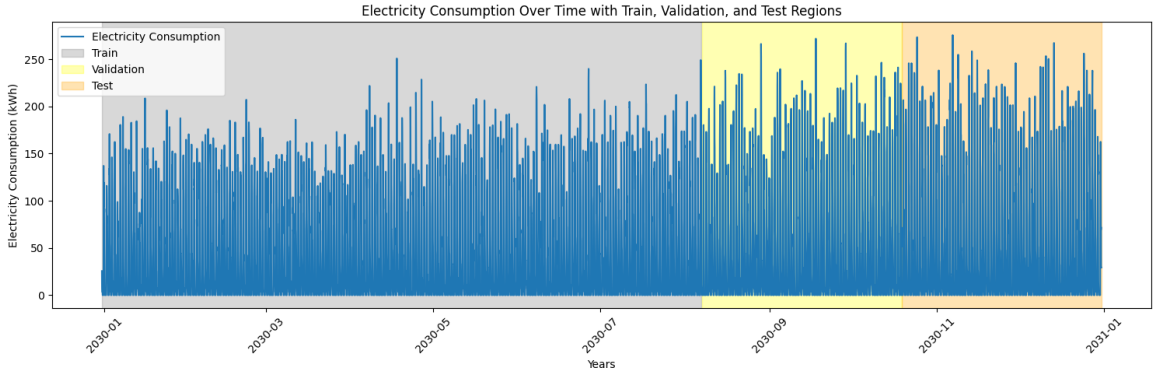

Figure 5.5: Charging session for the SDU Dataset

The hourly electricity consumption, illustrated in Figure 5.6, reveals a sparse dataset characterized by a high frequency of zero values. Most consumption values remain at zero between 04:00 and 13:00, followed by a noticeable peak between 15:00 and 19:00. This pattern aligns with typical residential charging behavior, where charging predominantly begins after work hours. As the battery approaches full capacity, the charging rate naturally decreases, contributing to the gradual decrease over time.

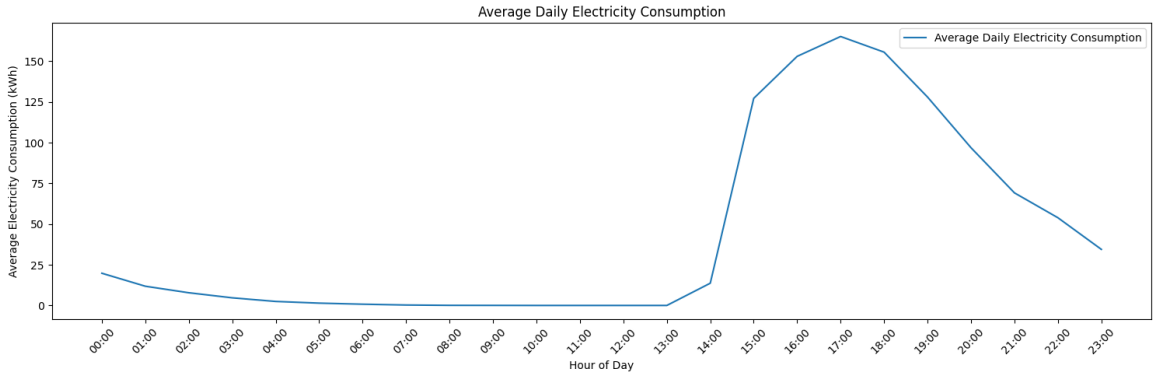

Figure 5.6: Hourly average electricity consumption for the SDU dataset

5.1.3 Colorado Baseload Dataset

As this study focuses on detecting transformer overload events, it is essential to establish a reliable baseload for the Colorado dataset. The baseload data is sourced from the IA-930 dataset, published by the U.S. Energy Information Administration (EIA), which provides hourly electricity demand data across 48 U.S. states. [43]

For Colorado specifically, the data are reported by the Public Service Company of Colorado (PSCO), which serves as the balancing authority (BA) for the region. Although this dataset does not provide electricity consumption figures exclusive to Boulder, it represents the aggregated demand across the entire PSCO service area, which includes Boulder.

Since the data is aggregated at the BA level, it does not offer insights into sector-specific consumption. However, according to the EIA, the commercial and residential sectors combined account for approximately 75% of Colorado's total electricity use, with the industrial sector contributing just over 25%. [45] The baseload is measured in the period from August 8, 2022 to March 3, 2023, which can be seen in Table 5.3.

BA Code	Timestamp	Demand (MWh)	Demand Forecast (MWh)	Net Generation (MWh)	Total Interchange (MWh)
PSCO	8/11/2023 12 a.m. MDT	5842	5690	5950	108
PSCO	8/11/2023 1 a.m. MDT	5430	5272	5702	272
PSCO	8/11/2023 2 a.m. MDT	5079	4969	5398	319
PSCO	8/11/2023 3 a.m. MDT	4956	4726	5247	291
PSCO	8/11/2023 4 a.m. MDT	4846	4535	5059	213

Table 5.3: Power demand and generation data for PSCO on August 11, 2022

Figure 5.7 presents the hourly electricity consumption for the PSCO service area. The dataset is split into two parts, the validation set and the test set, since the baseload is only required in evaluating the overload capturing.

The dataset includes some missing data, with the most notable gap occurring between February 28 and March 6, 2023.

The consumption pattern shows a clear seasonal trend for the baseload. The demand peaks at August 2022, followed by a sharp decline throughout October. A subsequent rise is observed during the winter months of December and January, before slightly decreasing again, approaching March 2023. The peak consumption seen in August is caused by a widespread use of air-conditioning during the warmer periods in Colorado, with a slight increase again in colder periods due to heating.

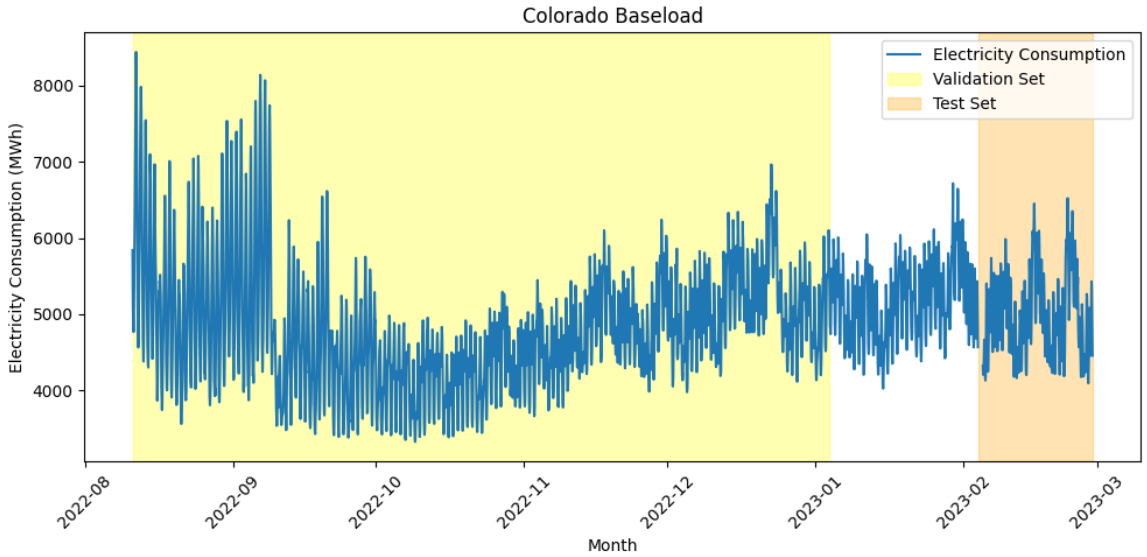

Figure 5.7: The baseload electricity consumption in Colorado

5.2 Data Preprocessing

This section outlines the preprocessing of the data. The first part of this section briefly covers how the three datasets have been preprocessed. Then, the applied features are presented along with contextual explanations of their relevance and functionality.

5.2.1 Colorado

As outlined in Section 5.1.1, the Colorado dataset provides EV charging sessions, including the start time, charging duration, and electricity consumption of individual charging sessions. However, the dataset presents several inconsistencies, such as changing date and time formats, empty values, duplicate entries, and irregular whitespace. Consequently, a comprehensive data cleaning process was performed to remove inconsistencies. Afterwards, the dataset is converted to an hourly granularity, which involves splitting the sessions into individual hours and fairly dividing the total electricity consumption of the charging session among the hours of the session. From the original 16 columns, including *State*, *Gasoline*

Savings, only three are used after converting the Colorado dataset, being: *Timestamp*, *Electricity Consumption*, and *Session Count*. The session count was not part of the original dataset, but was added during the conversion to help the models understand how many EVs are charging during each hour.

5.2.2 SDU

The SDU dataset has an inherent hourly granularity, where the only preprocessing step was to round the timestamp to the nearest hour. The dataset originally consists of 14 columns, including columns such as *Total Grid Load*, *Passed hour in the simulation*, and *Aggregated base load*. The study has selected 9 columns based on their relevance to the forecasting and classification task. These include *Timestamp*, *Aggregated charging load*, *Total number of EVs*, *Number of driving EVs*, *Number of charging EVs*, as well as the temporal features *Year*, *Month*, *Day*, and *Hour*. This subset was chosen to retain the key temporal and operational characteristics necessary for the models to effectively learn and predict charging demand patterns. The study has chosen to exclude the *Overload Duration [min]* column, since another threshold for overloads is applied, see Section 5.7.2.

5.2.3 Colorado Baseload Dataset

As mentioned in Section 5.1, this dataset serves as complementary data on the baseload. Section 5.1.3 showed the plot for the baseload, which contains missing data. To bypass this, the study chose to utilize a smaller region for the test set, skipping over a segment with missing data from January 4, 2023 to February 4, 2023. This results in a test set ranging from February 5, 2023 to February 28, 2023, from which the classification experiments will be conducted on.

5.2.4 Features

To enhance the predictive performance of the models, a plethora of features are added. These features provide additional context and structure that help the model learn more efficiently by highlighting relevant patterns and relationships within the dataset. The features of this study can be divided into: time-based features, cyclic features using cosine (cos) and sine (sin) transformation, and historical consumption-based features.

Timed-Based Features

Time-based features are designed to capture the cyclical and seasonal elements in the dataset. These features influence user behavior where adding these features can help the models better understand the underlying context and temporal patterns of electricity consumption, which can influence future values. Features such as day, month, weekend, year, and season

are commonly used to capture temporal patterns together with a holiday feature, granting the models context to the time perspective of the data. The holiday feature, especially, is useful to signify a probable change in user behavior, where events such as Christmas and Thanksgiving can greatly impact the user behavior. Additionally, a day/night indicator was included to differentiate between daytime and nighttime usage, with the division set at 06:00 and 18:00, respectively.
The time-based features used in this study are:

- Day of Week
- Month of Year
- Weekend
- Year
- Day/Night (Boolean)
- Is Holiday?
- Season of year

Cosine and Sine Transformed Cyclic Features

Most of the time-based features, such as hour, date, and month, are inherently cyclical. Representing these features as linear integers can mislead the models, since e.g. hour 23 and hour 0 are close time-wise but numerically distant. cos and sin can be applied to show the models the inherent cyclic nature in this type of data, capturing the data from both sides of the cycle.[46] The data is then transformed using trigonometric functions [47], resulting in the following cos and sin representations:

$$\sin\left(\frac{2\pi h}{P}\right),\ \cos\left(\frac{2\pi h}{P}\right)$$

where P is the period and h is the values being transformed, e.g, hour 2. The transformed features included are:

- Hour (Sin)
- Hour (Cos)
- Day of Week (Sin)
- Day of Week (Cos)
- Month (Sin)
- Month (Cos)

Weather Features

The weather has a notable impact on EV charging behavior, since the ambient temperature affects consumption patterns on real world data. The weather data used in this study comes from Meteostat [48], which is a weather and climate database for thousands of weather stations worldwide. The hourly temperature is incorporated as a feature exclusively in the Colorado dataset.

- Current Temperature °C

Historical Consumption Features

Historical consumption features are important in time series forecasting, as they provide context to the model about past behavior, potentially improving the model's capability to detect patterns, trends, and seasonality. This study divides the historical features into two categories: lag features and rolling averages. Lag features are used to give context about the electricity consumption at previous time steps. Rolling average calculates the average from the current time step and back to the previous time steps, which helps smooth out potential short-term fluctuations. Both datasets apply the historical features, which are the following:

- Lag
  - Electricity Consumption 1h
  - Electricity Consumption 6h
  - Electricity Consumption 12h
  - Electricity Consumption 24h
  - Electricity Consumption 1 Week
- Electricity Consumption Rolling Average 24 h

Selected Features for the Colorado and SDU dataset

The Colorado dataset applies all the previously mentioned features, since these added context about real-world consumption and temporal patterns. Contrarily, the SDU dataset utilizes far fewer features, since it is a synthetic dataset, and many features do not constructively add to the overall understanding of the dataset. This includes features such as: rolling average, 6h, 12h and 1w lag features. The rolling average and lag features negatively impacted the quality of forecasting, since the SDU dataset is sparse.
The final features for the datasets are as following:
Colorado Dataset (22 features in total):

- Original features: Session count

- Time-Based: Day of Week, Month, Weekend, Year, Hour, Day/Night, Holiday, Season
- Cyclical: Hour (Sin/Cos), Day of Week (Sin/Cos), Month (Sin/Cos)
- Weather: Temperature
- Historical: Lag (1h, 6h, 12h, 24h, 1w), Rolling Average (24h)

SDU Dataset (16 features in total):

- Original features: Aggregated charging load, total number of EVs, number of charging EVs, number of driving EVs
- Time-Based: Day of Week, Month, Year, Hour
- Cyclical: Hour (Sin/Cos), Day of Week (Sin/Cos), Month (Sin/Cos)
- Historical: Lag (1h, 24h)

5.3 Model Selection

The selection of models in this study is grounded in both their proven performance in prior literature, see Section 4 and their complementary strengths in capturing different aspects of the temporal patterns in EVCS electricity consumption forecasting. This also benefits the proposed ensemble architecture which can benefit from including a diverse set of models and utilizing bootstrap sampling with replacement i.e. multiple elements can be sampled more than once, seen in Section 2.4. This combination enables diverse base learners to contribute to an aggregated forecast with their unique perspectives on the time series.
By combining these models, the study aims to exploit the strengths of each category to create diversity for constructing an ensemble that, as a goal, can generate more reliable results than standalone models, both for forecasting EVCS consumption and in the context of predicting overload scenarios.
Unlike traditional bagging, which typically relies on a single model architecture to aggregate predictions, this ensemble incorporates diverse model architectures as base learners that each generate predictions. These predictions are aggregated through an averaging operation, ensuring each model contributes to the final prediction. To achieve this, the models included in the ensemble are:

LSTM and GRU

LSTM and GRU, see Section 2.3.1 and 2.3.2 are included due to their widespread usage in time series forecasting tasks, particularly in EVCS consumption forecasting. Their gating mechanisms are well-suited for capturing long-range dependencies in sequential data and have been proven to be stable at EVCS consumption forecasting.

Random Forest, Gradient Boosting, and AdaBoost

Random Forest, Gradient Boosting, and AdaBoost, see Section 2.4.1, 2.4.3, and 2.4.2, are selected for their robustness, interpretability, and efficiency. These models have demonstrated consistent strong performances in previous EVCS studies and serve as strong, complementary models to deep learning models.

xPatch and PatchMixer

xPatch and PatchMixer, see Section 4.2.2, 4.2.3, and 4.2.1, are more recent architectures designed to capture complex temporal dependencies with great performance over different types of datasets. Their inclusion introduces architectural diversity and allows for well-performing generalist predicting models to contribute to the ensemble. DPAD, although also attractive to implement, has a large computational cost which was not supported with the hardware setup, see Section 7.1.1, available for this study.

5.4 Technologies

This section presents the key technologies utilized in this study and outlines their respective roles and contributions.

5.4.1 PyTorch

PyTorch [49] is an open-source deep learning framework originally developed by Meta AI that provides a flexible and efficient platform for developing and training ML models. It is widely used in both academic research and the industry due to its intuitive API and strong community support. PyTorch provides a straightforward integration with CUDA, allowing operations to be offloaded to NVIDIA GPUs for high-performance computing (HPC), and GPU acceleration.
PyTorch is used in this study to implement deep learning models, manage the training pipeline, and perform tensor-based computations efficiently. It provides the foundation for implementing and training deep learning models such as LSTM, GRU, and xPatch with loss function implementations for MAE, MSE, and Huber loss.

5.4.2 PyTorch Lightning

PyTorch Lightning [50] is a high-level framework built on top of PyTorch, which simplifies the process of training, validating, and testing deep learning models. The key principle behind PyTorch Lightning is the modularization of code into reusable and standardized components. PyTorch Lightning handles the boilerplate code related to device placement, distributed training, precision settings, and more. This study applies PyTorch Lightning for

the aforementioned upsides and applies its broad options for processes related to training and optimizing deep learning models.

5.4.3 scikit-learn

Scikit-learn [51] is one of the most widely used open-source Machine Learning (ML) libraries in Python. The library provides simple and efficient tools for data preprocessing, model training, evaluation, and more. Scikit-learn enables consistent and easy to implement fit and predict methods, simplifying the process of constructing and evaluating ML pipelines. In this study, scikit-learn is primarily used to implement baseline models, including Random Forest, Gradient Boosting, and AdaBoost. Furthermore, scikit-learn supports data preprocessing, which this study uses to divide the time series data into training, validation, and test sets. Other important features in this library include the Min-Max Scaler, Max Abs Scaler, and resample function, enabling bootstrap sampling to be deployed.

5.4.4 Optuna

Optuna [52] is an open-source hyperparameter optimization framework designed to automate the process of tuning models. It uses an efficient search algorithm to find the best set of hyperparameters for a given model by minimizing or maximizing an objective function related to model performance on a validation set. Optuna also enables parallelization over multiple threads without modifying code and allows the user to define the search space in intuitive Python syntax. In this study, Optuna serves as a tool for hyperparameter tuning the models and tuning the ensemble for an optimal architecture.

5.5 Experimental Design

This section outlines the design of the experiments carried out to measure the ability of the models to forecast the EVCS consumption and capture overloads. The models are evaluated on both the Colorado and SDU dataset, where the Colorado dataset utilizes MAE, and the SDU dataset uses Huber Loss, further described in Section 5.6.3. The experiments are divided into two phases being: Model Tuning and Model Evaluation.

Model Tuning

- Hyperparameter tuning:
  - Performed independently under two different optimization objectives with and without bootstrap sampling:
    1. Tuning to minimize loss function
    2. Tuning to maximize recall and minimize loss function

- Architecture tuning:
  - Also conducted as two distinct experiments:
    1. Optimizing the configuration to minimize loss function
    2. Optimizing the configuration to maximize recall while minimizing loss function

Model Evaluation

- Forecasting EVCS electricity consumption using models trained without bootstrap sampling, to assess their accuracy and the effectiveness of the ensemble.
- Evaluating models trained without bootstrap sampling with the ensemble to measure each model's ability to capture overload events, with a focus on recall as the primary evaluation metric.

The model tuning phase consists of hyperparameter tuning models for the experiments conducted. The models trained with bootstrap sampling are used in architecture tuning to find an optimal ensemble configuration. The five best ensemble configurations will also be included to evaluate which models benefit the ensemble the most at each task.
For the EVCS electricity consumption forecasting experiment, the models are tuned to minimize MAE or Huber Loss, depending on the dataset examined. The experiment compares models trained without bootstrap sampling with the ensemble to evaluate their effectiveness in forecasting EVCS electricity consumption.
For the experiment of overload classification, the models are tuned to balance two objectives: maximizing recall to measure the effectiveness of overload capturing and minimizing MAE / Huber Loss to minimize the overall prediction error. This approach ensures the model not only detects overload events but also maintains accuracy in its forecasts.
Each of these phases is described in more detail in the following sections. Section 5.6 presents the hyperparameter and architecture tuning decisions made during the Model Tuning phase. Section 5.7 discusses the choices made for the Model Evaluation phase, which involves both the EVCS electricity consumption forecasting and overload classification.

5.6 Model Tuning

5.6.1 Hyperparameter Tuning

The hyperparameter tuning phase, as mentioned in Section 5.5, attempts to find the optimal hyperparameters for consumption forecasting and overload classification in both the Colorado and SDU datasets.

The hyperparameter space was explored using Bayesian optimization, which is implemented through Optuna's Tree-structured Parzen Estimator (TPE) sampler. [53] Bayesian optimization builds a probabilistic model of the objective function to make informed decisions on which hyperparameter to sample next. The TPE sampler specifically models the objective function as a probabilistic distribution and selects the hyperparameter configurations most likely to improve the performance, based on previous evaluations. This approach enables more promising regions of the hyperparameter space to be explored.
The hyperparameter selection is based on the standard configurations proposed in the original papers for each model. The default values of their proposed methods served as a reference for the construction of a tuning range by selecting values below and above the reference value. The search spaces for the models are detailed in Appendix A.1. A constant random seed of 42 was used to facilitate reproducibility for the experiments in the study.
To manage computational resources, the number of optimization trials was limited to 150 for tuning the models. The architecture tuning was set to run 100 trials, since it required substantially more time and resources, where DPAD was excluded due to its large computational costs, see Section 5.6.2. Furthermore, to optimize computational efficiency, 16-bit mixed precision was employed to reduce the numerical precision of operations, decreasing the memory usage and accelerating training speed, with a minimal trade-off in numerical accuracy. [54]
To fully utilize the available GPU resources on AAU's HPC platform, AI-LAB (see Section 7.1.1), a parallel training strategy was necessary. For this purpose, Distributed Data Parallel (DDP) was employed, which enables efficient training across multiple GPUs by distributing the workload across multiple workers [55]. In DDP, each GPU is assigned its own training process that performs both the forward and backward passes independently using a subset of the data. After computing the gradients locally, these processes synchronize with each other to average the gradients before updating the model parameters [56]. This ensures that all GPUs contribute to and benefit from the collective learning progress, significantly accelerating training while maintaining consistent model updates across devices.
The hyperparameter tuning may take more than 12 hours, which is the upper limit for tasks on AI-LAB. Therefore, the tuning is set to stop after 10.5 hours, so it stops before exceeding the 12-hour execution limit on AI-LAB. The 10.5-hour limit was selected to ensure sufficient time for Optuna to complete the ongoing trial, as the time constraint is only evaluated upon the completion of each trial, not during its execution. The study is then saved in a pickle file, which can be loaded to continue the study. This is accompanied by a safeguard of the objective, which allows failed trials to be skipped if, e.g. the trial attributes more GPU memory than possible resulting in the trial crashing safely.

5.6.2 Architectural Tuning

The objective of this process is to find an optimal configuration, where the base learners contribute to a robust aggregated forecasting model. To construct the ensemble, all possible subsets of the available individual models are generated, under the constraint that each model is used only once per subset and that each subset must contain at least three base learners to have sufficient diversity and ensemble strength. The model has $n = 7$ base learners, which means the total number of possible subsets are:

$$\begin{aligned} \text{Total subsets} &= \sum_{r=1}^{n} \binom{n}{r} = 2^n - 1 \\ &= 2^7 - 1 \\ &= 127 \end{aligned} \tag{5.1}$$

where r is the number of subsets.
By applying the constraint that only subsets of size $r \geq 3$ are considered, the number of valid subsets for architectural tuning becomes:

$$\begin{aligned} \text{Total subsets with } r \geq 3 &= 2^n - 1 - n - \frac{n(n-1)}{2} \\ &= 2^7 - 1 - 7 - \frac{7(7-1)}{2} \\ &= 99 \end{aligned} \tag{5.2}$$

Each model uses its corresponding hyperparameters found during the earlier hyperparameter tuning phase with bootstrap sampling. This ensures that each model contributes at its optimized performance level within the ensemble context of bootstrap sampling.

5.6.3 Choice of Loss Functions

During the experimentation of EVCS consumption forecasting and overload classification, MAE was found to work reliably in the Colorado data set, but showed sub-par results in the SDU data set. This was seen in that the models struggled to capture the temporal patterns due to the sparse nature of the SDU dataset. Furthermore, the models consistently underpredicted peak consumption values, which is particularly problematic, as these peaks are indicative of potential overload events.
This led the study towards exploring other options for the loss functions, namely MSE and Huber Loss. MSE was specifically chosen as an attempt to punish larger errors. This, though, affected the forecasts in the wrong direction, where MSE focused on minimizing the error at the lower regions at the expense of capturing peaks. To change that, the study attempted to fix this by using Hober Loss which contains both linear and exponential functions. Through experimentation with different δ values, it was found that $\delta = 0.25$ helped stabilize several

models in some experiments. However, some models continued to mainly fit the lower and middle consumption ranges.

5.7 Model Evaluation

5.7.1 EVCS Consumption Forecasting

To assess the impact of the ensemble on forecasting accuracy, the ensemble are compared against standalone models that were not trained using bootstrap sampling. The models are evaluated to gain a perspective into its accuracy and identify where the models may struggle to capture the time series dynamics. Each model is executed with their best performing hyperparameters.

5.7.2 Overload Classification

The process of classifying overloads differs substantially between the two datasets. This section clarifies the approach and specifications used to derive overload classifications for the Colorado dataset, followed by a description of how classification is handled for the SDU dataset.

Colorado

The overload classification consists of a baseload, seen in the Colorado consumption dataset in Section 5.2.3 and the predictions from the standalone models and the ensemble. However, these values are substantially far apart, with the baseload reaching 3415 to 6716 MWh in the test set, whereas the EVCS consumption remains between 0 to 82 kW in the Colorado dataset.
To accommodate this, the study will assume a transformer of 500kVA with a power factor of 1, meaning it can transmit 500kW of electricity, using the equation from Section 3.7.1. The time period for this transformer is set to 1 hour, resulting in the hourly electricity consumption of 500kWh.
The baseload is downscaled to reflect a localized scenario that a single transformer might encounter. In this context, downscaling refers to dividing the original aggregated state-level consumption by a constant factor, making the data representative of a smaller geographic and operational scope. Since the maximum value of the Colorado baseload dataset $X_{base,t}$ is 6716 MWh, it is downscaled by a factor of 13 to simulate a proportion of the grid served by a single transformer. This results in a new maximum value of:

$$max(\frac{X_{base,t}}{13 * 1000kWh}) = \frac{6716MWh}{13 * 1000kWh} \approx 516.62kWh$$

This adjusted value approaches the transformer capacity limit of 500 kWh, making the setup useful for detecting potential overloads, as shown in Figure 5.8.

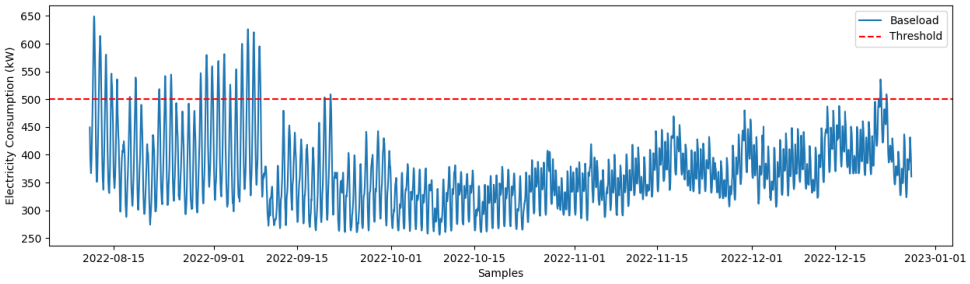

Figure 5.8: Colorado baseload with its downscaled baseload and transformer threshold

Meanwhile, EVCS consumption and model predictions are upscaled by a factor of 2 to further emphasize its contribution to total load. This results in the maximum upscaled consumption of:

$$max(X_{evcs,t}) \times 2 = 89.04kW \times 2 = 178.08kW$$

SDU

As mentioned in Section 5.1.2, the SDU dataset already contains information on overload with a defined grid overload threshold of 400 kW. However, the overload events almost exclusively occur as a rare event in the last year of simulation. Instead, this study pursues another region of the dataset, ranging from December 31, 2029 to December 31, 2030, mentioned in Section 5.5. To accommodate for that, the study has reduced the threshold to 250 kW to trigger more overload events, while retaining the original baseload values. An example of the baseload and EVCS consumption can be seen below:

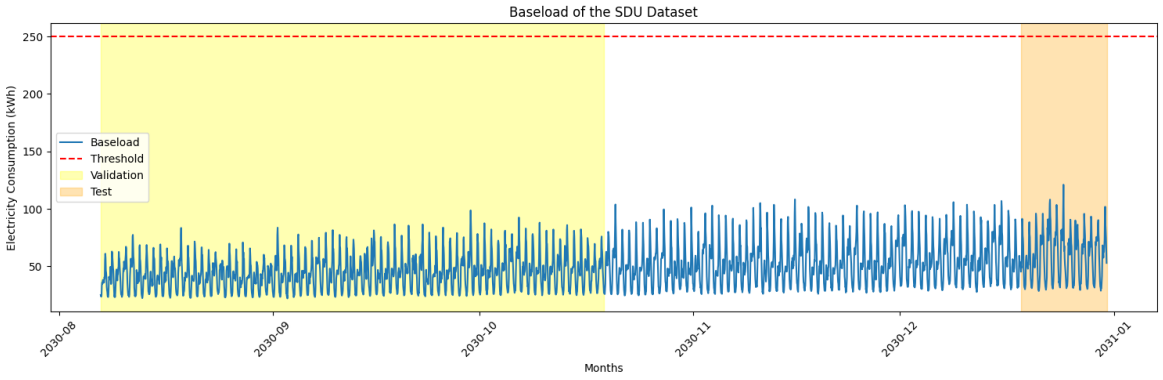

Figure 5.9: The baseload electricity consumption for the SDU dataset

6 Implementation

This chapter outlines the implementation details of the study. The first section presents the core components of the setup, with particular emphasis on the rolling window technique and the methodologies employed for model training and prediction. The second section elaborates on the structure of the experimental framework and the overall flow of the program.

6.1 Setup

6.1.1 Rolling Window Technique

The rolling window technique, see Figure 6.1, involves using a fixed-size window that slides across the time series, generating input sequences for the model to make predictions. In this study, a fixed forecast horizon $H = 24$ is used with a fixed lookback window of $T = 7 \times 24 = 168$. The $1 : 7$ ratio between the forecasting horizon and lookback window is used to fairly evaluate the models with the fixed-sized window. Following this, the stride $S = 24$ determines the step size that the window shifts after each prediction. In this setup, the stride is equivalent to the forecast horizon, ensuring each time step in the dataset is predicted only once. The rolling window technique is used in both the deep learning and ensemble learning setups, but requires different implementations.

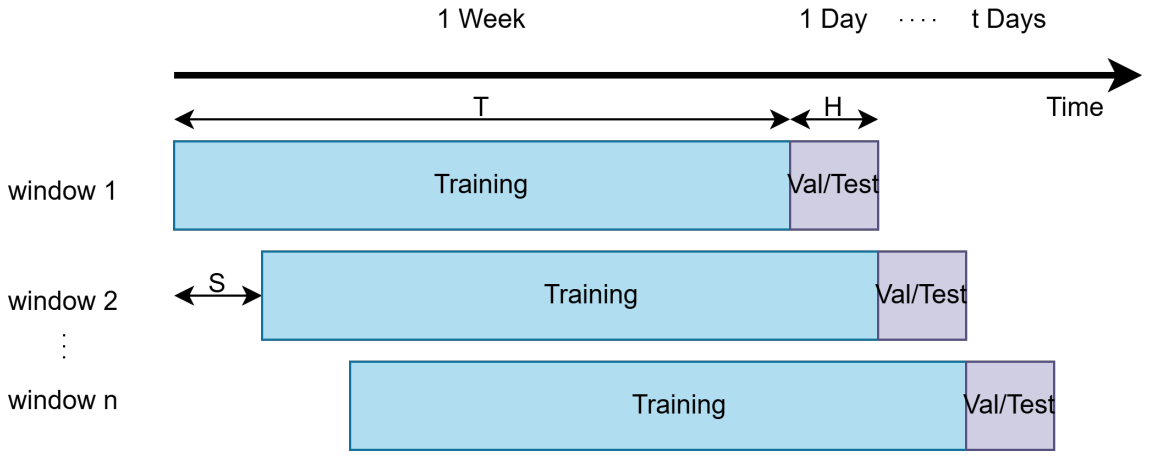

Figure 6.1: The rolling window technique

6.1.2 Deep Learning Setup

Each dataset is encapsulated within its own data module, which is a subclass of the PyTorch Lightning LightningDataModule. This module is responsible for setting up the time series for deep learning models by handling all data-related operations. This includes implementing data sampling, data preprocessing, data splitting, data loader setup, and the rolling window technique.
The data sampling process involves sampling the data to aggregate sessions and remove inconsistencies in the datasets, as mentioned in Section 5.2. Then, data preprocessing is implemented to extract the features described in Section 5.2.4.
Afterwards, the data is split into a 60% training set, 20% validation set, and 20% test set, as seen in Section 5.1. The data is then scaled with the Min-Max Scaler or MaxAbsScaler. This helps stabilize and accelerate the training process by preventing features with larger magnitudes from dominating the learning. Since larger values produce larger gradients, the model may prioritize minimizing larger errors over smaller ones and potentially ignore important information from smaller-scale temporal patterns. The scaling allows the pattern to spread more fairly across the input, leading to more balanced learning and better convergence.
The data loaders utilize both bootstrap sampling and the rolling window technique. The bootstrap sampling is applied via scikit-learn's resample function, which allows resampling with replacement, i.e. bootstrap sampling, to be implemented. Bootstrap sampling can be enabled or disabled based on the specific requirements of each experiment. To properly adapt this into the data loader, a custom class was implemented in the data loader, as seen in Figure 6.1. `dataset_size` describes the size of the dataset to be bootstrapped, whereas `random_state` determines which seed is used. `__iter__` implements the iterator to generate bootstrap samples, where the indices to be sampled are selected within the resample function, which the data loader uses to sample from.

```
1  class BootstrapSampler:
2      def __init__(self, dataset_size, random_state=None):
3          self.dataset_size = dataset_size
4          self.random_state = random_state
5
6      def __iter__(self):
7          indices = resample(range(self.dataset_size), replace=True,
8                             n_samples=self.dataset_size, random_state=self.random_state)
9          return iter(indices)
10
11     def __len__(self):
12         return self.dataset_size
```

Listing 6.1: Bootstrap sampling class

To implement the rolling window technique into the data loader, another custom class was created, see Listing 6.2. This class utilizes the rolling window specifications mentioned in Section 6.1.1 to properly handle the various rolling window configurations and properly shift the window by the stride length. In `__getitem__`, the start index is determined by taking the current index and multiplying it by the stride. Then, the `X_window`, i.e. the lookback window, and `y_target`, i.e. the forecast window, are found by slicing the dataset, and returned to the data loader.

```
class TimeSeriesDataset(Dataset):
  ...
  def __getitem__(self, index):
    start_idx = index * self.stride
    x_window = self.X[start_idx: start_idx + self.seq_len]
    y_target = self.y[start_idx + self.seq_len: start_idx + self.seq_len + self.pred_len]
    return x_window, y_target
```

Listing 6.2: Rolling window logic inside TimeSeriesDataset class

Training the deep learning models, see Listing 6.3, consists of wrapping the model in a `LightningModel` class with information such as the loss metric, optimizer, and learning rate of the optimizer. The trainer is then initialized to train the models with 16-bit mixed precision and the DDP-strategy, mentioned previously in Section 5.6.1.
For inference, deep learning models use only one device, since PyTorch Lightning does not manage the device placement as strictly during inference as during training, which can cause runtime errors and incorrect results, if the device is not set to 1.

```
model = LightningModel(model=model, criterion=criterion_map.get(args.criterion)(),
↪  optimizer=optimizer_map.get(args.optimizer), learning_rate=_hparams['learning_rate'])
    trainer = L.Trainer(max_epochs=_hparams['max_epochs'], log_every_n_steps=100,
    ↪  precision='16-mixed', callbacks=[pred_writer], enable_checkpointing=False,
    ↪  strategy='ddp_find_unused_parameters_true')
    trainer.fit(model, colmod)

    trainer = L.Trainer(max_epochs=_hparams['max_epochs'], log_every_n_steps=100,
    ↪  precision='16-mixed', callbacks=[pred_writer], enable_checkpointing=False,
    ↪  devices=1)
    trainer.predict(model, colmod)

```

Listing 6.3: Trainer Setup where mixed precision and ddp strategy is deployed

6.1.3 Ensemble Learning Setup

The ensemble learning model's data are also managed within the LightningDataModule to collect all data management in one place. The setup is made with a function called `sklearn_setup`, which is a separate setup for bootstrap sampling and rolling window technique. As seen in Listing 6.4, the training set has the optionality of being implemented with or without bootstrap sampling, as required for the experiments. Then, the rolling window technique is applied with the built-in Python function called Parallel, which uses multiple workers from all available CPU cores to generate the windows for faster processing speed. When this process is finished, every lookback window and forecast horizon is returned in separate NumPy arrays.

```
1   def sklearn_setup(self, set_name: str = "train"):
2   if set_name == "train":
3     if args.individual == 'False':
4       X, y = resample(self.X_train, self.y_train, replace=True,
          ↪  n_samples=len(self.X_train), random_state=SEED)
5     else:
6       X, y = self.X_train, self.y_train
7   elif set_name == "val":
8       X, y = self.X_val, self.y_val
9   elif set_name == "test":
10      X, y = self.X_test, self.y_test
11
12  max_start = len(X) - (seq_len + pred_len) + 1
13
14  results = Parallel(n_jobs=-1)(
15      delayed(process_window)(i, X, y, seq_len, pred_len) for i in range(0, max_start,
          ↪  stride)
16  )
17
18  X_window, y_target = zip(*results)
19  return np.array(X_window), np.array(y_target)
```

Listing 6.4: The scikit-learn setup for ensemble learning models, where max_start is the last possible start point for a window

Training and predicting with ensemble learning models remains relatively straightforward with the scikit-learn library, as seen in Listing 6.5. The `sklearn_setup` function is used to sample both the training and validation/test set, depending on the experiment. The model is then fit to the data, whereas a prediction is made for the validation/test set.

```
X_train, y_train = colmod.sklearn_setup("train")
X_val, y_val = colmod.sklearn_setup("val")

model.fit(X_train, y_train)
y_pred = model.predict(X_val)
```

Listing 6.5: Training and predicting with ensemble learning models

6.2 Hyperparameter Tuning

The hyperparameter tuning is made for two separate objectives, as mentioned in Section 5.6.1, with one being exclusive for MAE, and the other with a multi-objective setup for the recall score and MAE.

Table 6.1 shows the shared hyperparameters among the models. Here, the batch size is tuned because optimal performance can vary across models. Some models may benefit from more frequent gradient updates with smaller batches, while others may perform better with larger batches and more stable updates. For the DPAD model, a reduced batch size is used due to memory constraints because of the model's size.

The learning rate is selected from a logarithmic scale ranging from 10^{-2} to 10^{-4}, which is a commonly effective range for the Adam optimizer. To explore this space efficiently, log-uniform sampling is applied using Optuna, allowing values to be sampled evenly across several orders of magnitude. This approach ensures a fair search in logarithmic space, where small changes in learning rate can have a significant impact on training dynamics.

The epochs are tuned exclusively for the deep learning models, between 1000 and 2000 epochs, to make sure that the deep learning models can train for a stable amount of time while keeping the runtime somewhat low.

Hyperparameter	Colorado	SDU
Batch size	32 to 128, step 16	32 to 256, step 16
Learning rate	1e-4 to 1e-2, logarithmic	1e-4 to 1e-2, logarithmic
Max Epochs	1000 to 2000, step 100	1000 to 2000, step 100

Table 6.1: The hyperparameters used for every model. Step describes the step size that can be explored in the search space

The hyperparameter tuning process, illustrated in Figure 6.2, follows a structured pipeline to find optimal hyperparameters for each model.

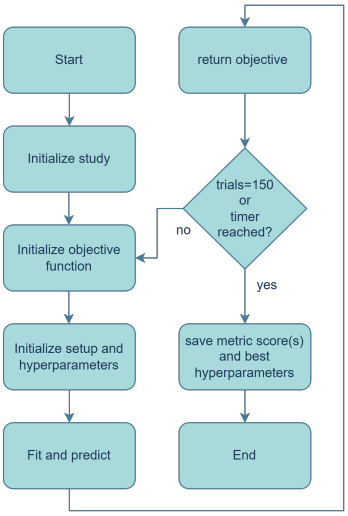

Figure 6.2: A flow chart of the hyperparameter tuning process

Initialize study

The tuning process begins by specifying key arguments, including the dataset used, forecast horizon, model used, and whether mixed precision is enabled. These parameters are passed to the study initialization, with the number of trials set to 150.

Initialize objective

It attempts to load an existing study to resume from previous trials, and if none is found, a new study is created.
Next, the objective function is defined to minimize the loss function. The study is configured to run for up to 150 trials or until a timeout of 10.5 hours is reached, ensuring that any ongoing trial is allowed to complete before termination, assuming a trial does not take more than 1.5 hours. The objective function is also safeguarded by a try except to allow the study to ignore trials that have failed and continue running, until 150 trials are reached.

6.2.1 Initialize Setup and Hyperparameters

The setup follows the process described in Section 6.1, where the window setup is initialized with the rolling window setup, using the fixed lookback window, forecast horizon, and stride. Afterward, Optuna suggests hyperparameters based on the selected model, previously explained in Section 5.6.1.

Fit and predict

In this step, the models are fit to the training set to create a prediction on the validation set, which computes the MAE score or recall score and MAE score, depending on the experiment chosen.

Return objective

The metric score(s) are returned through the objective, where the tuning will terminate if the maximum number of trials or maximum runtime has been reached. Otherwise, it will start a new trial with a new configuration of hyperparameters.

Save metric score(s) and best hyperparameters

The metric score(s) are then saved in a local CSV file with information on the model name, the trial reached, the best validation loss, and the best hyperparameters.

6.3 Architectural Tuning

The architecture tuning involves finding an optimal configuration of the models to aggregate the models. The study examines both objectives of minimizing MAE and the multi-objective of maximizing recall and minimizing MAE as described in Section 5.6.2. A flow chart of the architectural tuning can be seen in Figure 6.3.

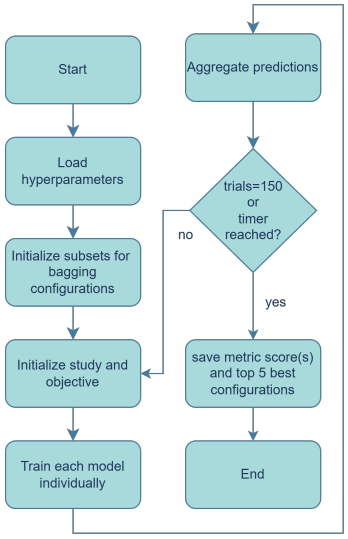

Figure 6.3: Flow chart of the architecture tuning process

Load hyperparameters

The first step loads the hyperparameters found in the hyperparameter tuning phase, where the base learner models are trained with bootstrap sampling.

Initialize subsets for bagging configurations

Optuna is used to tune different ensemble configurations by evaluating every possible subset of the available models trained with bootstrap sampling. Each subset, containing at least three unique base learners, is initialized and suggested as a categorical option. To ensure diversity, each model appears only once within a given subset, resulting in ensembles composed of distinct learners, as previously mentioned in Section 5.6.2.

Initialize study and objective

This study is then set up to attempt to optimize the metric score(s) by experimenting with different model configurations.

Train each model individually

Each trial selects a specific subset of model configurations, which are then run using Optuna. Once a subset is chosen, the corresponding models are initialized with their tuned hyperparameters and prepared for the prediction phase. The models are trained individually, where each of the predictions are saved to prepare for the prediction aggregation.

Aggregate predictions

After the individual models have made their predictions, the results are aggregated by computing the average of all the predictions of the base learners, which is then evaluated using the objective function(s).
Following this, the metric score(s) are calculated. If the study reaches the limit of 100 trials or exceeds the 10.5-hour time span, it will save the metric score(s).

Save metric score(s) and top 5 best configurations

The metric score(s) are lastly saved in a CSV file with the top 5 best-performing model configurations to get a better understanding on which models are typically featured in the best performing configurations.

6.4 EVCS Consumption Forecasting

The process of evaluating individual models and the ensemble is done using the process seen in Figure 6.4.

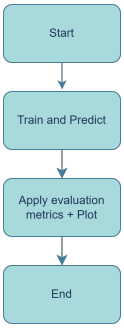

Figure 6.4: Flow chart of the EVCS consumption forecasting process

Train and predict

The process starts by picking either a model or the ensemble. Once the selection is made, the model is trained using its best hyperparameters. After training, it generates predictions for the test set.

Apply evaluating metrics + plot

Once the predictions are ready, a plot is created that shows the predictions and actual values. This plot is saved for later review. Then, MAE and MSE are calculated and saved in a CSV file.

6.5 Overload Classification

Finally, the overload classification process evaluates how effectively both the individually tuned models without bootstrapping and the ensemble detect overloads. The classification metrics described in Section 3.7.2 are applied, with recall serving as the primary metric to measure the models' ability to capture overload events together with MAE to capture its general forecasting ability.

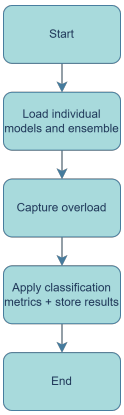

Figure 6.5: Flow chart of the classification process

Load standalone models and ensemble

Either a standalone model or the ensemble is loaded with its best-tuned hyperparameters.

Capture overload

To capture the overload, several processes are needed. Firstly, the model should predict with various setups for classification, including the scaling and downscaling for the Colorado and SDU dataset, as mentioned in Section 5.7.2. The overload is then classified using a region of the test set, as explained in Section 5.1.3 and 5.7.2.

Apply classification metrics and store results

The classification metrics are finally applied to evaluate the model's accuracy, precision, recall, and other metrics, depending on the experiment. Finally, the results are stored in a CSV with these metrics.

7 — Experiments and Results

This chapter presents the experiments conducted in this study. The chapter starts with a brief overview of the hardware specifications used during the experiments. Following that, the analysis of each experiment is presented in two subsections following the experiment design described in Section 5.5.

7.1 Experiment Specifications

7.1.1 AI-LAB

AI-LAB [57] is a HPC platform available exclusively for students at Aalborg University, designed to support computationally intensive workloads. It employs Slurm for job scheduling and task management, where Singularity containers are used to encapsulate the packages and dependencies.
The platform is powered by AMD EPYC 7543 processors with 32 cores and 128 CPU threads. It has 11 computing nodes in total, each equipped with eight NVIDIA L4 GPUs with 24 GB of RAM. These nodes are available for students at Aalborg University, where only one node is allowed to use during peak hours. It should be clarified that no nodes are guaranteed for us to be used, sometimes resulting in long queue times. During off hours, more nodes may be available, but this is limited by the traffic on AI-LAB.

7.2 Model Tuning

This section presents the results of hyperparameter tuning performed on the Colorado and SDU datasets with varying objective functions. The goal of this section is to provide optimal hyperparameters with the set objective, which is applicable to the main experiments for EVCS consumption forecasting and overload classification.

7.2.1 Tuning Results for Minimizing MAE for the Colorado Dataset

Table 7.1 shows the MAE scores for the non-bootstrap sampled and bootstrap sampled models on the Colorado dataset. Across all models, PatchMixer achieved the lowest MAE score in both scenarios.
Overall, the results show that deep learning models generally outperform ensemble learning models, where AdaBoost has a significantly higher MAE score than PatchMixer with 10.53 and 6.50, respectively. This may indicate that ensemble learning models have a greater

difficulty capturing the temporal patterns of the Colorado dataset. Furthermore, it is also seen that bootstrap sampling greatly reduces the MAE score of ensemble models.

Model	**Non-Bootstrap Sampling (MAE)**	**Bootstrap Sampling (MAE)**
AdaBoost	10.53	14.11
GRU	6.69	6.75
Gradient Boosting	8.20	14.35
LSTM	6.64	8.86
PatchMixer	**6.50**	**6.58**
Random Forest	9.76	14.54
xPatch	6.53	6.72

Table 7.1: MAE scores for models with and without bootstrap sampling on the Colorado dataset

Figure 7.1 shows that PatchMixer generally follows the trends but struggles to fully capture the upper regions. This limitation is even more pronounced in Figure 7.2, where AdaBoost focuses exclusively on the lower regions to minimize MAE, neglecting higher-value areas. A similar behavior is observed in its bootstrap-sampled version (Figure 7.3), which concentrates mainly on the middle regions. In contrast, Figure 7.4 demonstrates that GRU captures a broader range of values but still fails to capture peak values accurately.

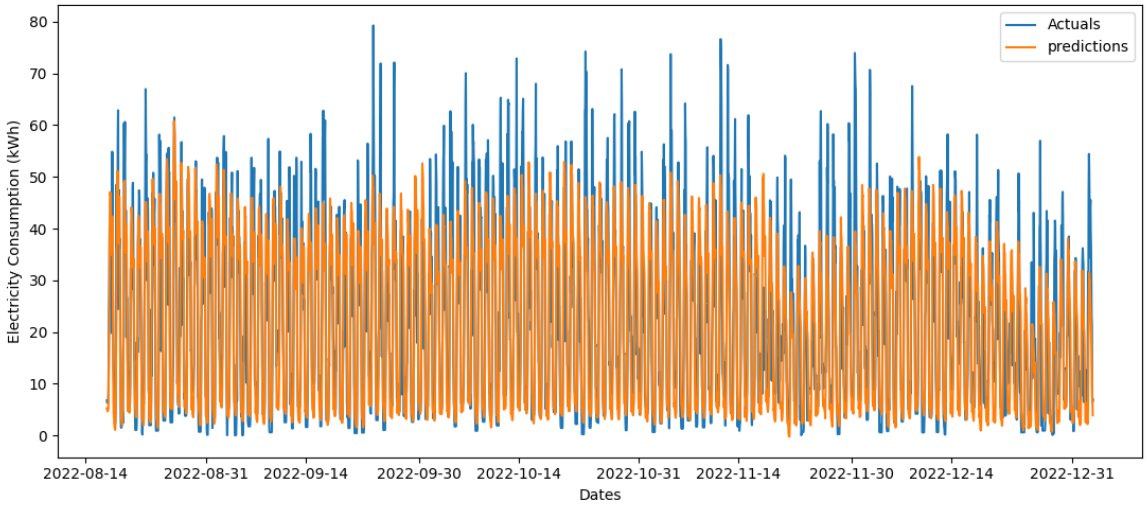

Figure 7.1: Non-bootstrap sampled PatchMixer for minimizing MAE

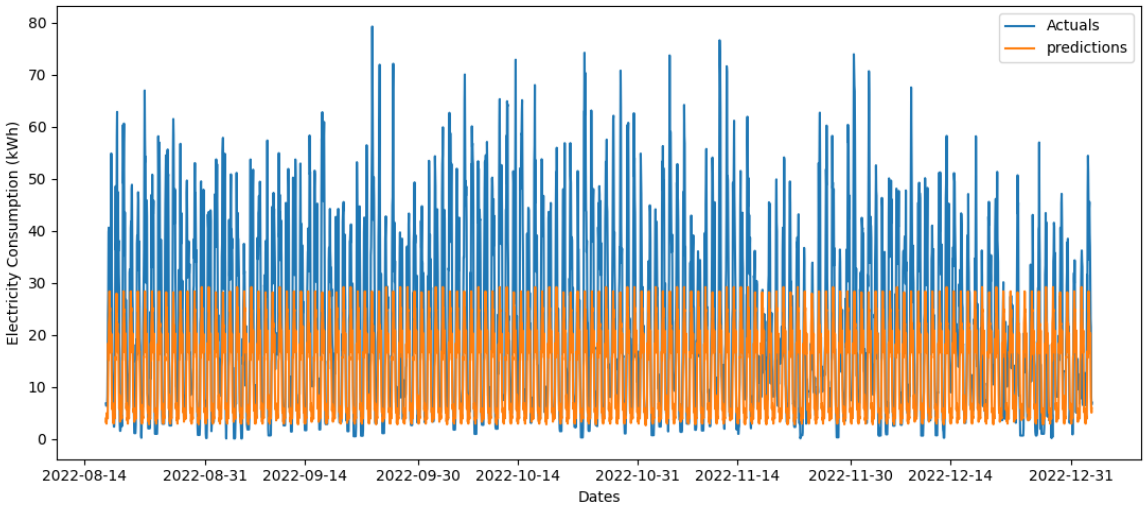

Figure 7.2: Non-bootstrap sampled AdaBoost for minimizing MAE

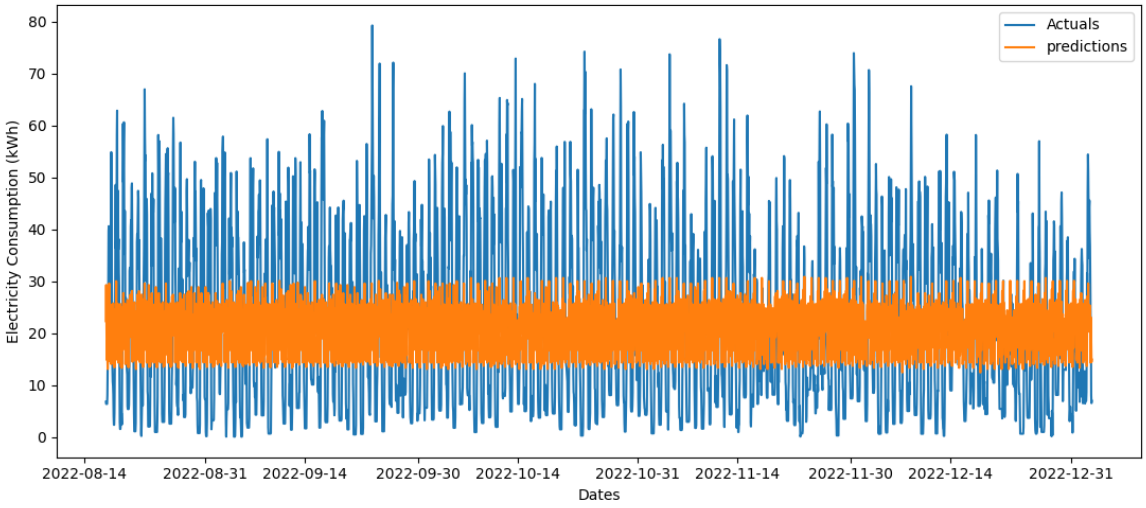

Figure 7.3: Bootstrap sampled AdaBoost for minimizing MAE

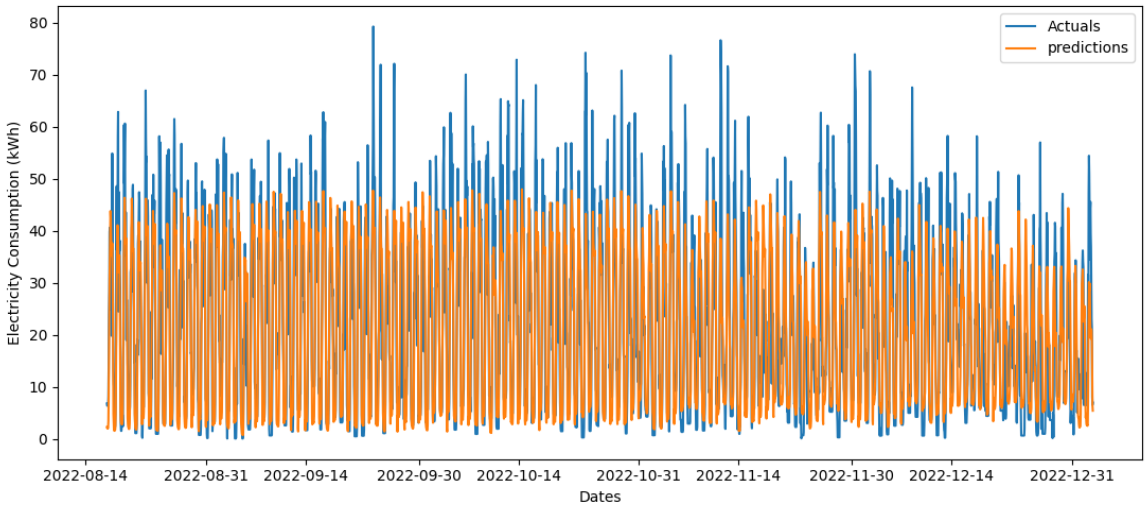

Figure 7.4: Bootstrap sampled GRU for minimizing MAE

7.2.2 Tuning Results for Minimizing Huber Loss for the SDU Dataset

As seen in Table 7.2, xPatch significantly outperformed every model with a Huber loss score of 2.59 for non-bootstrap sampling, where only GRU has a competitive Huber loss of 2.68. It is observed that the ensemble learning models generally perform the worst with 3.97 and 4.23 for AdaBoost and Random Forest, respectively. Gradient Boosting, however, achieved a Huber loss score comparable to LSTM and PatchMixer, different from the previous experiment, see Section 7.2.1. Overall, the ensemble models are still substantially worse when bootstrap sampling is applied.

Model	**Non-Bootstrap Sampling (Huber Loss)**	**Bootstrap Sampling (Huber Loss)**	**Non-Bootstrap Sampling (MAE)**	**Bootstrap Sampling (MAE)**
AdaBoost	3.97	48.21	16.00	48.71
GRU	2.68	10.60	10.81	10.93
Gradient Boosting	3.32	45.03	13.39	45.53
LSTM	3.16	12.46	12.71	12.78
PatchMixer	3.40	12.90	13.73	13.38
Random Forest	4.23	53.01	17.01	53.50
xPatch	**2.59**	**10.32**	**10.46**	**10.67**

Table 7.2: Huber Loss and MAE scores for SDU dataset

As shown in Figure 7.5, 7.6, 7.7, and 7.8, the forecast outputs display a repetitive and

smoothed pattern with minimal variation over time. Their limited responsiveness to temporal fluctuations implies an inability to capture the dynamic nature of the underlying time series, especially in the upper regions.

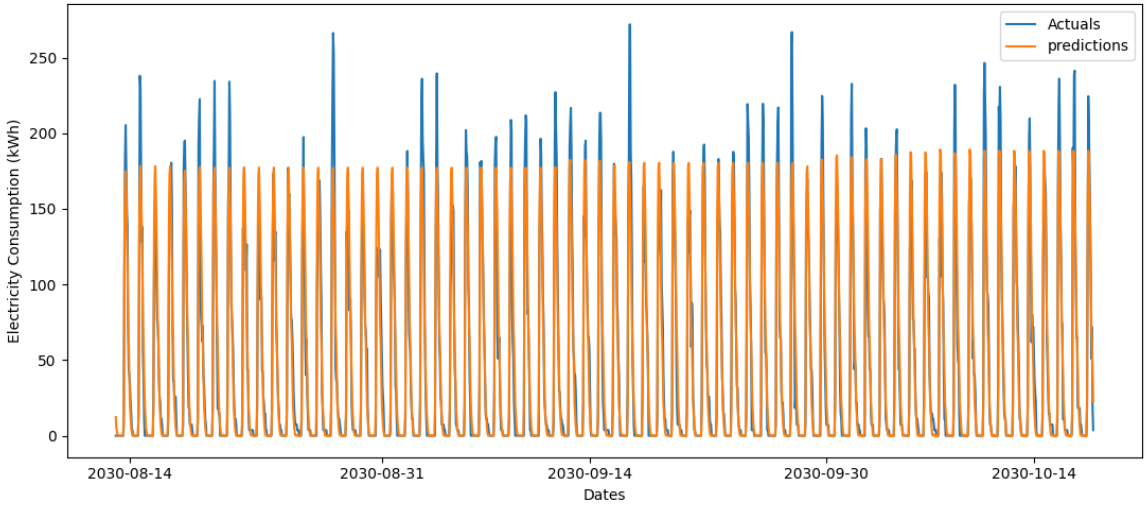

Figure 7.5: Non-Bootstrap sampled xPatch for minimizing Huber Loss

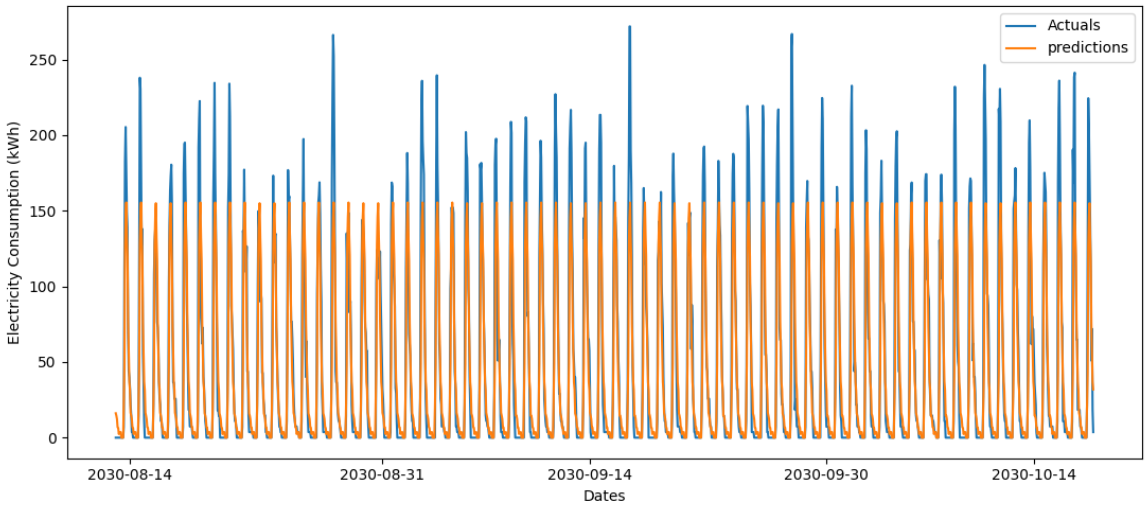

Figure 7.6: Non-Bootstrap sampled AdaBoost for minimizing Huber Loss

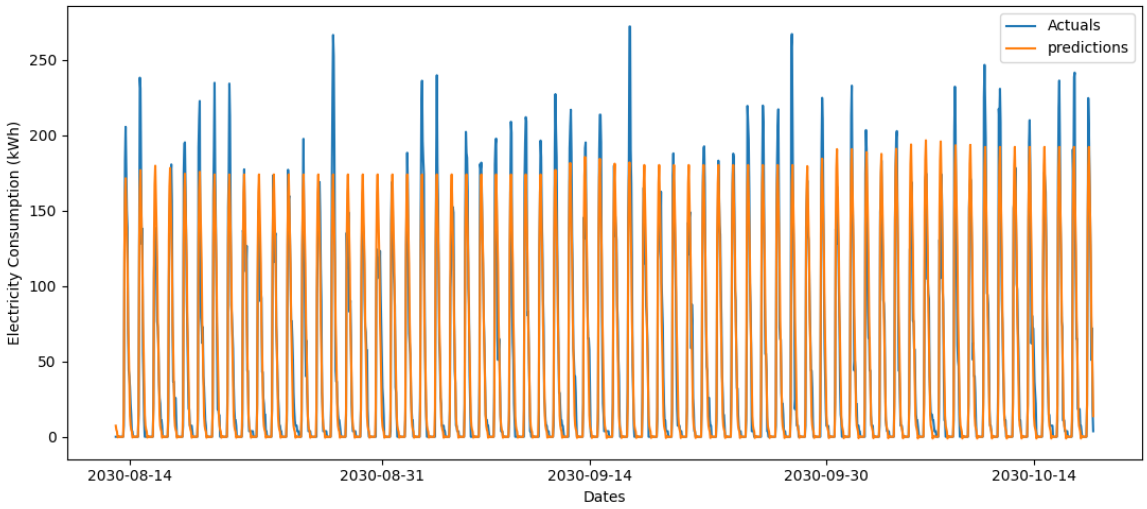

Figure 7.7: Bootstrap sampled xPatch for minimizing Huber Loss

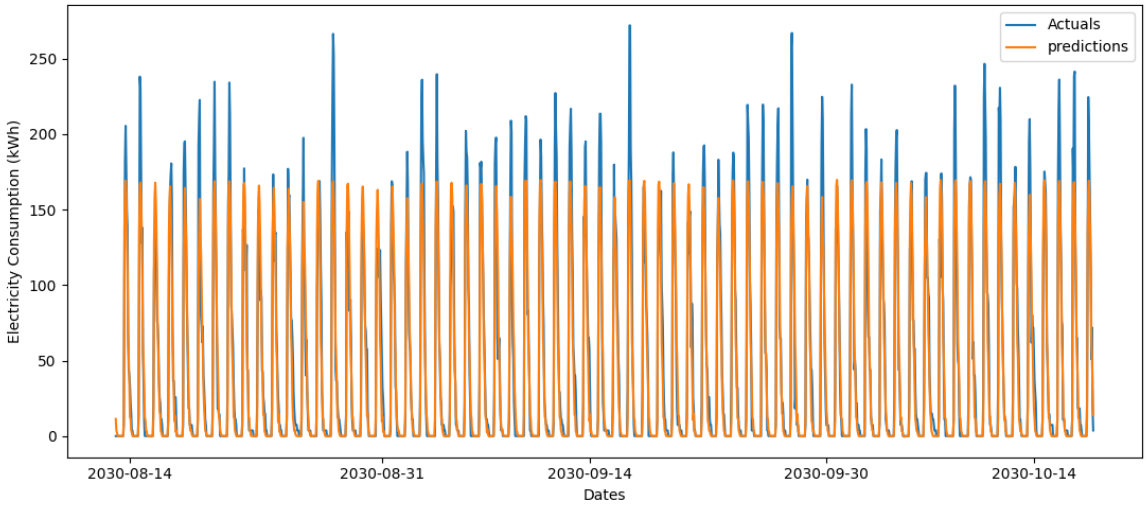

Figure 7.8: Bootstrap sampled GRU for minimizing Huber Loss

7.2.3 Tuning Results for Maximizing Recall and minimizing MAE for the Colorado Dataset

Table 7.3 presents the results of the multi-objective tuning process for the Colorado dataset. This shows that PatchMixer achieves the highest recall score of 0.91, whereas GRU and xPatch come close with 0.81 and 0.82, respectively. Looking at their MAE scores, it is seen that PatchMixer has a relatively higher MAE score of 23.05, capturing more overload events, whereas GRU and xPatch have lower MAE scores of 13.97 and 14.23, respectively. This indicates that PatchMixer may be overshooting on its predictions to get a higher recall score,

at the expense of the MAE score. For ensemble learning models, AdaBoost and Random Forest capture slightly above half of the overload events, indicating weaker performance on the recall metric.

	Non-bootstrap sampling		Bootstrap sampling	
Model	Recall	MAE	Recall	MAE
AdaBoost	0.54	21.11	0.70	30.16
Gradient Boosting	0.72	28.02	0.81	38.04
GRU	0.81	**13.97**	0.81	**13.95**
LSTM	0.73	14.13	0.76	14.28
PatchMixer	**0.91**	23.05	0.83	16.55
Random Forest	0.57	19.74	0.68	32.04
xPatch	0.82	14.23	**0.86**	16.99

Table 7.3: MAE and Recall scores for models with and without bootstrap sampling on the Colorado dataset

Table 7.9, 7.10, and 7.11 shows three substantially good models for GRU without bootstrap sampling and Bootstrap sampled xPatch and Gradient Boosting. Both GRU and xPatch exhibit good capturing in most regions, with slight misses on the upper regions. Gradient Boosting focus almost solely on the upper regions, hence its higher MAE score.

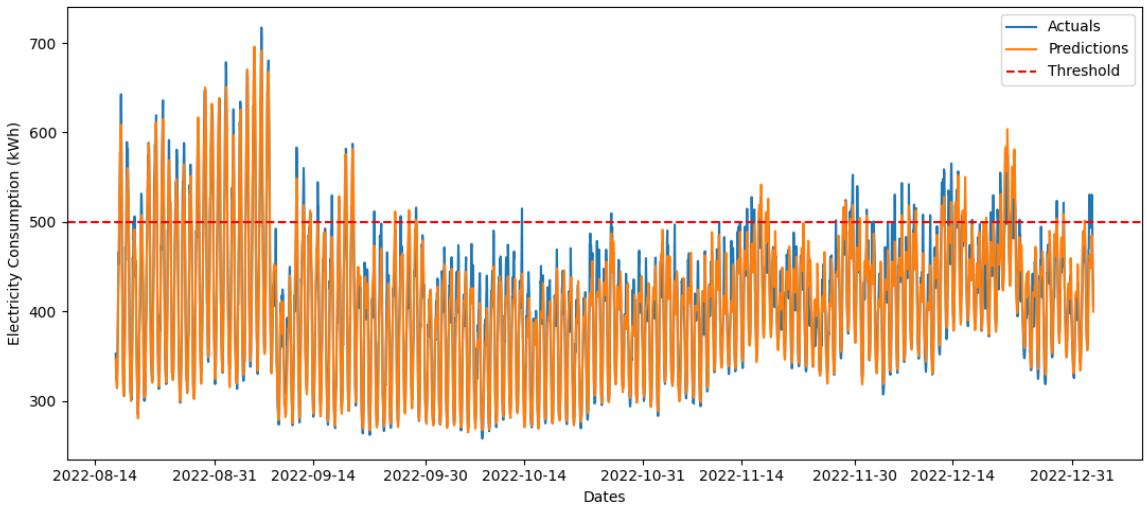

Figure 7.9: Non-Bootstrap sampled GRU for maximizing Recall and minimizing MAE

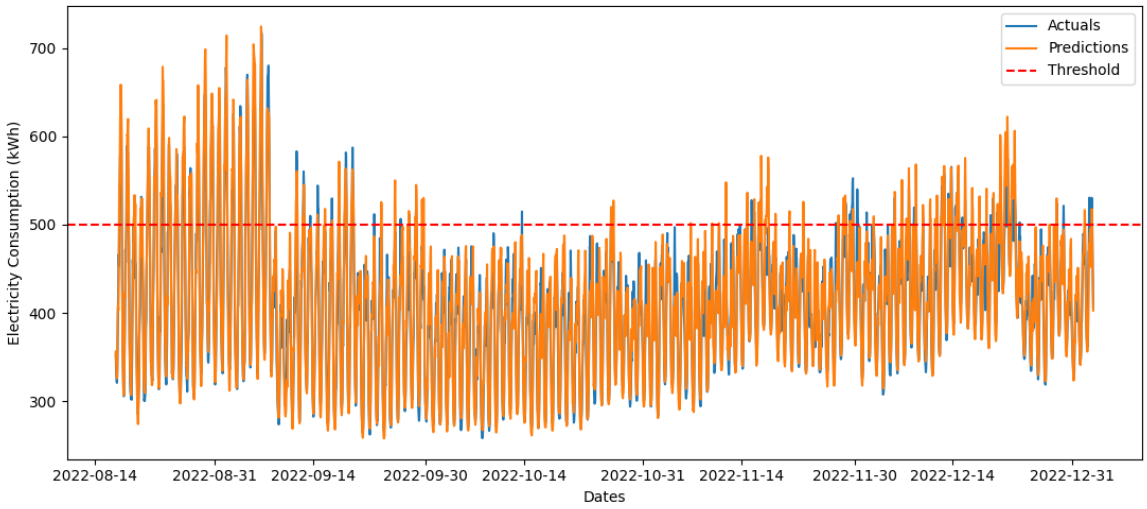

Figure 7.10: Bootstrap sampled xPatch for maximizing Recall and minimizing MAE

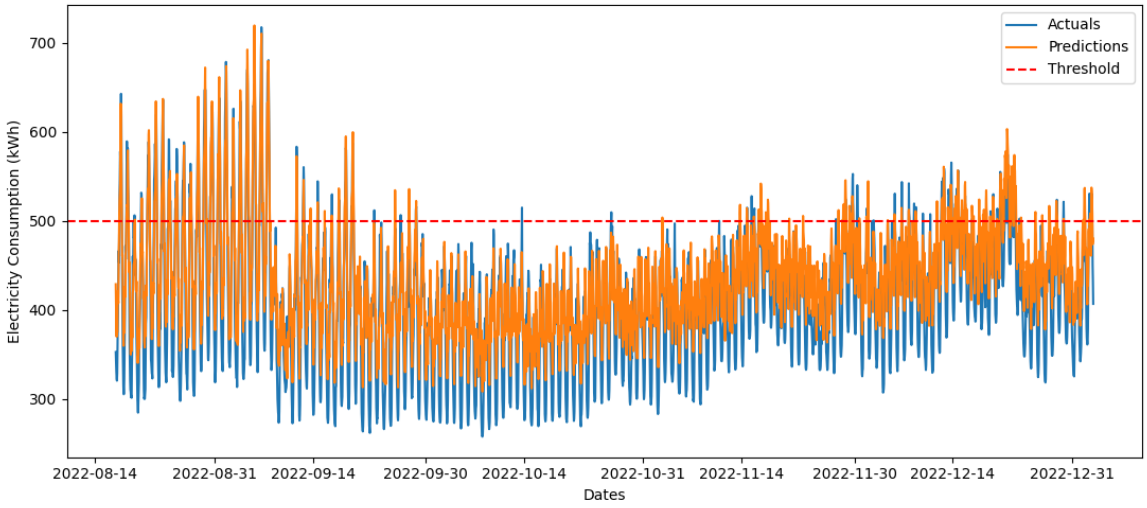

Figure 7.11: Bootstrap sampled Gradient Boosting for maximizing Recall and minimizing MAE

7.2.4 Tuning Results for Maximizing Recall and minimizing Huber Loss for the SDU Dataset

Table 7.4 presents the tuning results for overload classification on the SDU dataset. Both PatchMixer and xPatch successfully detect all or nearly all overload events. However, a notable difference between these two non-bootstrap sampling methods lies in their Huber loss scores. xPatch retains a stable Huber loss score of 3.30, while PatchMixer has significantly increased this score to 8.49, which was 3.40 in Table 7.2. GRU and Gradient Boosting remain around the 0.5 recall score mark, but it is seen that GRU has a significantly lower Huber

loss score, signifying that GRU may fit to lower regions, which only partially captures the overload events. More extreme cases of this can be seen for AdaBoost, LSTM, and Random Forest, which inadequately modeled overload events likely for the same reason. It was also observed that especially Gradient Boosting's Huber loss score got massively reduced, due to bootstrap sampling.

	Non-bootstrap sampling		Bootstrap sampling	
Model	Recall	Huber Loss	Recall	Huber Loss
AdaBoost	0.05	3.95	0.0	10.84
Gradient Boosting	0.5	4.81	0.61	20.17
GRU	0.33	**2.56**	0.22	**2.34**
LSTM	0.0	3.00	0.0	2.98
PatchMixer	**1.0**	8.49	**1.0**	10.94
Random Forest	0.0	3.92	0.0	12.75
xPatch	0.94	3.30	0.89	3.09

Table 7.4: Recall and Huber Loss scores for the SDU dataset

Figure 7.5 and 7.13 shows the best performances for non-bootstrap sampling. Here, it is observed that PatchMixer overshoots more extensively than xPatch, which is closer aligned to the actual values of the time series. Figure 7.14 and 7.15 shows GRU, with the lowest Huber loss score, and Gradient Boosting with the highest Huber loss. Here, it is seen that GRU undershoots the predictions slightly in most cases. Gradient Boosting, on the other hand, significantly overshoots many different patterns from the actual values shown, indicating that bootstrap sampling may have reduced its effectiveness in capturing the temporal patterns.

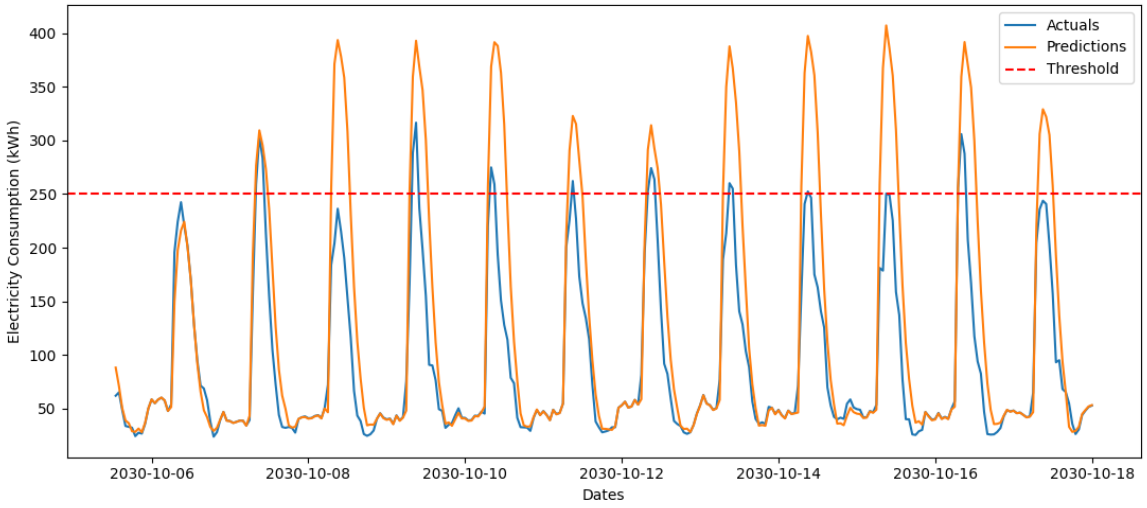

Figure 7.12: Non-Bootstrap sampled PatchMixer for minimizing Huber Loss and maximizing Recall

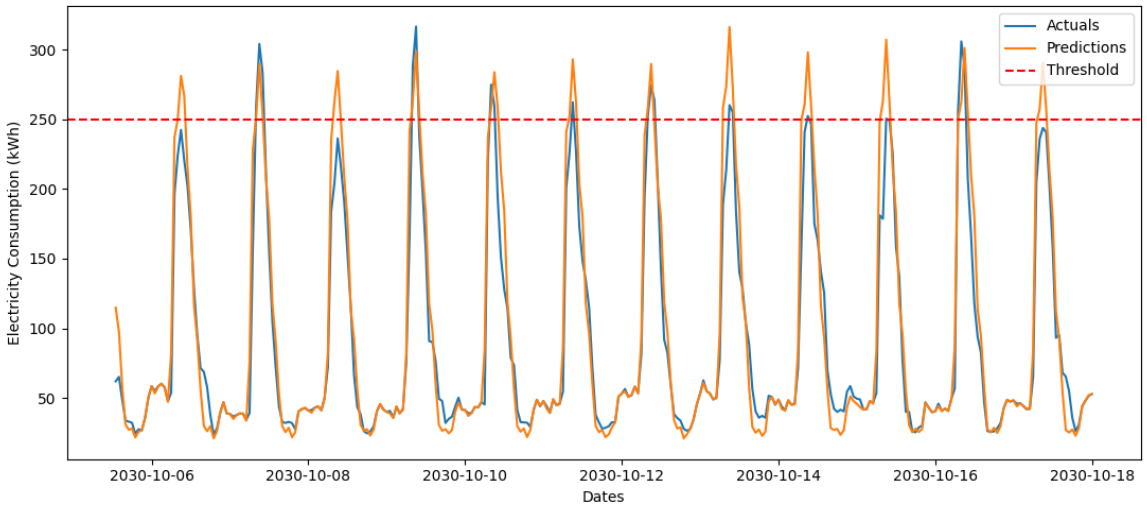

Figure 7.13: Non-Bootstrap sampled xPatch for minimizing Huber Loss and maximizing Recall

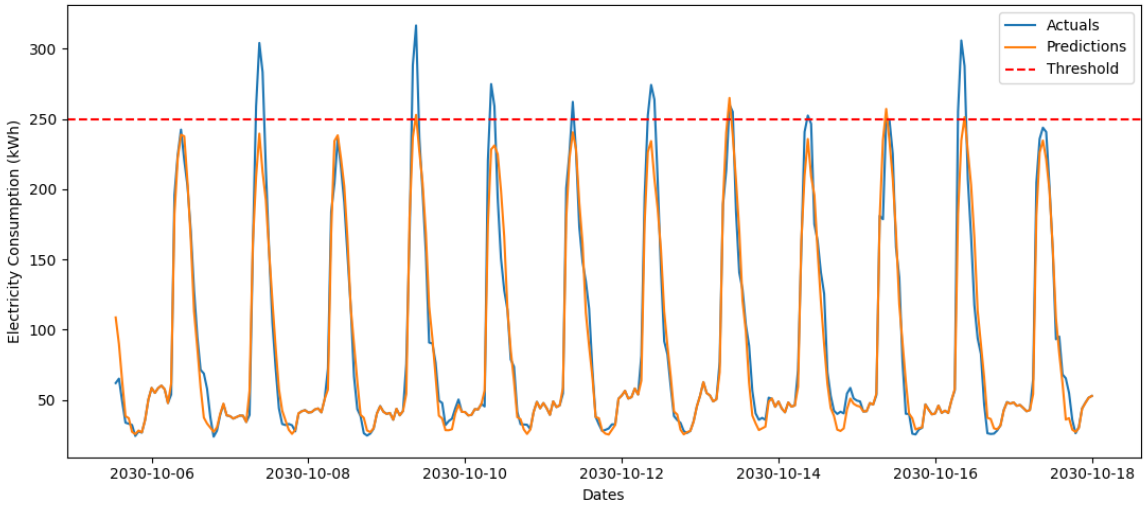

Figure 7.14: Bootstrap sampled GRU for minimizing Huber Loss and maximizing Recall

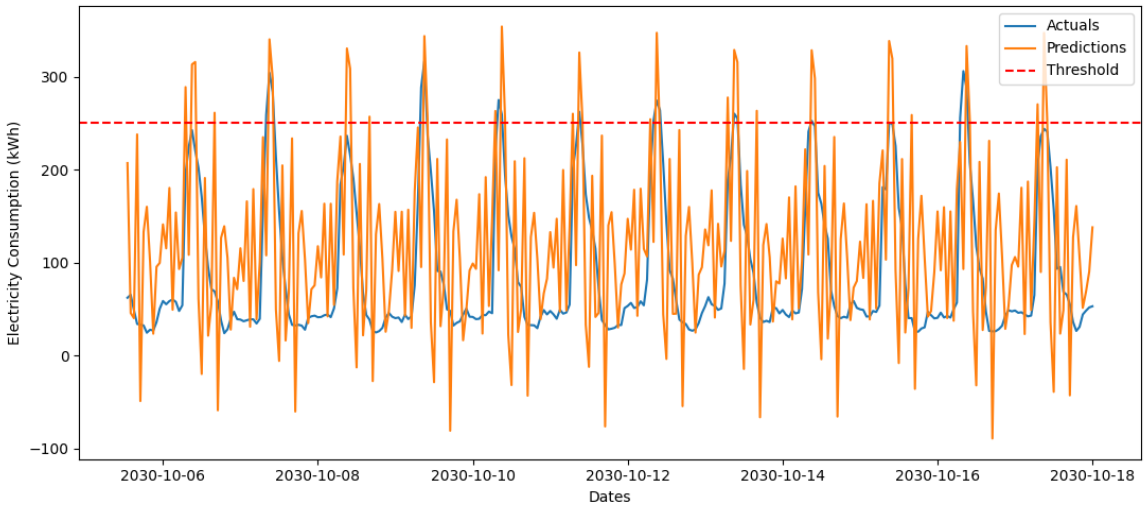

Figure 7.15: Bootstrap sampled Gradient Boosting for minimizing Huber Loss and maximizing Recall

7.3 Architecture Tuning

The architecture tuning experiment investigates the optimal ensemble configuration for each model on each dataset, aiming to identify the top 5 configurations that yield the lowest MAE / Huber loss score for the first experiment, or with the highest recall score and lowest MAE / Huber loss score.

7.3.1 Optimizing the Architecture to Minimize MAE for the Colorado Dataset

The best performing configuration combines Random Forest, Gradient Boosting, GRU, PatchMixer and xPatch, as seen in Table 7.5. This configuration achieves the lowest MAE and MSE score of 6.44 and 78.67, respectively, indicating a slightly stronger performance than the best performing standalone model: PatchMixer with an MAE score of 6.50, see Section 7.2.1. This result looks promising, since the strength of combining diverse model predictions can slightly improve the best standalone model. Looking at the configurations, it is seen that removing Random Forest from the top-performing configuration slightly worsens both MAE and MSE, suggesting it contributes positively to the forecast despite a weaker standalone performance. Furthermore, it seems like the smaller configurations of 3-4 models also tend to perform well.

Configuration	MAE	MSE
Random Forest, Gradient Boosting, GRU, PatchMixer, xPatch	**6.44**	**78.67**
Gradient Boosting, GRU, PatchMixer, xPatch	6.58	84.01
Random Forest, LSTM, GRU, PatchMixer	6.63	82.93
LSTM, PatchMixer, xPatch	6.75	89.41
LSTM, GRU, PatchMixer	6.79	89.27

Table 7.5: The top 5 configurations for the architecture tuning of the Colorado dataset

Table 7.16 and 7.17 show a different story, where it is generally observed that the ensemble for this experiment only focuses on the lower/mid regions. The ensemble therefore is not optimal in capturing the actual fluctuations and intricacies when tuned to minimize MAE for the Colorado dataset.

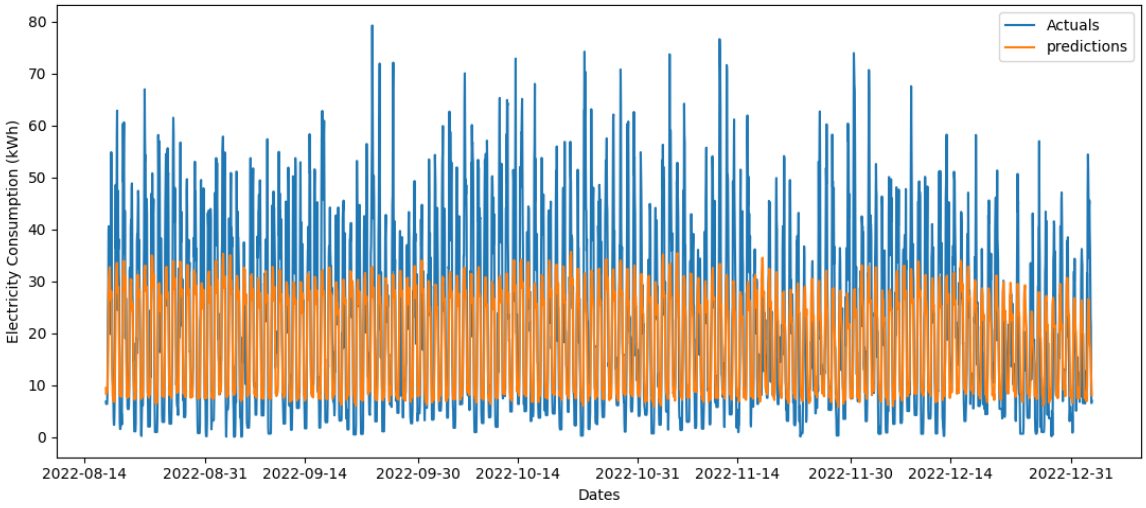

Figure 7.16: The Configuration with the lowest MAE score from the top-5

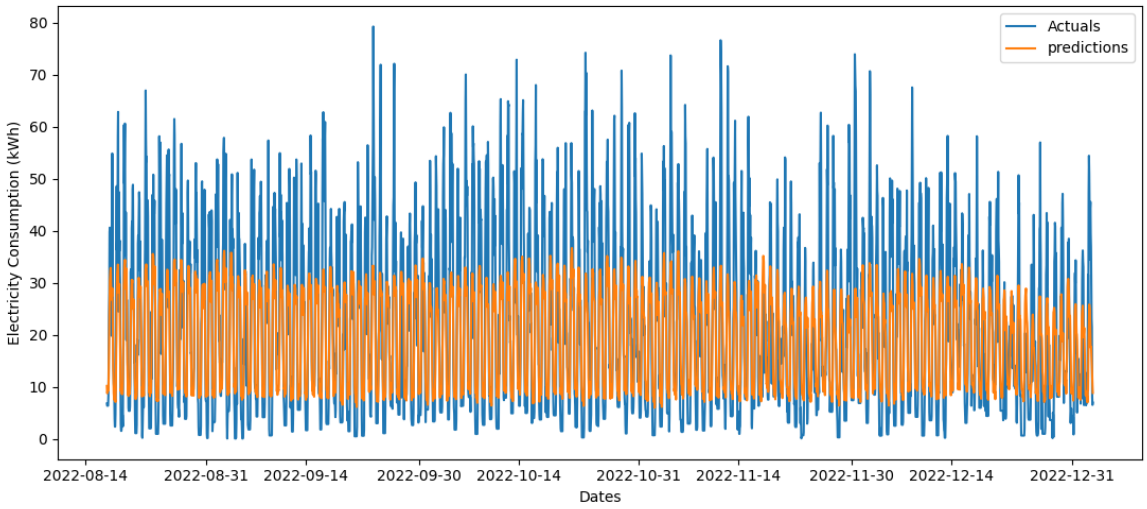

Figure 7.17: The Configuration with the highest MAE score from the top-5

7.3.2 Optimizing the Architecture to Minimize Huber Loss for the SDU Dataset

As seen in Table 7.6, the best performing configuration is GRU, PatchMixer and xPatch with the Huber loss score of 2.78. This finding is consistent with expectations, where specifically xPatch and GRU have the best results. The plots showed that xPatch and GRU tended to undershoot the upper regions, whereas PatchMixer had better success capturing the upper parts. Combining the predictions highlights the potential of prediction aggregation and the concept of the ensemble as a whole. Our findings also reveal that ensemble learning models did not contribute to the best configurations for the SDU dataset.

Configuration	**Huber Loss**	**MAE**	**MSE**
GRU, PatchMixer, xPatch	**2.78**	**11.26**	**430.43**
LSTM, GRU, PatchMixer	3.08	12.45	550.38
AdaBoost, LSTM, GRU, PatchMixer, xPatch	4.25	17.14	839.30
Random Forest, LSTM, GRU, PatchMixer, xPatch	4.46	17.95	794.61
AdaBoost, GRU, PatchMixer, xPatch	4.53	18.24	895.91

Table 7.6: The top-5 configurations for the architecture tuning of the SDU dataset

The plots, shown in Figure 7.18 and 7.19, show the best and worst top-5 configuration. Here, it is clear that the best performing configuration captures the upper region better compared to the worst performing configuration. Although better at capturing the lower and middle

regions, it still struggles to capture the upper parts and its intricacies, demonstrating room for improvement.

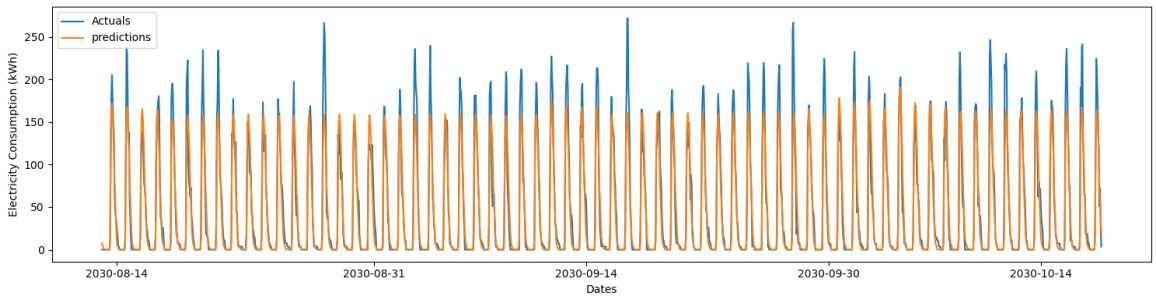

Figure 7.18: Plot for the configuration with the top-5 lowest Huber loss score

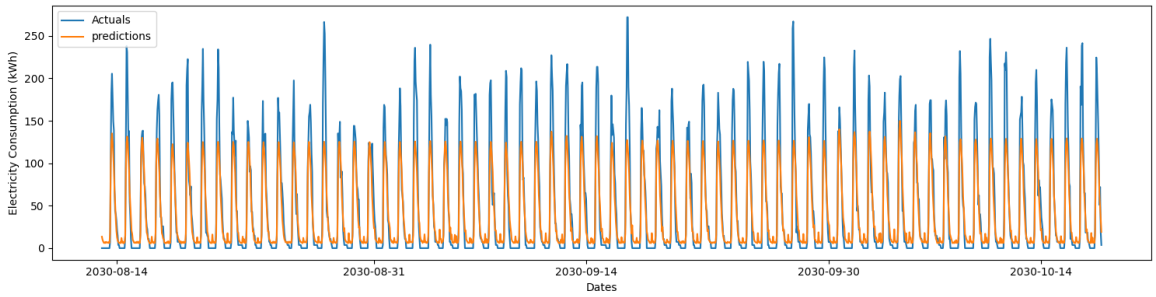

Figure 7.19: Plot for the configuration with the top-5 highest Huber loss score

7.3.3 Optimizing the Architecture to Maximize Recall while Minimizing MAE for the Colorado Dataset

Table 7.7 shows the ensemble and the highest recall scored configurations for capturing overloads. By examining the recall score, it is clear that the models effectively capture most of the overload events. However, as seen previously in this table, the MAE score is high, hinting that the model may overshoot to achieve this recall score. This can also be attributed to the choice of models, which consists mainly of the ensemble models that have been seen to overshoot in the Colorado dataset. The other configurations combine mainly LSTM and GRU with the ensemble learning models, which reduces the capacity to capture overloads, in exchange for a lower MAE and MSE score.

Configuration	Recall	MAE	MSE
Random Forest, Gradient Boosting, AdaBoost	**0.86**	44.51	2771.38
Random Forest, Gradient Boosting, GRU	0.80	34.82	1645.96
Random Forest, Gradient Boosting, AdaBoost, LSTM	0.79	34.45	1624.00
Random Forest, Gradient Boosting, AdaBoost, GRU	0.78	**33.27**	**1514.72**
Random Forest, AdaBoost, LSTM	0.77	34.34	1618.06

Table 7.7: Best performing ensemble configuration for classifying overloads, by maximizing recall and minimizing Huber loss for the Colorado dataset

However, when looking at Figure 7.20, 7.21 and 7.22, it is observed that the ensemble does not overshoot fully above the actual values. Rather, it follows the upper regions of the data, thus effectively capturing most of the overload events. One thing to notice is that it does that at the expense of capturing the lower regions, which is reasonable with the objective of solely capturing overloads.

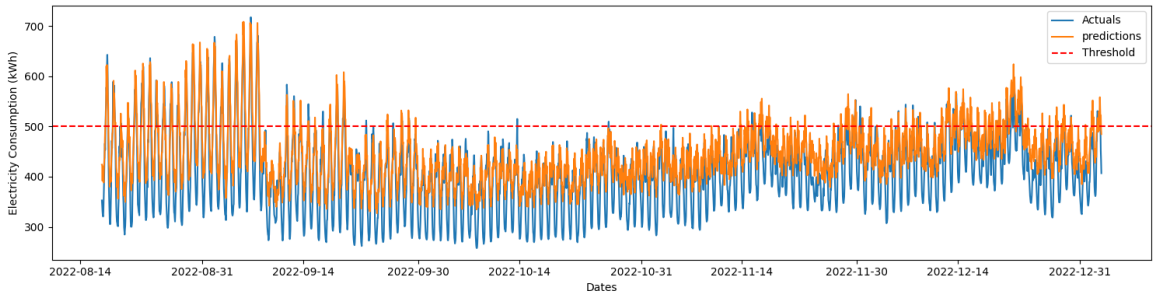

Figure 7.20: Plot for the configuration with the top-5 highest recall score

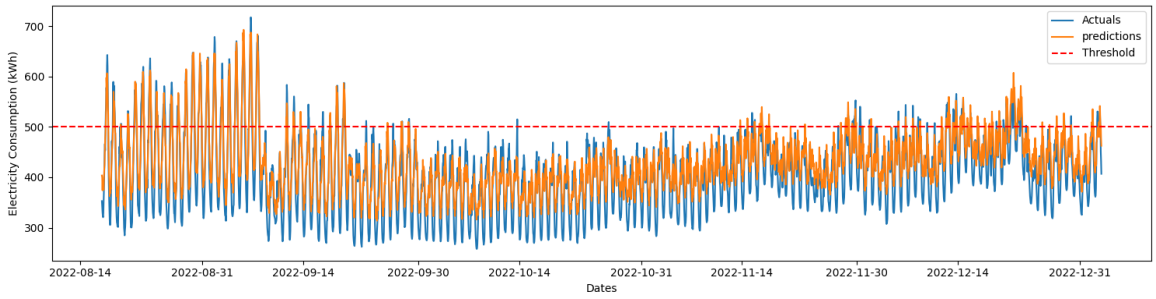

Figure 7.21: Plot for the 4th placed configuration

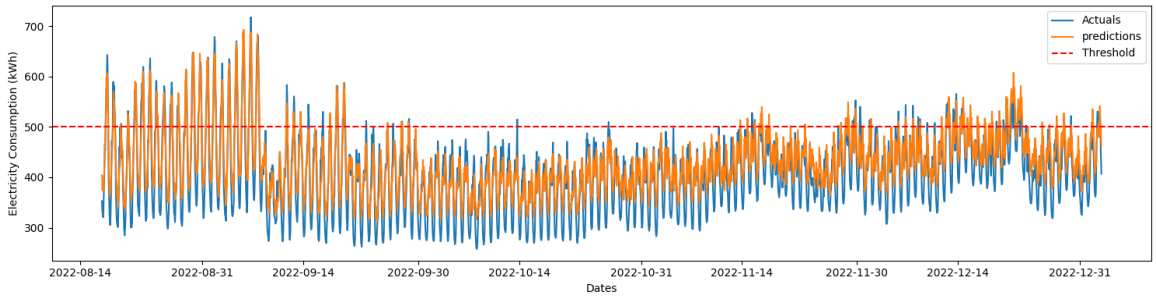

Figure 7.22: Plot for the configuration with the top-5 lowest recall score

7.3.4 Optimizing the Architecture to Maximize Recall While Minimizing Huber loss for the SDU Dataset

Table 7.8 shows the results for the overload classification for the SDU dataset. Here, the models are proficient in capturing overload, with recall scores between 0.94 and 0.83 with relatively low MAE scores. This suggests that the configuration was able to capture the temporal features. Especially the 5th place configuration, although slightly lower in recall score, achieves a lower Huber loss and MAE, indicating that it is effective at minimizing the forecast error.

Configuration	**Rec**	**Huber**	**MAE**
Gradient Boosting, AdaBoost, GRU, PatchMixer, xPatch	**0.94**	4.17	16.81
Gradient Boosting, LSTM, PatchMixer	**0.94**	6.83	27.43
Gradient Boosting, AdaBoost, LSTM, PatchMixer	0.89	3.64	14.69
Random Forest, Gradient Boosting, PatchMixer, xPatch	0.89	27.77	111.19
RandomForestRegressor, AdaBoostRegressor, LSTM, PatchMixer, xPatch	0.83	**3.09**	**12.47**

Table 7.8: The top-5 best performing configurations for the SDU dataset

Figure 7.23, 7.24, and 7.25 showcase some of the best configurations for overload classification for the SDU data. From this, it is observed that especially the best performing configuration in Figure 7.23 and the second placed configuration on Figure 7.24 have a tendency of overshooting their predictions. This improves the rate of TPs, but nevertheless decreases the precision score, given that more FPs are prominent because of the overshooting at peak hours. However, Figure 7.25 follows the forecasting error more closely, hence its lower Huber loss and MAE score seen in Table 7.8.

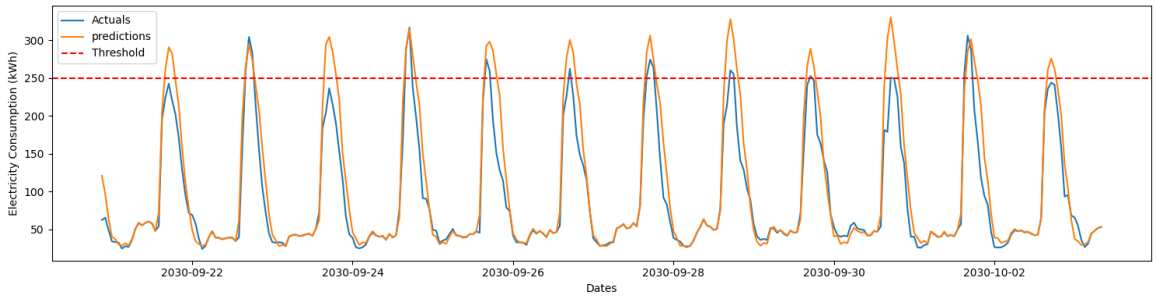

Figure 7.23: Plot for the configuration with the top-5 highest recall score

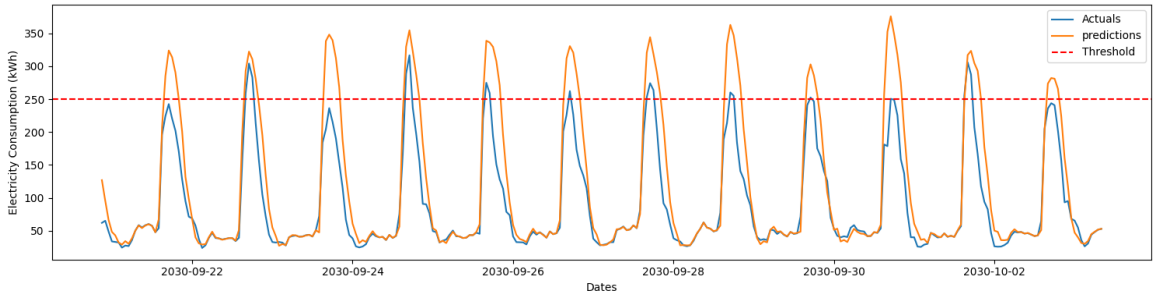

Figure 7.24: Plot for the 2nd placed configuration

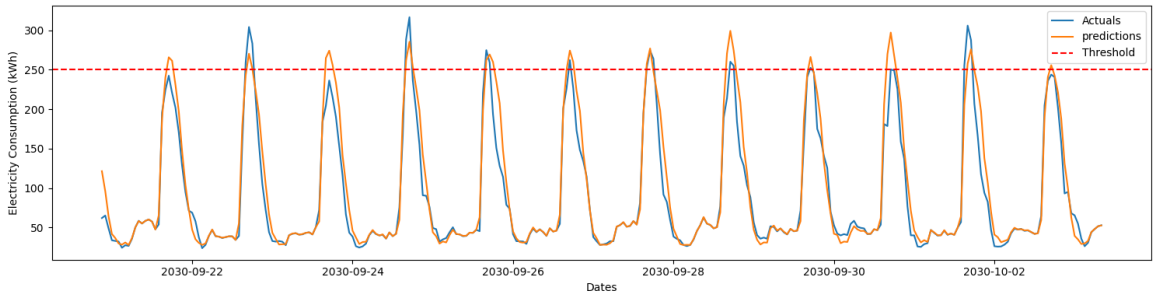

Figure 7.25: Plot for the configuration with the top-5 lowest recall score

7.4 EVCS Consumption Forecasting

The EVCS consumption forecasting experiments assess the performance of each standalone model and the ensemble with the best configuration, specifically for the EVCS consumption forecasting task.

7.4.1 EVCS Consumption Forecasting for Colorado Dataset

Table 7.9 presents the MAE and MSE results for the EVCS consumption forecasting experiment in the Colorado dataset. It indicates that deep learning models may perform better than ensemble learning models. This is reflected in the MAE scores generally being lower, where PatchMixer and AdaBoost have an MAE score of 7.59 and 8.05, respectively, showcasing the deviation. Furthermore, it is observed that the ensemble performs the worst, which was also expected to some extend since the best configuration for this ensemble, see Table 7.16, strictly captures the middle regions for minimizing MAE.

Model	MAE	MSE
Ensemble	9.29	168.80
AdaBoost	8.05	127.73
Gradient Boosting	8.48	136.05
GRU	8.04	127.96
LSTM	7.93	122.74
PatchMixer	**7.59**	**110.81**
Random Forest	8.54	146.20
xPatch	7.63	113.82

Table 7.9: MAE and MSE results from EVCS consumption forecasting for the Colorado dataset

In Figure 7.26, PatchMixer's predictions closely align with the actual values, indicating strong performance, though largely missing the peak values. Figure 7.27 shows that while xPatch captures the general trends of the test data, it tends to underestimate slightly consistently. In Figure 7.29, the ensemble model demonstrates difficulty capturing the overall pattern, with predictions consistently lower than the actual values. Similarly, Figure 7.30 illustrates that Random Forest insufficiently learned the upper and lowest parts. Finally, Figure 7.28 shows Gradient Boosting, which produces predictions that are generally close to the actual values. However, instances of overprediction at non-peaking hours contribute to its higher error metrics. This indicates that MAE may not be an appropriate loss function to effectively capture all regions of the time series, especially the upper region.

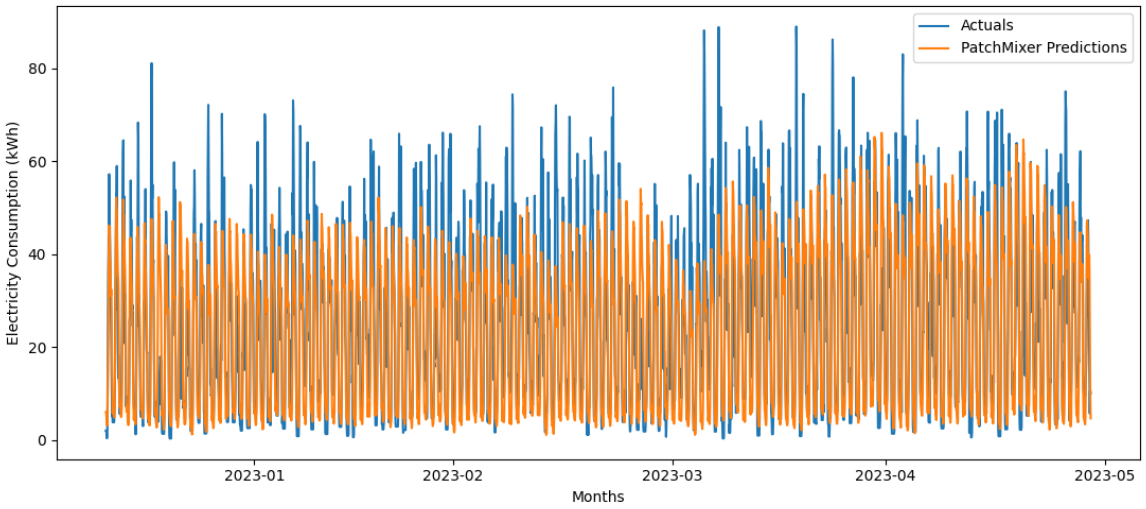

Figure 7.26: Plot for EVCS Consumption forecasting using PatchMixer on Colorado Dataset

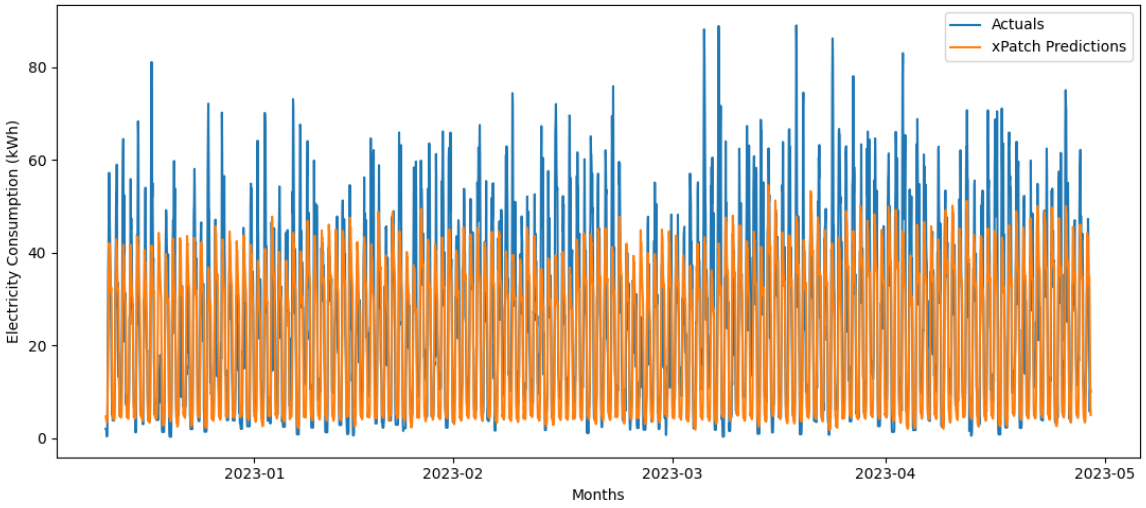

Figure 7.27: Plot for EVCS Consumption forecasting using xPatch on Colorado Dataset

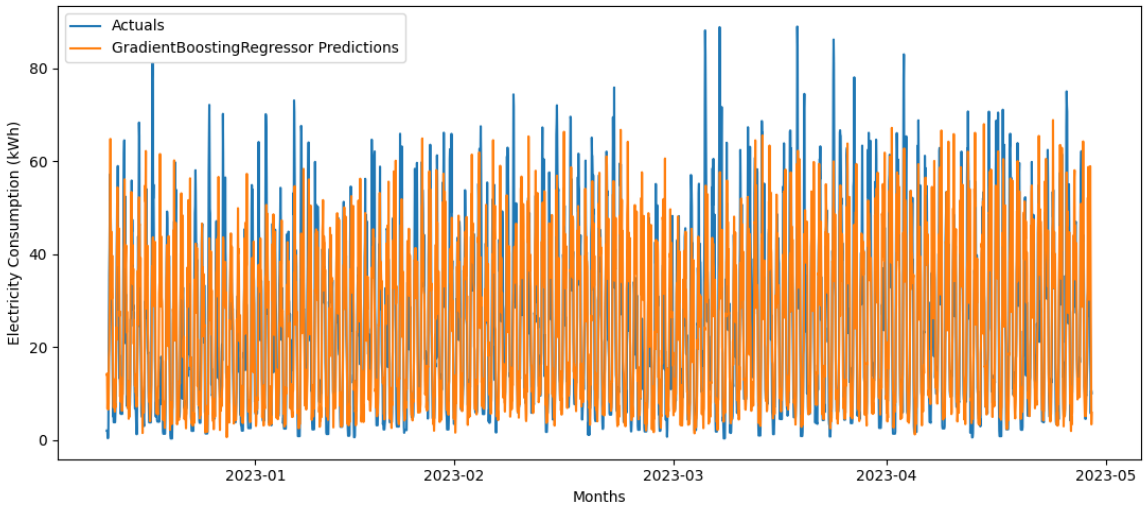

Figure 7.28: Plot for EVCS Consumption forecasting using Gradient Boosting on Colorado Dataset

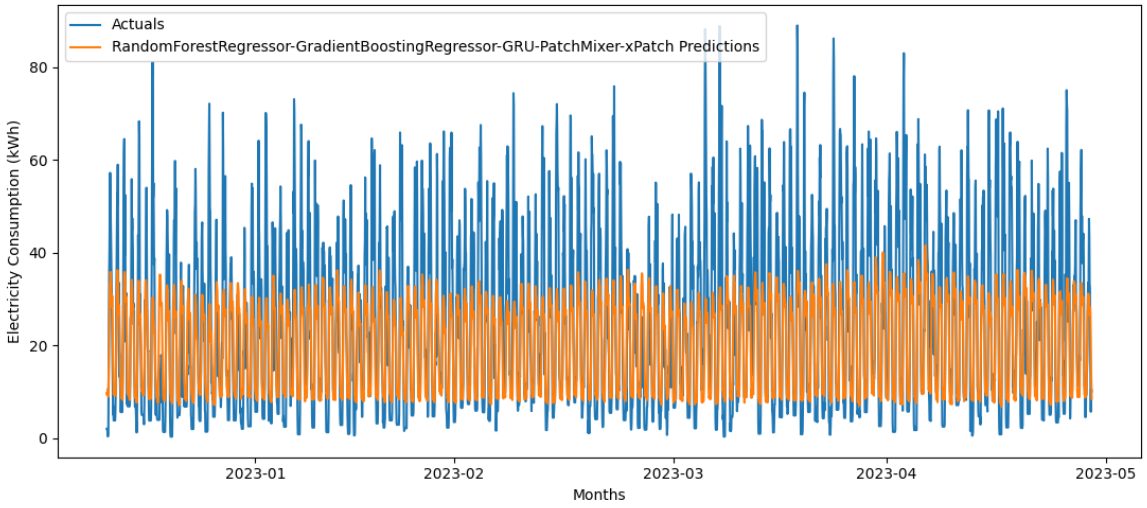

Figure 7.29: Plot for EVCS Consumption forecasting using Ensemble on Colorado Dataset

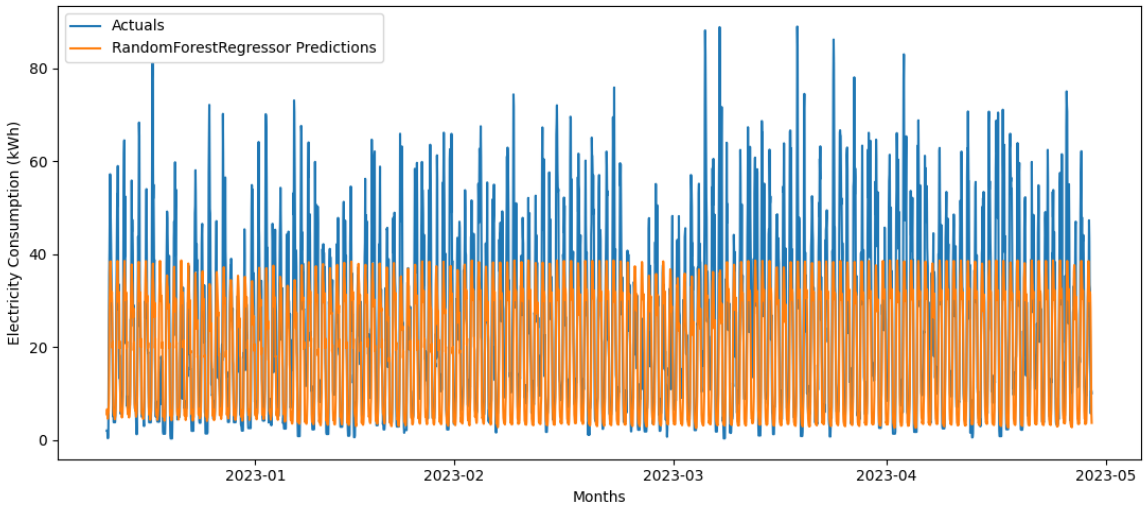

Figure 7.30: Plot for EVCS Consumption forecasting using Random Forest on Colorado Dataset

7.4.2 EVCS Consumption Forecasting for SDU Dataset

Table 7.10 shows the result for the EVCS consumption forecasting. Here, it is observed that PatchMixer has the best performing scores substantially outperforming the rest of the models. Again, the ensemble learning models seem to have a weaker performance. Surpringly, though, xPatch has a worse score overall, which is different from the expected outcome based on its corresponding hyperparameter tuning phase, seen in Section 7.2.2. In this experiment, the ensemble seems to have a stable performance that can be attributed to the rather stable models across the board.

Model	Huber Loss	MAE	MSE
Ensemble	4.66	18.78	1140.04
AdaBoost	4.75	19.11	1233.07
Gradient Boosting	5.31	21.34	1417.83
GRU	4.59	18.46	1085.39
LSTM	4.62	18.57	1109.68
PatchMixer	**4.05**	**16.32**	**880.26**
Random Forest	4.52	18.18	1086.17
xPatch	6.12	24.60	1915.38

Table 7.10: The MAE, MSE, and Huber Loss scores from EVCS consumption for SDU Dataset

Figure 7.31, 7.32, 7.33, 7.34, and 7.35 showcase rather flat predictions, ignoring the upper regions as seen previously. Gradient boosting, however, seems to capture some patterns, though not close to the upper part and not showcasing the same consumption pattern as the

upper part.

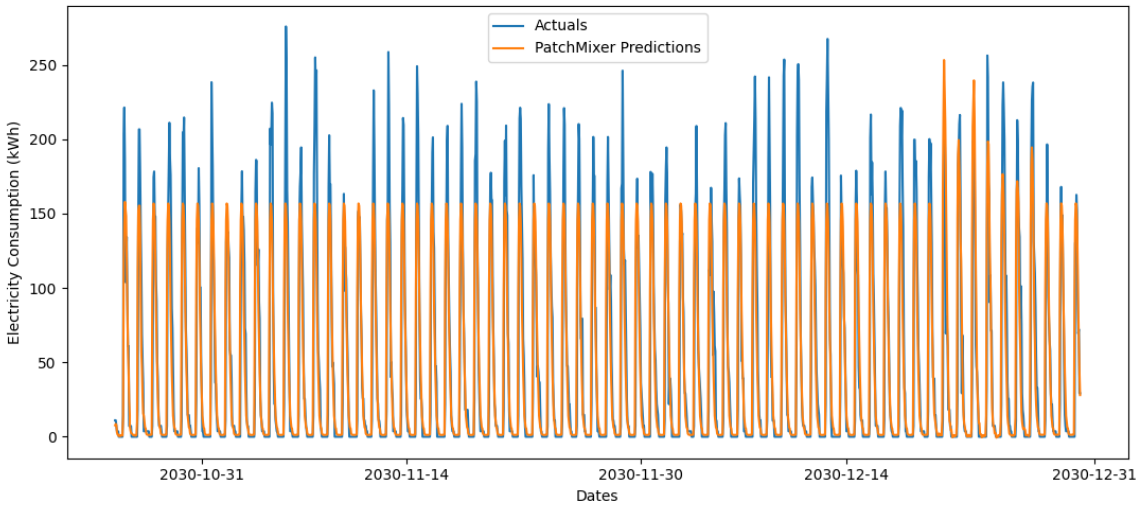

Figure 7.31: Plot for EVCS Consumption forecasting using PatchMixer on SDU Dataset

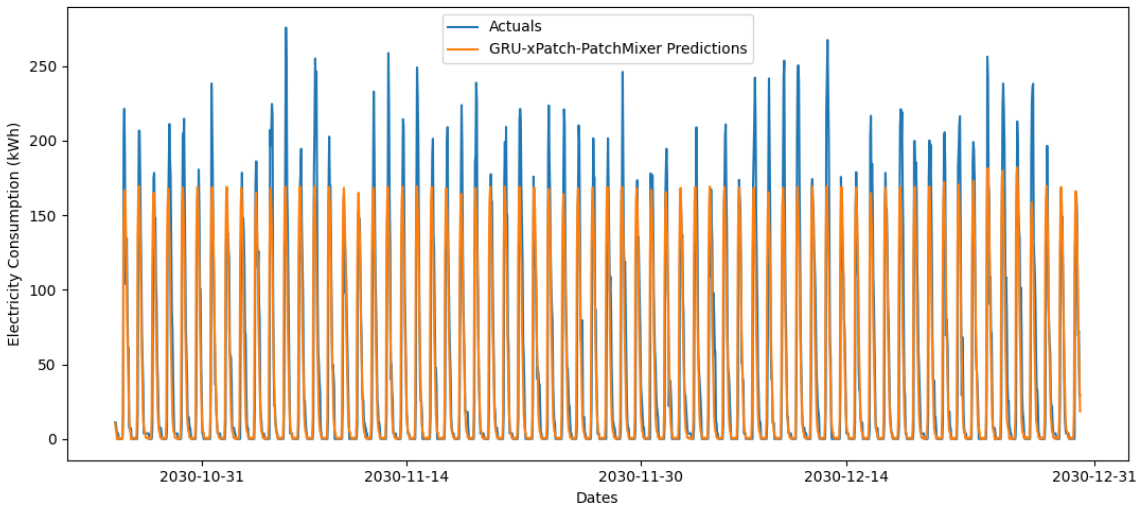

Figure 7.32: Plot for EVCS Consumption forecasting using Ensemble on SDU Dataset

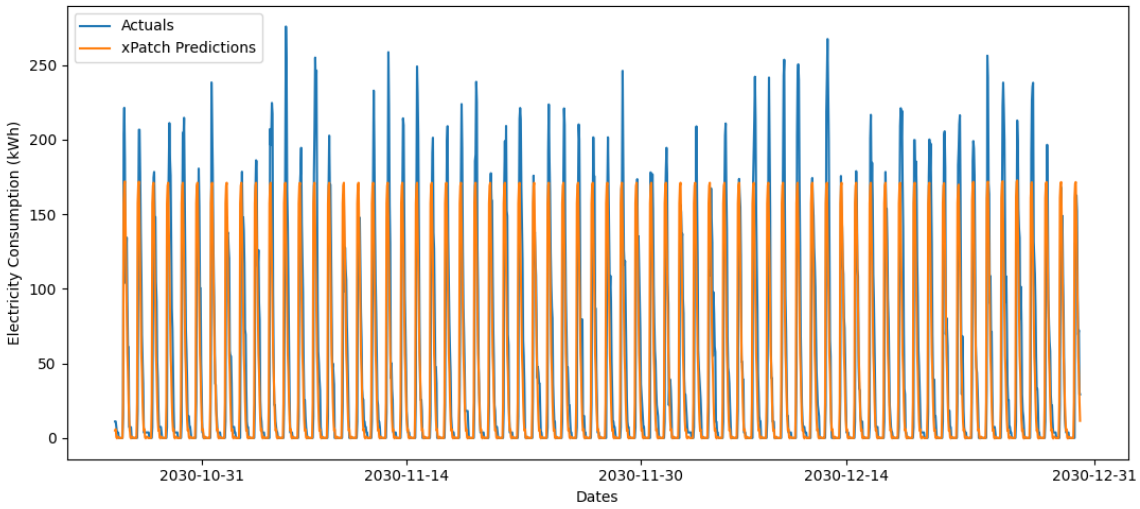

Figure 7.33: Plot for EVCS Consumption forecasting using xPatch on SDU Dataset

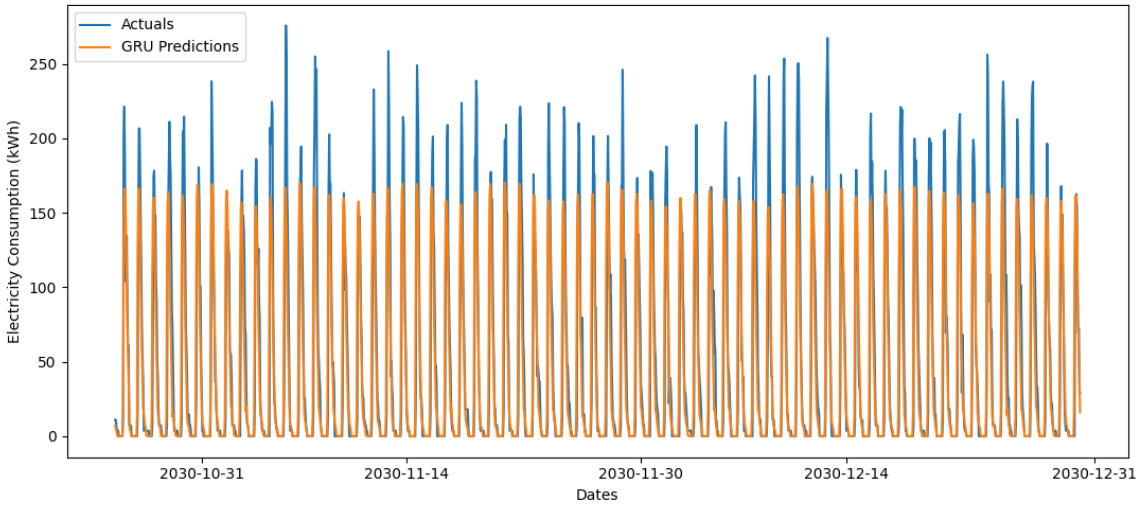

Figure 7.34: Plot for EVCS Consumption forecasting using GRU on SDU Dataset

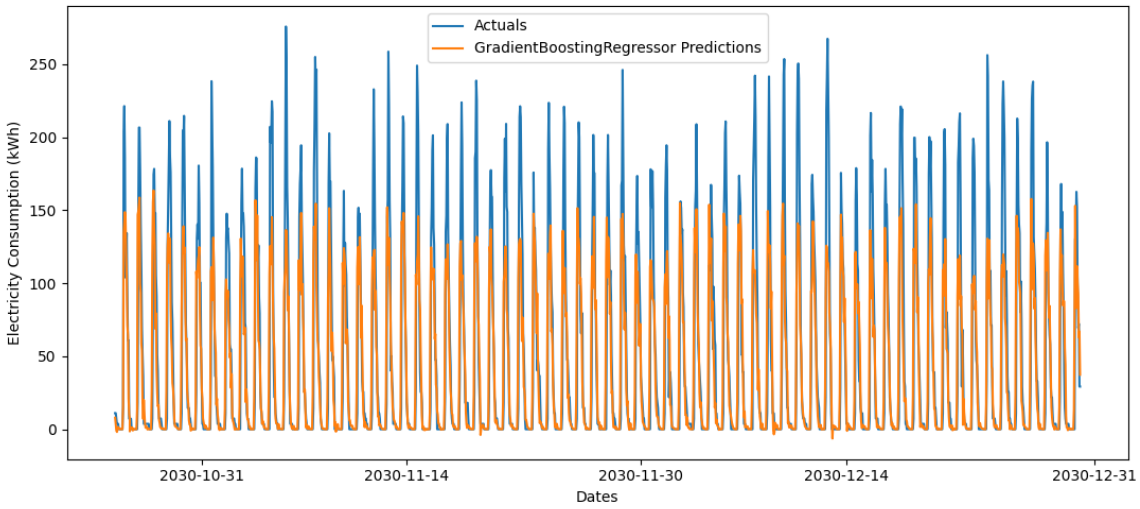

Figure 7.35: Plot for EVCS Consumption forecasting using Gradient Boosting on SDU Dataset

7.5 Overload Classification

The overload classification experiment evaluates the performance of standalone models trained without bootstrap sampling against the best-performing ensemble model for each dataset. The primary objective is to assess the models' capacity to detect overload events, as measured by the recall metric.

7.5.1 Colorado

Table 7.11 shows the overload classification for Colorado. PatchMixer achieves the highest recall score of 0.88. However, it is seen that it loses some of its precision during that process, indicating that it contributes to numerous FPs. Oppositely, GRU, xPatch and LSTM, although with slightly lower recall score, achieve a significantly higher precision, indicating that these models are less likely to overshoot. The ensemble in this experiment shows a somewhat low recall score of 0.41 compared to its tuning phase, which had a recall score of 0.86, see Table 7.7.

Model	MAE	Acc	Pre	Rec
Ensemble	31.02	0.91	0.88	0.41
AdaBoost	32.94	0.88	**1.0**	0.14
Gradient Boosting	32.61	0.92	0.76	0.59
GRU	**14.73**	**0.95**	0.87	0.76
LSTM	15.35	**0.95**	0.98	0.67
PatchMixer	20.84	0.91	0.63	**0.88**
Random Forest	31.41	0.92	0.89	0.45
xPatch	14.87	**0.95**	0.88	0.71

Table 7.11: The MAE and classification metric scores for the Colorado dataset

Figure 7.36, 7.37, and 7.38 visually seem relatively effective at capturing overloads. This is especially the case for PatchMixer, which overshoots but ensures that it captures almost all overload events. GRU captures slightly less, but follows the forecasting error much more closely, compared to xPatch and PatchMixer. The ensemble, see Figure 7.39, however does not follow the data equally well. This can be attributed to its configuration mainly, where the models in general perform substantially worse than its best performing counterparts.

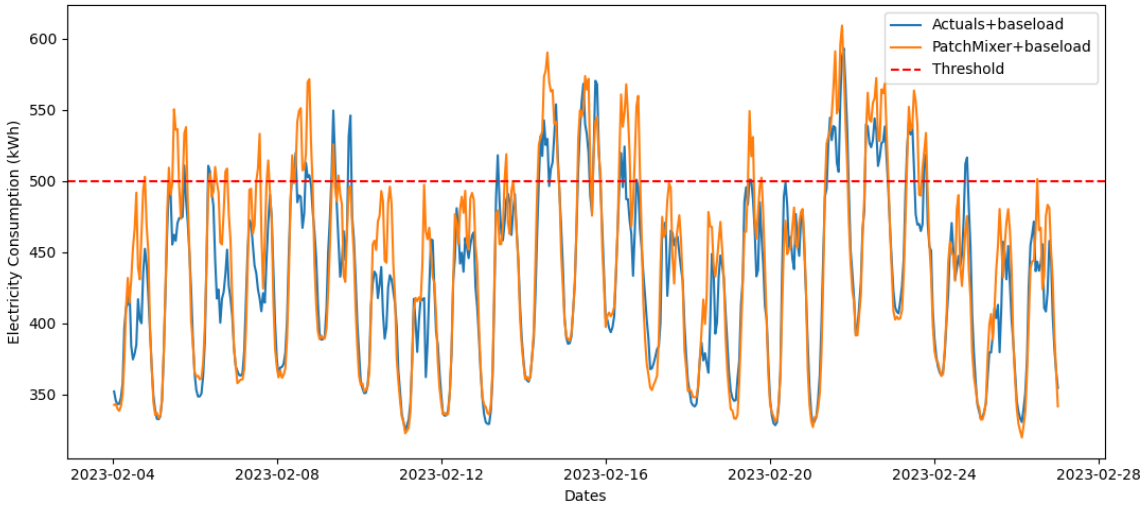

Figure 7.36: Overload classification for PatchMixer for Colorado Dataset

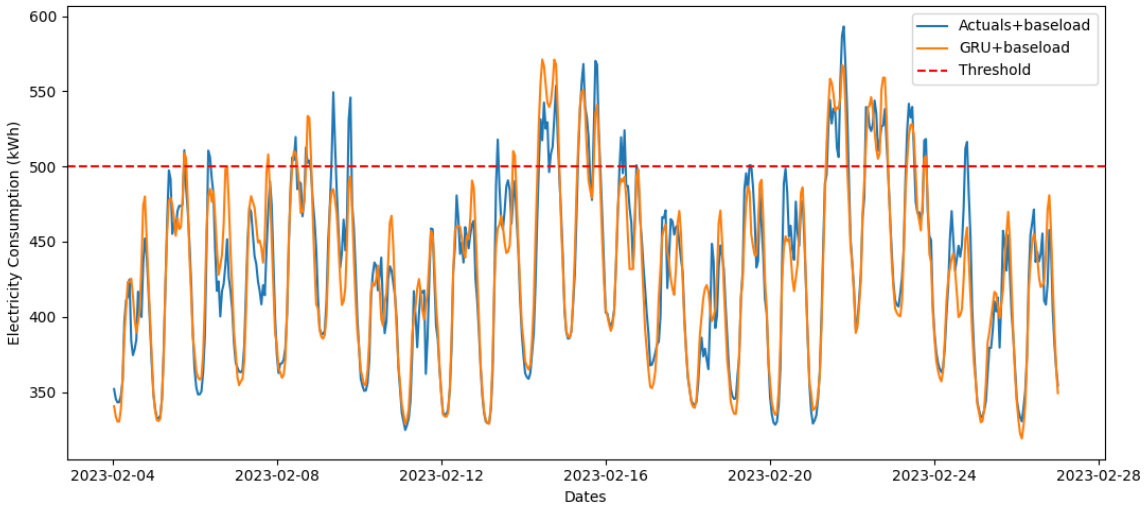

Figure 7.37: Overload classification for GRU for Colorado Dataset

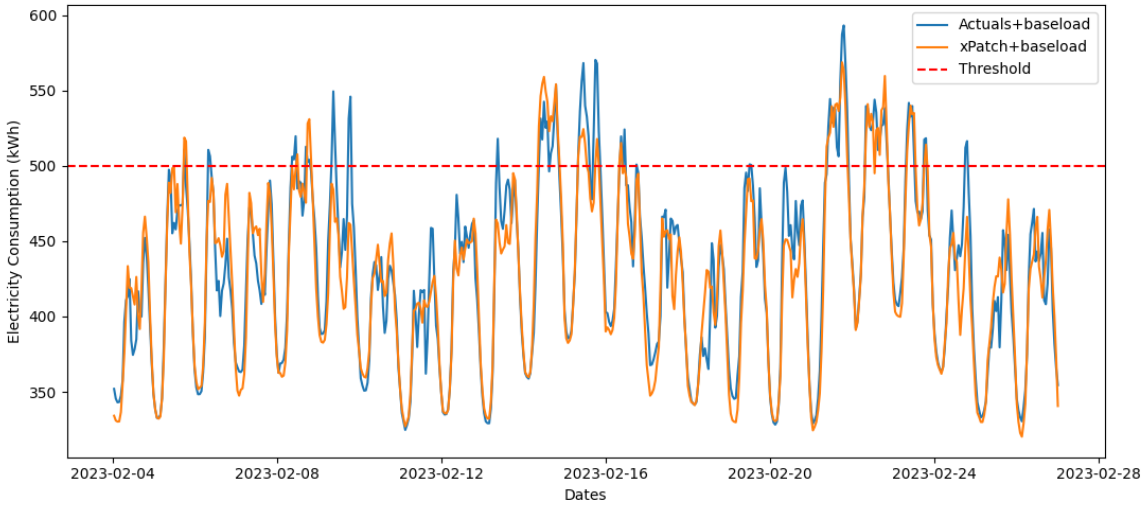

Figure 7.38: Overload classification for xPatch for Colorado Dataset

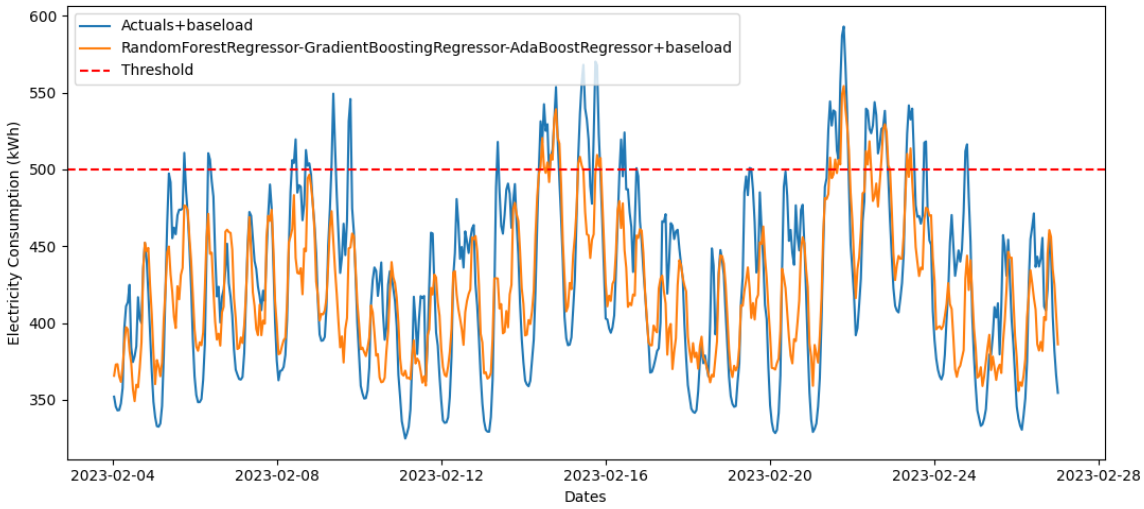

Figure 7.39: Overload classification for Ensemble for Colorado Dataset

7.5.2 SDU

For table 7.12, it is clear that only xPatch and PatchMixer captures overloads at an acceptable rate. The cause of this can be attributed to the larger gap between the baseload and threshold, making it necessary to capture the upper regions effectively. The ensemble, nevertheless, gets a 0.10 rec score, which is not sufficient.

Model	MAE	Huber	Acc	Pre	Rec
Ensemble	37.41	9.32	0.88	**1.0**	0.10
AdaBoost	21.52	5.36	0.87	0	0.0
Gradient Boosting	23.67	5.89	0.87	0.67	0.05
GRU	20.46	5.09	0.87	0	0.0
LSTM	**20.17**	**5.02**	0.87	0	0.0
PatchMixer	25.44	6.33	**0.91**	0.67	0.54
Random Forest	20.36	5.06	0.87	0	0.0
xPatch	26.49	6.59	0.88	0.53	**0.64**

Table 7.12: Overload classification for the SDU dataset

Figure 7.40 and 7.41 showcases different performances, where xPatch consistently predicts the overloads, but not in a sufficient timeframe to capture the full duration of the overloads, hence its not a higher score. Contrarily, PatchMixer, although with a good recall score, does not fully capture the temporal patterns and overshoots in the middle parts of the data. This finding suggests that it may not fully capture the upper regions optimally. Figure 7.42 and 7.43 reveal the upper regions were largely missed, hence their low recall score.

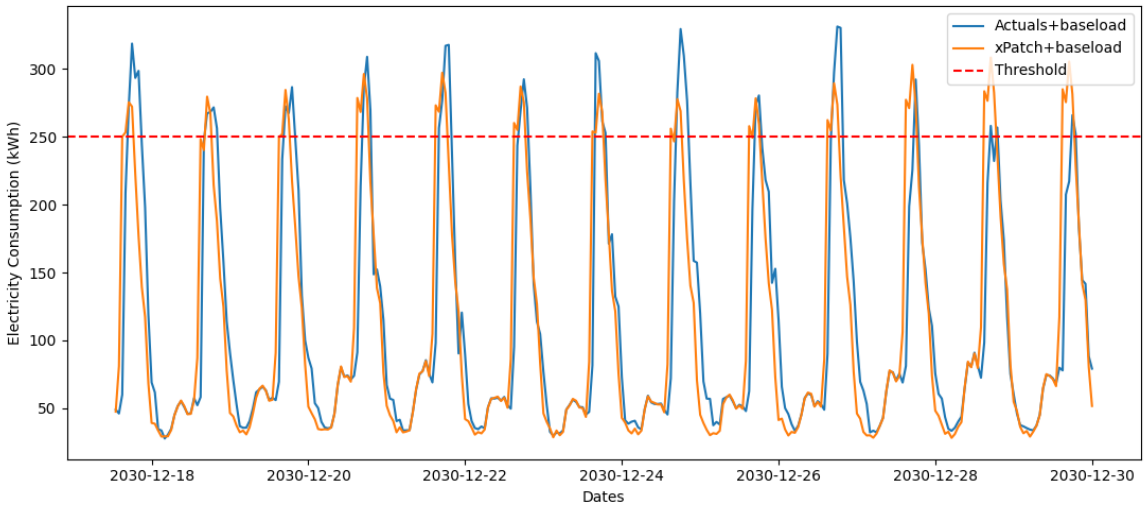

Figure 7.40: Overload classification for xPatch

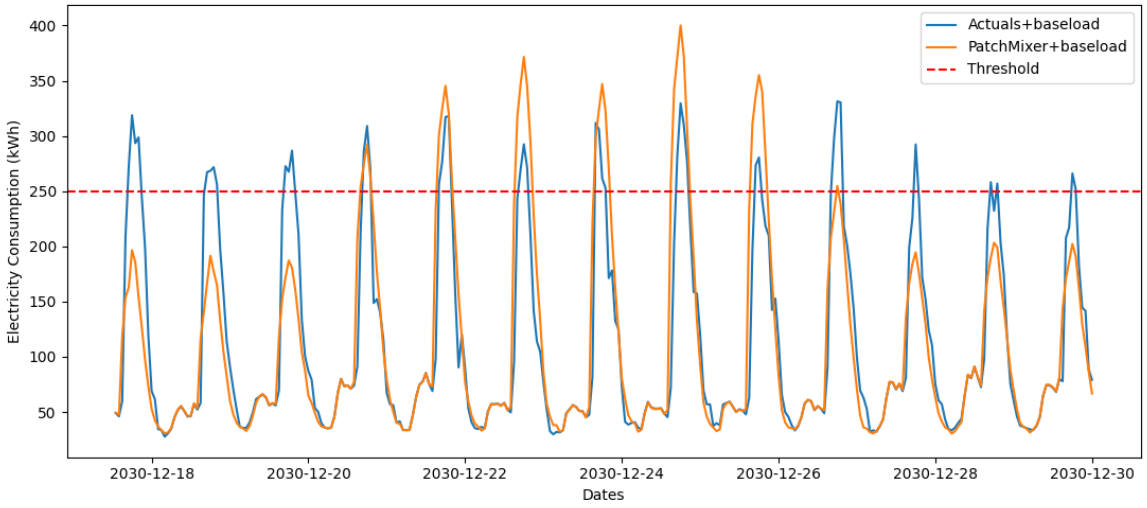

Figure 7.41: Overload classification for PatchMixer

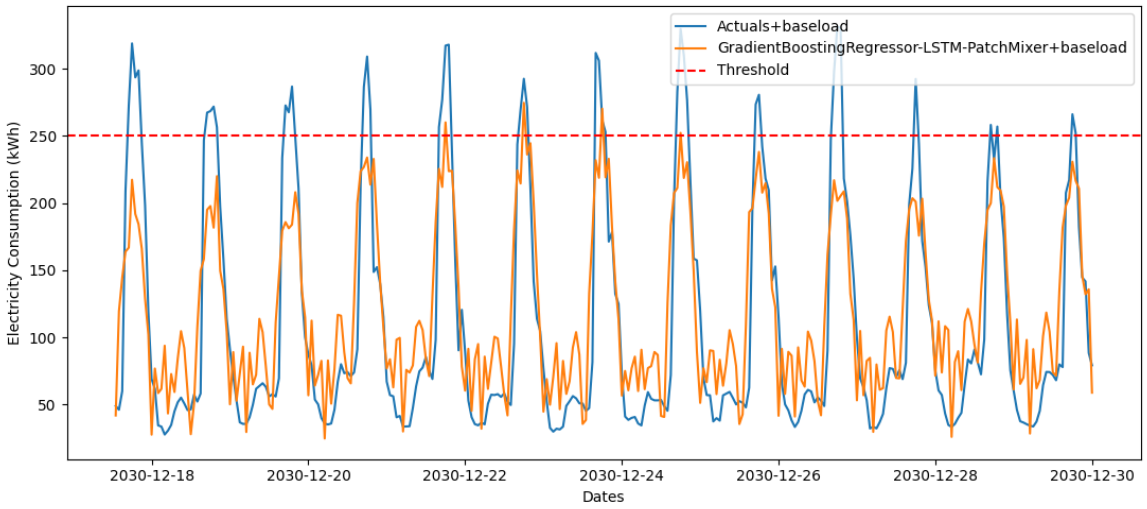

Figure 7.42: Overload classification for Ensemble

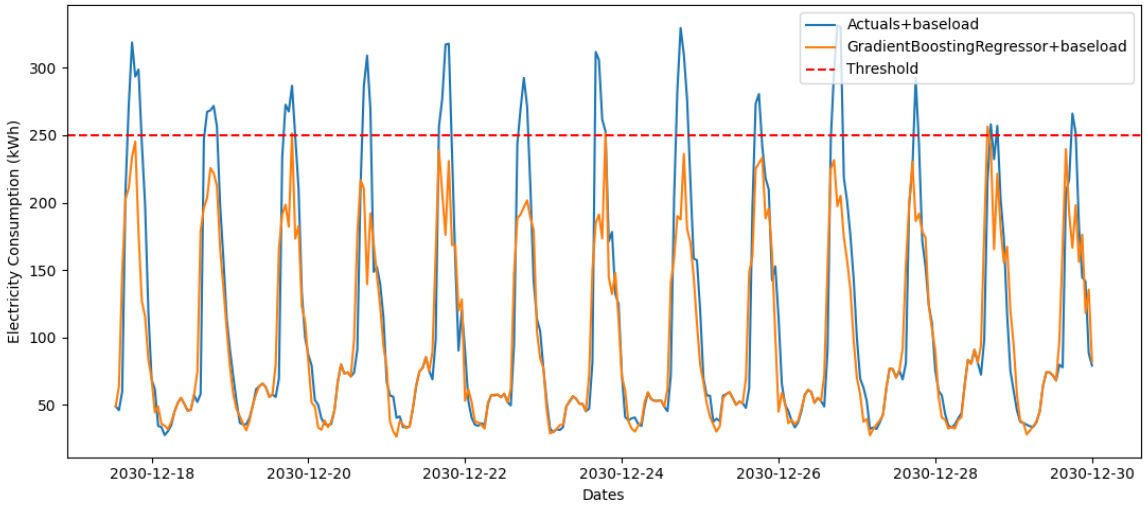

Figure 7.43: Overload classification for Gradient Boosting

8 — Discussion

8.1 Evaluating Hyperparameter Tuning Results

The hyperparameter tuning experiments across the Colorado and SDU datasets reveal that ensemble learning models are overall ineffective, on this setup, at capturing the upper regions of the data in almost all cases. Contrary to the study in Section 4.1 finding that Random Forest and AdaBoost outperformed LSTM and Gradient Boosting in EVCS consumption forecasting, this study suggests that these neural network-based approaches are better suited to capture the underlying patterns and complexities of the dataset, leading to improved consumption forecast accuracy and better event detection capabilities.
Among the deep learning models, PatchMixer and xPatch achieve strong performances across the board, suggesting the patching technique and their architectures are very effective at forecasting and classifying EVCS consumption data.
When looking at the MAE and Huber loss hyperparameter tunings, it is clear that the models are capable of capturing the lower and middle regions of the time series but fail substantially at capturing the upper part. This came as a surprise, as it was expected that the models were able to capture some of the seasonality and not almost exclusively remain on the same range.
The overload hyperparameter tunings generally went as expected for the Colorado dataset, where PatchMixer, GRU, and xPatch outperformed the other models. The Colorado dataset went substantially well, with most models achieving above 0.50 overload capturing, indicating that MAE + recall may be an effective combination to achieve better capture of the upper regions of the data. Alternatively, the SDU dataset went surprisingly different than expected, since e.g. LSTM and GRU had relatively low scores in many models, with only xPatch and xPatch achieving great results. This highlights the instability that the models has on the SDU dataset.

8.2 Evaluating Architecture Tuning Results

Overall, the ensemble results showed an underwhelming performance in the MAE and Huber loss experiments, where it struggled to account for the upper regions. However, during the multi-objective tuning, it was somewhat surprising that tuning on both recall and a loss function was effective at capturing the upper regions in the tuning phase. Another surprise was the multi-objective Colorado dataset tuning, where the three ensemble learning models combined outperformed the other models, when classifying overloads for Colorado. This

strategy of staying at the upper regions achieves a solid MAE score, which would rank top 2 with the found score in Table 7.3.
Classification tuning for the SDU dataset was also extremely good, almost capturing all overloads in most configurations. These high scores were somewhat expected as the threshold and baseload were relatively close, but nevertheless impressive. The results can therefore be misleading in representing their effectiveness in unseen data.

8.3 Evaluating EVCS Consumption Forecasting

The consumption forecasts came with relatively expected results for most of the models. PatchMixer and xPatch, although with the best scores, inadequately modelled the upper regions, also seen in almost all other models. Gradient Boosting, though, achieved a substantially better result, which successfully captures all regions of the data, though with a slightly lower MAE score. However, this is generally better, since underrepresenting regions of data is not optimal, and also not in EVCS consumption forecasting. The ensemble in general had a slightly weak performance across the architecture tunings with flat and repetitive patterns not capturing the time series properly.

8.4 Evaluating Overload Classification Results

The overload classification had different results when comparing both datasets. The Colorado dataset had the models retain most of its recall score from the tuning phase, which indicates that the models still were able to generalize to new data to a large extent. The same cannot be said for the SDU dataset, where only xPatch and PatchMixer retained an acceptable recall score. One major factor for this result comes from the major gap between baseload and threshold, which requires capturing the upper regions of the forecast substantially to get acceptable scores.

8.5 DPAD + Fredformer Issues with Crashing During Tuning

It was initially planned for both DPAD and a recent good performing transformer-based model, called Fredformer to be a part of the ensemble. However, both of these models required an impressive amount of computational power, which was not feasible for the AI-LAB hardware setup. Fredformer tended to crash instantly on initialization. DPAD on the other hand was able to train and sometimes generate predictions. However, most of the time, the model stops suddenly, even though the model was significantly downscaled. This occurred despite implementing safeguards in the objective function to handle failed trials and allow the tuning process to continue tuning. It appeared that the model exceeded the available memory capacity on the GPU nodes, causing the underlying job to be terminated

by the system. However, these terminations are not always correctly captured or reported by the Slurm workload manager. As a result, the job may appear to still be running in Slurm's interface, while in reality, the process has stalled or been forcefully killed due to a memory fault. This effectively destroyed a lot of progress in ensemble architecture tuning, where about 10 studies were attempted with DPAD, but without success. The next plan was to implement it as an individual model, but since the deadline was getting close, this was not actualized.

8.6 Limitations of the study

The study has several limitations that need to be addressed. First of all, the diversity of the ensemble was not as broad as planned, since DPAD and Fredformer especially killed a lot of progress in the grand time scheme. Adding more diversity could have better represented the capabilities of the ensemble, especially in cases where most models underperform significantly, thus reducing all scores of the ensemble (if in the configuration).
Another limitation was the availability of AI-LAB, since the traffic at times made the HPC service almost impossible to use, with up to an hour of wait time. Combined with the fact that these models could not be run on one of the students' computer, a lot of possible progress was also lost. Furthermore, the scheduling of AI-LAB was not optimally made, where smaller tasks were not stacked on the same node, but rather allocated to separate individual nodes. As our project required a full resource capabilities of one whole node, this delayed the tasks even further, in some cases, where better scheduling could have saved a lot of time.
Third of all, a substantial amount of progress was also lost in pursuing on using the whole SDU dataset for training, validation, and testing. This required remarkably longer training times, and with very poor results overall from these old experiments, resources and time could have been placed differently.

9 — Conclusion

The increasing adoption of electric vehicles (EVs), driven by international sustainability goals, has led to a rapid expansion of electric vehicle charging stations (EVCS). While this transition supports carbon neutrality, it introduces substantial challenges to the existing power infrastructure, particularly the risk of transformer overload, degradation, and voltage instability due to rising and volatile electricity demand.

This study focuses on forecasting the electricity consumption of EVCS by analyzing time series data, aiming to evaluate standalone models with a proposed ensemble model on EVCS electricity consumption forecasting and measuring their ability to capture overloads. A real-world EVCS dataset based in Colorado is examined together with a synthetic dataset for Danish residential charging. The time series are aggregated at an hourly resolution with a 24-hour forecast horizon. The dataset is split into training, validation, and test sets using a 60/20/20 ratio.

The ensemble model introduced combines diverse learners, including tree-based algorithms and deep learning architectures, to improve forecasting accuracy and overload detection. The models evaluated include xPatch, PatchMixer, LSTM, GRU, Random Forest, Gradient Boosting and AdaBoost.

Deep learning models, particularly PatchMixer and xPatch, consistently outperformed both ensemble and standalone models across all experiments in electricity consumption forecasting and overload classification. These models achieved the lowest forecasting errors overall, demonstrating strong capabilities in learning temporal patterns in EVCS electricity demand. However, a key limitation was their difficulty in modeling extreme upper consumption values, which affected their accuracy during periods of peak load—an important factor in energy planning and infrastructure protection.

Despite this, their performance in overload detection was promising on the Colorado dataset, achieving high recall scores of 0.88 (PatchMixer) and 0.71 (xPatch), indicating effective identification of potential transformer stress events. Performance on the SDU dataset proved more challenging, with lower recall scores of 0.54 and 0.64, partly due to a wider gap between baseload and threshold levels.

In contrast, the ensemble models delivered mixed results. While the architecture had the potential to combine complementary strengths, its performance was limited by unstable configurations and inconsistent or underperforming base learners. This instability hindered its ability to generalize effectively across both datasets and forecasting objectives.

10 — Future Work

10.1 Better Preprocessing for Sparse Datasets such as the SDU Dataset

Forecasting on the SDU dataset requires a more comprehensive preprocessing pipeline to allow models to effectively capture underlying patterns. One of the main challenges arises from especially the SDU dataset's sparse nature, specifically the prevalence of zeros, which often leads to instability during training and underfitting. To address this, a broader exploration of preprocessing techniques may be necessary, including strategies such as masking zeros and interpolating between consecutive zeros. For example, applying linear or quadratic interpolation can help smooth transitions in the data, with linear interpolation fitting a straight line and quadratic interpolation fitting a curved line between non-zero values that bracket sequences of zeros. Methods such as these, may allow the models to better capture the temporal patterns in the data.

10.2 Exploring the Feature Attribution and Failure Points

Understanding why some models underperform on certain datasets, e.g. SDU in this case, could benefit from a comprehensive feature selection analysis. This process can identify whether any features may have negatively impacted the results in both the context of electricity consumption forecasting and transformer overload capturing, and if some certain types are specifically effective.

10.3 Different Forecasting Horizons

Although this study focuses solely on a 24-hour forecasting horizon, incorporating additional horizons of 6 and 12 hours could yield better results. This can be attributed to the fact that shorter forecasts are generally more accurate to do. These shorter time frames also mimic how real world implementations of overload detection can operate. This, however, could also entail a smaller granularity of the data, e.g. 15 minuter or 30 minutes, effectively allowing overload signs to be captured earlier with finer granularity.

A — Appendix

A.1 The search space for every model

The section contains the Optuna search space for all models.

```
AdaBoost = {
    'n_estimators': trial.suggest_int('n_estimators', 50, 200),
    'learning_rate_model': trial.suggest_float('learning_rate_model', 0.01, 1.0),
}
RandomForest = {
    'n_estimators': trial.suggest_int('n_estimators', 50, 200),
    'max_depth': trial.suggest_int('max_depth', 1, 20),
    'min_samples_split': trial.suggest_int('min_samples_split', 2, 20),
    'min_samples_leaf': trial.suggest_int('min_samples_leaf', 1, 20),
    'max_features': trial.suggest_float('max_features', 0.1, 1.0),
}
GradientBoosting = {
    'n_estimators': trial.suggest_int('n_estimators', 50, 200),
    'max_depth': trial.suggest_int('max_depth', 1, 20),
    'min_samples_split': trial.suggest_int('min_samples_split', 2, 20),
    'min_samples_leaf': trial.suggest_int('min_samples_leaf', 1, 20),
    'max_features': trial.suggest_float('max_features', 0.1, 1.0),
    'learning_rate_model': trial.suggest_float('learning_rate_model', 0.01, 1.0),
}
```

Listing A.1: Search space for all ensemble learning models

```
LSTM = {
  'hidden_size': trial.suggest_int('hidden_size', 50, 200),
  'num_layers': trial.suggest_int('num_layers', 1, 10),
  'dropout': trial.suggest_float('dropout', 0.0, 1),
}
GRU = {
    'hidden_size': trial.suggest_int('hidden_size', 50, 200),
    'num_layers': trial.suggest_int('num_layers', 1, 10),
    'dropout': trial.suggest_float('dropout', 0.0, 1),
}
PatchMixer = {
    "enc_in": params['input_size'],
    "seq_len": params['seq_len'],
    "pred_len": params['pred_len'],
    "batch_size": params['batch_size'],
    "patch_len": trial.suggest_int("patch_len", 4, 32, step=4),
    "stride": trial.suggest_int("stride", 1, 16, step=1),
    "mixer_kernel_size": trial.suggest_int("mixer_kernel_size", 2, 16, step=2),
    "d_model": trial.suggest_int("d_model", 128, 1024, step=64),
    "dropout": trial.suggest_float("dropout", 0.0, 0.5, step=0.05),
    "head_dropout": trial.suggest_float("head_dropout", 0.0, 0.5, step=0.05),
    "e_layers": trial.suggest_int("e_layers", 1, 4),
}
xPatch = {
    "seq_len": params['seq_len'],
    "pred_len": params['pred_len'],
    "enc_in": params['input_size'],
    "patch_len": trial.suggest_int('patch_len', 12, 48, step=6),
    "stride": trial.suggest_int('stride', 12, 48, step=6),
    "padding_patch": trial.suggest_categorical('padding_patch', ['end', 'None']),
    "revin": trial.suggest_int('revin', 0, 1),
    "ma_type": trial.suggest_categorical('ma_type', ['reg', 'ema']),
    "alpha": trial.suggest_float('alpha', 0.0, 1.0),
    "beta": trial.suggest_float('beta', 0.0, 1.0),
}

```

Listing A.2: Search space for all deep learning models

Specific Search space for PatchMixer, Gradient Boosting and xPatch for SDU

```
PatchMixer = {
    'enc_in': params['input_size'],
    'seq_len': params['seq_len'],
    'pred_len': params['pred_len'],
    'batch_size': params['batch_size'],
    'patch_len': trial.suggest_int("patch_len", 4, 32, step=4),
    'stride': trial.suggest_int("stride", 2, 16, step=2),
    'mixer_kernel_size': trial.suggest_int("mixer_kernel_size", 2, 16, step=2),
    'd_model': trial.suggest_int("d_model", 128, 1024, step=64),
    'dropout': trial.suggest_float("dropout", 0.0, 0.8, step=0.1),
    'head_dropout': trial.suggest_float("head_dropout", 0.0, 0.8, step=0.1),
    'e_layers': trial.suggest_int("e_layers", 1, 10),
}

GradientBoosting = {
    'n_estimators': trial.suggest_int('n_estimators', 100, 500),
    'max_depth': trial.suggest_int('max_depth', 1, 10),
    'min_samples_split': trial.suggest_int('min_samples_split', 2, 20),
    'min_samples_leaf': trial.suggest_int('min_samples_leaf', 1, 20),
    'max_features': trial.suggest_float('max_features', 0.1, 1.0),
    'learning_rate_model': trial.suggest_float('learning_rate_model', 0.01, 1.0),
    'subsample': trial.suggest_float('subsample', 0.3, 1.0),
}

xPatch = {
    'seq_len': params['seq_len'],
    'pred_len': params['pred_len'],
    'enc_in': params['input_size'],
    'patch_len': = trial.suggest_int('patch_len', 2, 16, step=2),
    'stride': trial.suggest_int('stride', 1, 7, step=2),
    'padding_patch': trial.suggest_categorical('padding_patch', ['end', 'None']),
    'revin': trial.suggest_int('revin', 0, 1),
    'ma_type': trial.suggest_categorical('ma_type', ['reg', 'ema']),
    'alpha': trial.suggest_float('alpha', 0.0, 1.0),
    'beta':  trial.suggest_float('beta', 0.0, 1.0),
}

```

Listing A.3: Specific Search space for PatchMixer, Gradient Boosting and xPatch for SDU

A.2 Tuning Results

This sections covers all the tuning results from both Colorado dataset and the SDU dataset

A.2.1 Colorado Tuning Result

This section covers the full tuning results for the Colorado dataset.

Tuning minimizing MAE With Bootstrap Sampling

Model	MAE	Parameters
AdaBoost	14.11	{'batch_size': 32, 'learning_rate': 0.0009137344842562993, 'max_epochs': 1300, 'num_workers': 10, 'n_estimators': 123, 'learning_rate_model': 0.657301480230918}
GRU	6.75	{'batch_size': 128, 'learning_rate': 0.002183521345460817, 'max_epochs': 1600, 'num_workers': 9, 'hidden_size': 143, 'num_layers': 1, 'dropout': 0.23265791159785204}
Gradient Boosting	14.35	{'batch_size': 128, 'learning_rate': 0.0018689669036629699, 'max_epochs': 1900, 'num_workers': 6, 'n_estimators': 122, 'max_depth': 1, 'min_samples_split': 11, 'min_samples_leaf': 18, 'max_features': 0.4917613280300664, 'learning_rate_model': 0.031397442469832317}
LSTM	8.86	{'batch_size': 48, 'learning_rate': 0.005177372559029542, 'max_epochs': 1400, 'num_workers': 7, 'hidden_size': 124, 'num_layers': 8, 'dropout': 0.709007457107967}
PatchMixer	**6.58**	**{'batch_size': 32, 'learning_rate': 0.0001120549333127051, 'max_epochs': 1700, 'num_workers': 5, 'patch_len': 32, 'stride': 11, 'mixer_kernel_size': 8, 'd_model': 256, 'dropout': 0.45, 'head_dropout': 0.15, 'e_layers': 3}**
Random Forest	14.54	{'batch_size': 48, 'learning_rate': 0.0001483174723491183, 'max_epochs': 1200, 'num_workers': 8, 'n_estimators': 63, 'max_depth': 13, 'min_samples_split': 2, 'min_samples_leaf': 19, 'max_features': 0.9794293248463268}
xPatch	6.72	{'batch_size': 32, 'learning_rate': 0.0001835073640022036, 'max_epochs': 1300, 'num_workers': 9, 'patch_len': 12, 'stride': 24, 'padding_patch': 'end', 'revin': 0, 'ma_type': 'ema', 'alpha': 0.839193453449457, 'beta': 0.7190804218731096}

Table A.1: Hyperparameter Tuning Results for Minimizing MAE with Bootstrap Sampling

Tuning minimizing MAE Without Bootstrap Sampling

Model	MAE	Parameters
AdaBoost	10.53	{'batch_size': 80, 'learning_rate': 0.008963332707659527, 'max_epochs': 1700, 'num_workers': 11, 'n_estimators': 61, 'learning_rate_model': 0.8546071447383281}
Gradient Boosting	8.20	{'batch_size': 32, 'learning_rate': 0.0006333956308102298, 'max_epochs': 1200, 'num_workers': 10, 'n_estimators': 144, 'max_depth': 18, 'min_samples_split': 15, 'min_samples_leaf': 4, 'max_features': 0.6592172883644293, 'learning_rate_model': 0.388634205880954}
GRU	6.69	{'batch_size': 96, 'learning_rate': 0.0024656915080670467, 'max_epochs': 1800, 'num_workers': 7, 'hidden_size': 91, 'num_layers': 1, 'dropout': 0.44611302927266927}
LSTM	6.64	{'batch_size': 80, 'learning_rate': 0.00544215587526865, 'max_epochs': 1900, 'num_workers': 9, 'hidden_size': 67, 'num_layers': 1, 'dropout': 0.27527815284264673}
PatchMixer	**6.5**0	**{'batch_size': 48, 'learning_rate': 0.00021128018387968383, 'max_epochs': 1200, 'num_workers': 8, 'patch_len': 32, 'stride': 9, 'mixer_kernel_size': 2, 'd_model': 128, 'dropout': 0.35, 'head_dropout': 0.0, 'e_layers': 1}**
Random Forest	9.76	{'batch_size': 80, 'learning_rate': 0.000737097364475848, 'max_epochs': 1800, 'num_workers': 10, 'n_estimators': 113, 'max_depth': 1, 'min_samples_split': 13, 'min_samples_leaf': 15, 'max_features': 0.1268005874199549}
xPatch	6.53	{'batch_size': 48, 'learning_rate': 0.00015659795416669107, 'max_epochs': 1000, 'num_workers': 6, 'patch_len': 12, 'stride': 36, 'padding_patch': 'None', 'revin': 0, 'ma_type': 'reg', 'alpha': 0.7972611256000964, 'beta': 0.17020757865248334}

Table A.2: Hyperparameter Tuning Results for Minimizing MAE without Bootstrap Sampling

Tuning minimizing MAE and Maximizing Recall With Bootstrap Sampling

Model	Recall	MAE	Parameters
AdaBoostRegressor	0.7021	30.17	{'batch_size': 64, 'learning_rate': 0.0009260691434031928, 'max_epochs': 1200, 'num_workers': 6, 'n_estimators': 132, 'learning_rate_model': 0.9313535667341897}
GradientBoostingRegressor	0.8109	38.05	{'batch_size': 48, 'learning_rate': 0.00010824735790287389, 'max_epochs': 1500, 'num_workers': 5, 'n_estimators': 150, 'max_depth': 20, 'min_samples_split': 19, 'min_samples_leaf': 2, 'max_features': 0.13710322525399174, 'learning_rate_model': 0.2557229816731754}
GRU	0.8083	**13.95**	{'batch_size': 64, 'learning_rate': 0.001552187907543553, 'max_epochs': 1700, 'num_workers': 9, 'hidden_size': 91, 'num_layers': 1, 'dropout': 0.7107218143412196}
LSTM	0.7591	14.29	{'batch_size': 32, 'learning_rate': 0.002763612267571373, 'max_epochs': 1400, 'num_workers': 12, 'hidden_size': 76, 'num_layers': 1, 'dropout': 0.22721670516047598}
PatchMixer	0.8342	16.55	{'batch_size': 48, 'learning_rate': 0.005415696015300989, 'max_epochs': 1800, 'num_workers': 5, 'patch_len': 16, 'stride': 12, 'mixer_kernel_size': 8, 'd_model': 832, 'dropout': 0.4, 'head_dropout': 0.0, 'e_layers': 2}
RandomForestRegressor	0.6813	32.04	{'batch_size': 48, 'learning_rate': 0.00016068210171536588, 'max_epochs': 1700, 'num_workers': 5, 'n_estimators': 93, 'max_depth': 16, 'min_samples_split': 13, 'min_samples_leaf': 1, 'max_features': 0.675461333214573}
xPatch	**0.8627**	16.99	{'batch_size': 32, 'learning_rate': 0.009375393931457811, 'max_epochs': 1800, 'num_workers': 11, 'patch_len': 48, 'stride': 24, 'padding_patch': 'None', 'revin': 0, 'ma_type': 'reg', 'alpha': 0.8205177177113407, 'beta': 0.5824502202238674}

Table A.3: Hyperparameter tuning minimizing MAE and maximizing Recall With Bootstrap Sampling

Tuning minimizing MAE and Maximizing Recall Without Bootstrap Sampling

Model	Rec	MAE	Parameters
AdaBoost	0.5430	21.11	{'batch_size': 128, 'num_workers': 10, 'learning_rate': 0.00033210742092950875, 'max_epochs': 1500, 'n_estimators': 56, 'learning_rate_model': 0.8292611812822679}
GradientBoosting	0.7151	28.02	{'batch_size': 128, 'num_workers': 10, 'learning_rate': 0.0013602132014801727, 'max_epochs': 1700, 'n_estimators': 303, 'max_depth': 5, 'min_samples_split': 20, 'min_samples_leaf': 6, 'max_features': 0.7422010599616656, 'learning_rate_model': 0.9777609683476102, 'subsample': 0.7288036971785529}
GRU	0.8145	**13.97**	{'batch_size': 256, 'num_workers': 10, 'learning_rate': 0.00416833788862982, 'max_epochs': 2000, 'hidden_size': 103, 'num_layers': 1, 'dropout': 0.47421195537747074}
LSTM	0.7339	14.13	{'batch_size': 48, 'num_workers': 10, 'learning_rate': 0.00238897789029602, 'max_epochs': 1700, 'hidden_size': 185, 'num_layers': 1, 'dropout': 0.3451288320034417}
PatchMixer	**0.9140**	23.05	{'batch_size': 192, 'num_workers': 10, 'learning_rate': 0.0029608318844730765, 'max_epochs': 1500, 'patch_len': 12, 'stride': 8, 'mixer_kernel_size': 8, 'd_model': 768, 'dropout': 0.8, 'head_dropout': 0.0, 'e_layers': 3}
RandomForest	0.5699	19.74	{'batch_size': 112, 'num_workers': 10, 'learning_rate': 0.00017945347215010435, 'max_epochs': 1700, 'n_estimators': 158, 'max_depth': 1, 'min_samples_split': 3, 'min_samples_leaf': 2, 'max_features': 0.29006409036536485}
xPatch	0.8172	14.23	{'batch_size': 32, 'num_workers': 10, 'learning_rate': 0.0022504039576051542, 'max_epochs': 1200, 'patch_len': 12, 'stride': 1, 'padding_patch': 'None', 'revin': 0, 'ma_type': 'reg', 'alpha': 0.7576358668280231, 'beta': 0.26169390850128227}

Table A.4: Results from tuning with minimizing MAE and maximizing recall Without Bootstrap Sampling

A.2.2 SDU Tuning Result

This section covers the full tuning results for the SDU dataset.

Tuning minimizing Huber loss With Bootstrap Sampling

Model	Huber loss	Parameters
AdaBoost	12.15	{'batch_size': 48, 'learning_rate': 0.005360475246066047, 'max_epochs': 1700, 'num_workers': 6, 'n_estimators': 111, 'learning_rate_model': 0.0166561782513406 6}
Gradient Boosting	11.35	{'batch_size': 112, 'learning_rate': 0.001321503620156157, 'max_epochs': 1700, 'num_workers': 9, 'n_estimators': 430, 'max_depth': 4, 'min_samples_split': 14, 'min_samples_leaf': 18, 'max_features': 0.8948724470619752, 'learning_rate_model': 0.5622454397802673, 'subsample': 0.5428375829157808}
GRU	2.71	{'batch_size': 256, 'learning_rate': 0.0084404811425464, 'max_epochs': 1700, 'num_workers': 6, 'hidden_size': 169, 'num_layers': 1, 'dropout': 0.19749305236112333}
LSTM	3.17	{'batch_size': 240, 'learning_rate': 0.006737826136406453, 'max_epochs': 1800, 'num_workers': 8, 'hidden_size': 168, 'num_layers': 1, 'dropout': 0.08926779431569574}
PatchMixer	3.31	{'batch_size': 256, 'learning_rate': 0.0004646414616809131, 'max_epochs': 2000, 'num_workers': 9, 'patch_len': 4, 'stride': 10, 'mixer_kernel_size': 2, 'd_model': 320, 'dropout': 0.4, 'head_dropout': 0.0, 'e_layers': 10}
Random Forest	13.34	{'batch_size': 64, 'learning_rate': 0.0002765831743953899, 'max_epochs': 1300, 'num_workers': 8, 'n_estimators': 194, 'max_depth': 2, 'min_samples_split': 15, 'min_samples_leaf': 19, 'max_features': 0.11993729200314268}
xPatch	**2.64**	**{'batch_size': 64, 'learning_rate': 0.0004865675722861067, 'max_epochs': 1400, 'num_workers': 14, 'patch_len': 6, 'stride': 1, 'padding_patch': 'end', 'revin': 0, 'ma_type': 'reg', 'alpha': 0.4689754211609729, 'beta': 0.6857634099892445}**

Table A.5: Hyperparameter Tuning Results for Minimizing Huber loss with Bootstrap Sampling

Tuning minimizing Huber loss without Bootstrap Sampling

Model	Huber loss	Parameters
AdaBoostRegressor	3.97	{'batch_size': 112, 'learning_rate': 0.001180923466722264, 'max_epochs': 1100, 'num_workers': 9, 'n_estimators': 151, 'learning_rate_model': 0.9143031551070804}
GradientBoostingRegressor	3.32	{'batch_size': 128, 'learning_rate': 0.000834870457404289 3, 'max_epochs': 1300, 'num_workers': 9, 'n_estimators': 325, 'max_depth': 2, 'min_samples_split': 2, 'min_samples_leaf': 12, 'max_features': 0.7905460467424213, 'learning_rate_model': 0.38415814166769624, 'subsample': 0.39729124539174876}
GRU	2.68	{'batch_size': 160, 'learning_rate': 0.008214250139084259, 'max_epochs': 1700, 'num_workers': 10, 'hidden_size': 179, 'num_layers': 1, 'dropout': 0.8081170085414112}
LSTM	3.16	{'batch_size': 192, 'learning_rate': 0.008405810836213588, 'max_epochs': 1800, 'num_workers': 12, 'hidden_size': 144, 'num_layers': 1, 'dropout': 0.624867070588652}
PatchMixer	3.40	{'batch_size': 64, 'learning_rate': 0.0008146399215302535, 'max_epochs': 1700, 'num_workers': 6, 'patch_len': 32, 'stride': 6, 'mixer_kernel_size': 6, 'd_model': 960, 'dropout': 0.4, 'head_dropout': 0.0, 'e_layers': 2}
RandomForestRegressor	4.23	{'batch_size': 80, 'learning_rate': 0.0024309462831591075, 'max_epochs': 2000, 'num_workers': 10, 'n_estimators': 111, 'max_depth': 16, 'min_samples_split': 3, 'min_samples_leaf': 3, 'max_features': 0.9596702162606467}
xPatch	**2.59**	**{'batch_size': 208, 'learning_rate': 0.0006554334484062519, 'max_epochs': 2000, 'num_workers': 8, 'patch_len': 16, 'stride': 5, 'padding_patch': 'None', 'revin': 0, 'ma_type': 'reg', 'alpha': 0.6803081419478867, 'beta': 0.7493277554345201}**

Table A.6: Hyperparameter Tuning Results for Minimizing Huber loss without Bootstrap Sampling

Tuning minimizing Huber Loss and maximizing Recall with Bootstrap Sampling

Model	Recall	Huber Loss	Parameters
AdaBoostRegressor	0.0	10.8374	{'batch_size': 256, 'num_workers': 10, 'learning_rate': 0.0025081870046723045, 'max_epochs': 1000, 'n_estimators': 54, 'learning_rate_model': 0.0584444870721746}
GRU	0.2222	**2.3371**	{'batch_size': 128, 'num_workers': 10, 'learning_rate': 0.0036149068236998, 'max_epochs': 1800, 'hidden_size': 156, 'num_layers': 1, 'dropout': 0.3070727620397399}
GradientBoostingRegressor	0.6111	20.1735	{'batch_size': 192, 'num_workers': 10, 'learning_rate': 0.0005934535032636055, 'max_epochs': 1400, 'n_estimators': 488, 'max_depth': 1, 'min_samples_split': 17, 'min_samples_leaf': 6, 'max_features': 0.6098031852726209, 'learning_rate_model': 0.9488819068920191, 'subsample': 0.9070970684413281}
LSTM	0.0	2.9810	{'batch_size': 48, 'num_workers': 10, 'learning_rate': 0.00696072774267959, 'max_epochs': 1800, 'hidden_size': 194, 'num_layers': 6, 'dropout': 0.2953120207344502}
PatchMixer	**1.0**	10.9357	{'batch_size': 256, 'num_workers': 10, 'learning_rate': 0.002739975780721063, 'max_epochs': 1300, 'patch_len': 16, 'stride': 8, 'mixer_kernel_size': 16, 'd_model': 512, 'dropout': 0.8, 'head_dropout': 0.0, 'e_layers': 5}
RandomForestRegressor	0.0	12.7543	{'batch_size': 256, 'num_workers': 10, 'learning_rate': 0.00287052409356019, 'max_epochs': 1000, 'n_estimators': 159, 'max_depth': 2, 'min_samples_split': 5, 'min_samples_leaf': 18, 'max_features': 0.9983553622788395}
xPatch	0.8889	3.0871	{'batch_size': 224, 'num_workers': 10, 'learning_rate': 0.004404840341980456, 'max_epochs': 1700, 'patch_len': 16, 'stride': 7, 'padding_patch': 'None', 'revin': 0, 'ma_type': 'reg', 'alpha': 0.6397468821662603, 'beta': 0.2902473814947507}

Table A.7: Results from tuning with minimizing Huber loss and maximizing Recall with Bootstrap Sampling

Tuning minimizing Huber Loss and Maximizing Recall without Bootstrap Sampling

Model	Recall	Huber Loss	Parameters
AdaBoostRegressor	0.0556	3.9520	{'batch_size': 64, 'num_workers': 10, 'learning_rate': 0.00017729405605056275, 'max_epochs': 1400, 'n_estimators': 58, 'learning_rate_model': 0.9722194994346534}
GradientBoostingRegressor	0.5	4.8077	{'batch_size': 224, 'num_workers': 10, 'learning_rate': 0.0009265401570267737, 'max_epochs': 1400, 'n_estimators': 338, 'max_depth': 3, 'min_samples_split': 19, 'min_samples_leaf': 14, 'max_features': 0.13996840741548092, 'learning_rate_model': 0.8208930394073057, 'subsample': 0.8785629204400206}
LSTM	0.0	2.9963	{'batch_size': 192, 'num_workers': 10, 'learning_rate': 0.00526833387860358, 'max_epochs': 1600, 'hidden_size': 164, 'num_layers': 1, 'dropout': 0.9297010312416186}
PatchMixer	1.0	8.4945	{'batch_size': 144, 'num_workers': 10, 'learning_rate': 0.0008285155531289423, 'max_epochs': 1400, 'patch_len': 8, 'stride': 14, 'mixer_kernel_size': 16, 'd_model': 320, 'dropout': 0.8, 'head_dropout': 0.1, 'e_layers': 4}
RandomForestRegressor	0.0	3.9177	{'batch_size': 176, 'num_workers': 10, 'learning_rate': 0.0014413815103835393, 'max_epochs': 1300, 'n_estimators': 144, 'max_depth': 9, 'min_samples_split': 5, 'min_samples_leaf': 5, 'max_features': 0.8399588878134902}
xPatch	**0.9444**	3.3034	{'batch_size': 96, 'num_workers': 10, 'learning_rate': 0.004895508586161393, 'max_epochs': 1900, 'patch_len': 14, 'stride': 7, 'padding_patch': 'None', 'revin': 0, 'ma_type': 'reg', 'alpha': 0.9217499028959255, 'beta': 0.29425111586615627}
GRU	0.3333	**2.5610**	{'batch_size': 224, 'num_workers': 10, 'learning_rate': 0.00874224191832517, 'max_epochs': 1700, 'hidden_size': 156, 'num_layers': 3, 'dropout': 0.31926899758389626}

Table A.8: Results from tuning with minimizing Huber loss and maximizing Recall without Bootstrap Sampling

Bibliography

[1] M. H. Hassoun, "Multi-layer perceptrons," in *Fundamentals of Artificial Neural Networks*, Springer, 2020, pp. 93–124. DOI: 10.1007/978-3-030-42227-1_5. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-42227-1_5.

[2] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015. [Online]. Available: https://arxiv.org/abs/1511.08458.

[3] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," *arXiv preprint arXiv:1610.02357*, 2017. [Online]. Available: https://arxiv.org/abs/1610.02357.

[4] B.-S. Hua, M.-K. Tran, and S.-K. Yeung, "Pointwise convolutional neural networks," *arXiv preprint arXiv:1712.05245*, 2018. [Online]. Available: https://arxiv.org/abs/1712.05245.

[5] R. J. Nemati *et al.*, "A framework for classification of gabor based frequency selective bone radiographs using cnn," *Arabian Journal for Science and Engineering*, vol. 46, no. 4, pp. 4141–4152, Apr. 2021, ISSN: 2191-4281. DOI: 10.1007/s13369-021-05339-7. [Online]. Available: https://doi.org/10.1007/s13369-021-05339-7.

[6] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997. DOI: 10.1162/neco.1997.9.8.1735.

[7] R. Al-Douri, M. Y. Azeez, H. A. Jalab, J. Abawajy, and Z. Mohd Saaya, "A hybrid deep learning model for load forecasting based on multi-head attention mechanism and 1d-cnn," *Information*, vol. 15, no. 9, p. 517, 2024. DOI: 10.3390/info15090517. [Online]. Available: https://www.mdpi.com/2078-2489/15/9/517.

[8] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, *Empirical evaluation of gated recurrent neural networks on sequence modeling*, 2014. arXiv: 1412.3555. [Online]. Available: https://arxiv.org/abs/1412.3555.

[9] D. I. Mienye and Y. Sun, "A survey of ensemble learning: Concepts, algorithms, applications, and prospects," *arXiv preprint arXiv:2412.17323*, 2022. DOI: 10.1109/ACCESS.2022.3207287. [Online]. Available: https://ieeexplore.ieee.org/document/9893798.

[10] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001. DOI: 10.1023/A:1010933404324.

[11] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[12] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *Annals of Statistics*, vol. 29, no. 5, pp. 1189–1232, 2001. DOI: 10.1214/aos/1013203451.

[13] IEA, *Outlook for electric vehicle charging infrastructure*, 2024. [Online]. Available: https://www.iea.org/reports/global-ev-outlook-2024/outlook-for-electric-vehicle-charging-infrastructure.

[14] U. D. of Energy's Vehicle Technologies Office, *Electric vehicle charging stations*, 2025. [Online]. Available: https://afdc.energy.gov/fuels/electricity-stations.

[15] U. D. of Transportation, *Charger types and speeds*, 2025. [Online]. Available: https://www.transportation.gov/rural/ev/toolkit/ev-basics/charging-speeds.

[16] T. A. Short, *Electric Power Distribution Handbook*, 2nd ed. Boca Raton Florida: CRC Press, 2014. DOI: 10.1201/b16747.

[17] U. D. of Energy, *Estimating appliance and home electronic energy use*, 2025. [Online]. Available: https://www.energy.gov/energysaver/estimating-appliance-and-home-electronic-energy-use.

[18] U. D. of Energy, *How it works: Electric transmission & distribution and protective measures*, 2023. [Online]. Available: https://www.energy.gov/sites/default/files/2023-11/FINAL_CESER%20Electricity%20Grid%20Backgrounder_508.pdf.

[19] Y. Li and A. Jenn, *Impact of electric vehicle charging demand on power distribution grid congestion*, 2024. [Online]. Available: https://doi.org/10.1073/pnas.2317599121.

[20] N. Panossian, M. Muratori, B. Palmintier, A. Meintz, T. Lipman, and K. Moffat, "Challenges and opportunities of integrating electric vehicles in electricity distribution systems," *Current Sustainable/Renewable Energy Reports*, vol. 9, no. 2, pp. 27–40, 2022. DOI: 10.1007/s40518-022-00201-2. [Online]. Available: https://doi.org/10.1007/s40518-022-00201-2.

[21] S. Electric, *Why is the transformer rating in kva instead of kw?* 2025. [Online]. Available: https://eshop.se.com/in/blog/post/why-is-the-transformer-rating-in-kva-instead-of-kw.html.

[22] IEEE Power and Energy Society, "Ieee guide for loading mineral-oil-immersed transformers and step-voltage regulators," *IEEE Std C57.91-2011 (Revision of IEEE Std C57.91-1995)*, pp. 1–123, 2012. DOI: 10.1109/IEEESTD.2012.6166928.

[23] M. Aldossary, H. A. Alharbi, and N. Ayub, "Optimizing electric vehicle (ev) charging with integrated renewable energy sources: A cloud-based forecasting approach for eco-sustainability," *Mathematics*, vol. 12, no. 17, 2024, ISSN: 2227-7390. DOI: 10.3390/math12172627. [Online]. Available: https://www.mdpi.com/2227-7390/12/17/2627.

[24] A. Jadon, A. Patil, and S. Jadon, "A comprehensive survey of regression based loss functions for time series forecasting," [Online]. Available: https://arxiv.org/abs/2211.02989.

[25] P. J. Huber, "Robust Estimation of a Location Parameter," *The Annals of Mathematical Statistics*, vol. 35, no. 1, pp. 73–101, 1964. DOI: 10.1214/aoms/1177703732. [Online]. Available: https://doi.org/10.1214/aoms/1177703732.

[26] Google. "Classification: Accuracy, recall, precision, and related metrics." Google Machine Learning Crash Course. (2025), [Online]. Available: https://developers.google.com/machine-learning/crash-course/classification/accuracy-precision-recall (visited on 05/20/2025).

[27] Y. Kim and S. Kim, "Forecasting charging demand of electric vehicles using time-series models," *Energies*, vol. 14, no. 5, p. 1487, 2021. DOI: 10.3390/en14051487. [Online]. Available: https://www.mdpi.com/1996-1073/14/5/1487.

[28] M. Boulakhbar, M. Farag, K. Benabdelaziz, T. Kousksou, and M. Zazi, "A deep learning approach for prediction of electrical vehicle charging stations power demand in regulated electricity markets: The case of morocco," *Cleaner Energy Systems*, vol. 3, p. 100 039, 2022. DOI: 10.1016/j.cles.2022.100039. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2772783122000371.

[29] J. Zhu, Z. Yang, Y. Guo, J. Zhang, and H. Yang, "Short-term load forecasting for electric vehicle charging stations based on deep learning approaches," *Applied Sciences*, vol. 9, 2019. DOI: 10.3390/app9091723. [Online]. Available: https://www.mdpi.com/2076-3417/9/9/1723.

[30] M. Chang, S. Bae, G. Cha, and J. Yoo, "Aggregated electric vehicle fast-charging power demand analysis and forecast based on lstm neural network," *Sustainability*, vol. 13, 2021. DOI: 10.3390/su132413783. [Online]. Available: https://www.mdpi.com/2071-1050/13/24/13783.

[31] A. Ostermann and T. Haug, "Probabilistic forecast of electric vehicle charging demand: Analysis of different aggregation levels and energy procurement," *Energy Informatics*, vol. 13, 2024. DOI: 10.1186/s42162-024-00319-1. [Online]. Available: https://energyinformatics.springeropen.com/articles/10.1186/s42162-024-00319-1.

[32] C. Hecht, J. Figgener, and D. U. Sauer, "Predicting electric vehicle charging station availability using ensemble machine learning," *Energies*, vol. 14, 2021. DOI: 10.3390/en14237834. [Online]. Available: https://www.mdpi.com/1996-1073/14/23/7834.

[33] A. Almaghrebi, F. Aljuheshi, M. Rafaie, K. James, and M. Alahmad, "Data-driven charging demand prediction at public charging stations using supervised machine learning regression methods," *Energies*, vol. 13, 2020. DOI: 10.3390/en13164231. [Online]. Available: https://www.mdpi.com/1996-1073/13/16/4231.

[34] F. Ren, C. Tian, G. Zhang, C. Li, and Y. Zhai, "A hybrid method for power demand prediction of electric vehicles based on sarima and deep learning with integration of periodic features," *Energy*, vol. 250, p. 123 738, 2022, ISSN: 0360-5442. DOI: https://doi.org/10.1016/j.energy.2022.123738. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360544222006417.

[35] paperswithcode, *Time series forecasting*, 2025. [Online]. Available: https://paperswithcode.com/task/time-series-forecasting.

[36] H. Zhou *et al.*, "Informer: Beyond efficient transformer for long sequence time-series forecasting," in *The Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Virtual Conference*, vol. 35, AAAI Press, 2021, pp. 11 106–11 115.

[37] G. Lai, *Multivariate time series data sets*, 2017. [Online]. Available: https://github.com/laiguokun/multivariate-time-series-data.

[38] M. P. I. for Biogeochemistry, *Jena climate dataset*, 2025. [Online]. Available: https://www.bgc-jena.mpg.de/wetter/.

[39] X. Yuan and L. Chen, "D-pad: Deep-shallow multi-frequency patterns disentangling for time series forecasting," *arXiv preprint arXiv:2403.17814*, 2024. DOI: 10.48550/arXiv.2403.17814.

[40] A. Stitsyuk and J. Choi, "Xpatch: Dual-stream time series forecasting with exponential seasonal-trend decomposition," *arXiv preprint arXiv:2412.17323*, 2024. DOI: 10.48550/arXiv.2412.17323. [Online]. Available: https://arxiv.org/abs/2412.17323.

[41] Y. Nie, N. H. Nguyen, P. Sinthong, and J. Kalagnanam, "A time series is worth 64 words: Long-term forecasting with transformers," *ICLR 2023*, 2023. DOI: 10.48550/arXiv.2211.14730.

[42] Z. Gong, Y. Tang, and J. Liang, "Patchmixer: A patch-mixing architecture for long-term time series forecasting," *arXiv preprint arXiv:2310.00655*, 2023. DOI: 10.48550/arXiv.2310.00655. [Online]. Available: https://arxiv.org/abs/2310.00655.

[43] C. City of Boulder, *Electric vehicle charging station data*, 2023. [Online]. Available: https://open-data.bouldercolorado.gov/datasets/95992b3938be4622b07f0b05eba95d4c_0/about.

[44] K. Christensen, *Multi-agent based simulation framework for evaluating digital energy solutions and adoption strategies*, 2022. [Online]. Available: https://doi.org/10.21996/kksb-v823.

[45] U.S. Energy Information Administration (EIA). "Colorado state energy profile." Accessed: 2025-05-31. (2024), [Online]. Available: https://www.eia.gov/state/analysis.php?sid=CO.

[46] R. Han and L. Yu, "Evaluate and compare machine learning models for temperature forecasting of meteorological data in hohhot," *IEEE*, 2023. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10386247.

[47] T. Mahajan, G. Singh, and G. Bruns, "An experimental assessment of treatments for cyclical data," Accessed: 2025-05-21, Master's thesis, California State University Monterey Bay, 2021. [Online]. Available: https://scholarworks.calstate.edu/downloads/pv63g5147.

[48] Meteostat, *The weather's record keeper*, https://meteostat.net/en/, Accessed: 2025-05-31.

[49] PyTorch, *Pytorch*, 2025. [Online]. Available: https://pytorch.org/projects/pytorch/.

[50] L. AI, *Welcome to pytorch lightning*, 2025. [Online]. Available: https://lightning.ai/docs/pytorch/stable/.

[51] scikit-learn, *Scikit-learn*, 2025. [Online]. Available: https://scikit-learn.org/stable/.

[52] Optuna, *Optimize your optimization*, 2023. [Online]. Available: https://optuna.org/#key_features.

[53] Optuna, *Efficient optimization algorithms*, 2025. [Online]. Available: https://optuna.readthedocs.io/en/stable/tutorial/10_key_features/003_efficient_optimization_algorithms.html.

[54] P. Lightning, *N-bit precision (intermediate)*, 2025. [Online]. Available: https://lightning.ai/docs/pytorch/stable/common/precision_intermediate.html.

[55] L. AI, *Ddpstrategy*, 2025. [Online]. Available: https://lightning.ai/docs/pytorch/stable/api/lightning.pytorch.strategies.DDPStrategy.html#lightning.pytorch.strategies.DDPStrategy.

[56] L. AI, *Gpu training (intermediate)*, 2025. [Online]. Available: https://lightning.ai/docs/pytorch/stable/accelerators/gpu_intermediate.html#distributed-data-parallel.

[57] CLAAUDIA, *Ai-lab*, 2024. [Online]. Available: https://hpc.aau.dk/ai-lab/.