Vrije Universiteit Amsterdam

Bachelor Thesis

Real-time Monitoring and Predictive Modeling for Space Occupancy

Author: Mikita Volakh (2691857)

1st supervisor:	drs. Kees Verstoep
daily supervisor:	drs. Kees Verstoep
2nd reader:	Prof. dr. ir. Henri E. Bal

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

December 19, 2023

Abstract

The Internet of Things (IoT) refers to a network of interconnected sensors that gather and exchange data, enabling remote access to monitoring, automation, and data analysis. Within the IoT domain, amidst the growing scale of incoming data and the demand for precise predictive analysis, this paper focuses on presenting a system aimed at visualizing real-time sensor data coupled with predictions of future trends in terms of room occupancy.

The presented system utilizes data accumulated from a number of sensors located on several floors of the NU building at Vrije Universiteit Amsterdam. Additionally, it provides classification of room occupancy status based on a number of metrics, such as CO2, noise, and light intensity levels. To find the most accurate prediction model, a comparative analysis between Long Short-Term Memory (LSTM), Gated Recurrent Unit (GRU), and Convolutional Neural Network (CNN) architectures is conducted, where these models are evaluated based on their accuracy in forecasting room occupancy trends using historical sensor data.

Contents

1

Introduction

1.1 Context

The Internet of Things provides various solutions in a number of sectors including healthcare (1), logistics and supply chains (2), manufacturing (3), agriculture (4) and urban infrastructure (5) (6). We propose a room occupancy monitoring system that combines real-time data with predictive analytics, integrating the designs and principles utilized across these diverse domains within the Internet of Things landscape. Combining precise predictive and observational methods from healthcare, data-driven approaches used in logistics, and visualization techniques used in urban development, we aim to create a comprehensive, scalable, and adaptive IoT framework.

Although our system is in principle designed to be used as a diagnostic and monitoring tool, it retains adaptability to be used by other parties, such as staff members or students that are in need of the system to find unoccupied spaces. In this thesis, we address multiple challenges in designing such a system.

The presented system was specifically tailored for use within the NU building at Vrije Universiteit Amsterdam. However, it can be generalized for use in other environments that make use of similar architecture and data accumulated from a network of sensors. The system holds the potential to enable advancements in the following domains:

- Resource optimization

  Monitoring diverse factors like CO2 levels, temperature, and illumination offers insights into resource usage, including power consumption, heating, and air conditioning. Analyzing these data points enables the identification of inefficient patterns, which allows optimization of operations and improves cost efficiency. By understanding how these factors come together, organizations can take strategic measures to minimize waste of resources and enhance sustainability.

- Building management

  Recognizing and keeping track of trends and patterns over time offers smarter decision-making in terms of building management. Based on the data usage, the periods when the spaces tend to be least occupied could be identified, facilitating an optimization of operations that demand vacant spaces, such as room maintenance or cleaning schedules.

- Occupants welfare

  Adjusting environmental conditions based on occupancy patterns can improve productivity and well-being. Analyzing data coming from IoT devices that are integrated into office spaces can introduce optimal conditions for enhanced work performance and increased employee satisfaction with their overall quality of life. (7) This also allows for a personalized approach to setting up individual offices, matching each person's preferences.

- Building safety

  Continuous monitoring of room occupancy in real time plays a central role in enhancing building safety. It enables prompt identification of irregular or unforeseen trends, allowing for quick identification of emergencies. Interpreting the data offers diverse insights; for instance, a sudden, rapid increase in carbon dioxide levels or organic gas concentrations could indicate a potential fire event.

1.2 Problem Statement

This project addresses the need for holistic tools capable of addressing several concerns surrounding resource optimization, building management enhancement, and reinforcing safety measures. While similar systems exist (8), our solution focuses on enhancing data processing efficiency and the introduction of a robust prediction component, filling a gap in existing solutions that primarily rely on real-time data. The emphasis on improving data handling efficiency and incorporating predictive analytics offers a more comprehensive approach to managing resources and optimizing space utilization. Hence, the successful deployment of a system that merges predictive analytics with live data monitoring promises smarter building management, optimization of resources, and efficient sensor diagnostics.

1.3 Research Questions

This thesis addresses the problem by creating a monitoring tool that integrates real-time data monitoring with predictive analytics. Specifically, the project aims to develop a complete system that handles incoming live data streams from MQTT sensors across various floors of a university building, offering users access to a visual representation of real-time statistics and room occupancy status. Moreover, the predictive component involves the implementation of neural network models to accurately forecast room occupancy for the upcoming hours based on patterns observed in historical data collected over a recent time frame. The crucial technical challenge here requires making useful predictions of future trends and occupancy status classification, as well as the successful merging of this predictive element with the real-time monitoring part.

The following research questions are addressed within the problem's framework:

- How can a system be designed to integrate predictive room occupancy models with real-time monitoring, and what challenges might arise in implementing such an architecture?
- How does the predictive accuracy of LSTM, GRU, and 1D CNN neural network models differ when forecasting room occupancy trends based on historical sensor data?

1.4 Research Methodology

The methodology for developing the application involved the creation of a functional prototype aimed at uncovering potential challenges in integrating a real-time monitoring component with predictive models. Initially, a server infrastructure was established to efficiently handle incoming MQTT data streams, ensuring efficient processing, storage, and access of real-time data within the database. Subsequently, a RESTful API was constructed to communicate data with the clients. This was followed by the development of client-side components designed to visualize and display crucial environmental parameters.

Throughout the construction of the neural network models, comprehensive data preprocessing was executed. This preprocessing phase was essential in transforming historical data into a format tailored to meet the specific requirements of our predictive models.

To evaluate the predictive neural network models employed for time-series forecasting, a comprehensive regression analysis was conducted, utilizing regression error metrics such as mean squared error (MSE), mean absolute error (MAE), and Huber loss functions. These metrics were the primary models' performance estimators.

1.5 Thesis Contributions

This thesis aims to contribute as follows:

- Development of an interactive, user-friendly web application designed for real-time monitoring of sensor readings, allowing users to access and interpret environmental metrics such as carbon dioxide, noise, and light levels in a visual representation.
- Introduction of a scalable document-oriented architecture to facilitate the efficiency of data retrieval and storage. We achieve it by implementing an asynchronous, non-blocking infrastructure capable of handling vast amounts of data. We utilize a non-relational database that is specifically tailored for the specific requirements of the stored data and its type, effectively allowing us to store large volumes of incoming sensor data.
- Advancement in predictive capabilities through the integration of various time-series prediction neural network models with the real-time monitoring component. These models are tailored to forecast future environmental parameters. The integration allows the user to observe future trends in environmental changes.
- Evaluation of the presented time-series prediction neural network models. The evaluation process aims to validate the reliability and performance of these models in forecasting environmental parameters.

1.6 Plagiarism Declaration

I confirm that this thesis is my own work, is not copied from any other source, and has not been submitted elsewhere for assessment. All sources used in this paper are properly cited and acknowledged.

1.7 Thesis Structure

So far, this paper has introduced the context and societal impact of our project within its domain, as well as stated the problems we would like to address. Each upcoming chapter dives into specific aspects, offering more detailed insights, explanations, and evidence to substantiate our findings.

In Chapter 2, we navigate through the methodologies, contributions, and limitations employed by similar systems in the field of IoT. This provides additional context for developing an efficient system to address the stated problems.

Chapter 3 delves into the foundational aspects of the system, providing an elaborate description of the utilized sensors and the MQTT protocol used to communicate the data between the sensors and the server.

Chapter 4 provides an in-depth overview of a system design, mainly focusing on communications between different system's components. Additionally, it elaborates on the motivation behind using LSTM, GRU and CNN neural network models in the context of the presented problem.

Chapter 5 offers detailed explanations about how everything is put into action. It discusses the server setup (including handling of MQTT packets, how data is stored in the database, predicting future trends at scheduled times, and managing API routes) and the client implementation (creating the Graphical User Interface, getting real-time updates from the server, and making calls using RESTful API). This chapter also covers the implementation of time-series prediction neural network models. It explores the fine-tuning of their hyperparameters and the preparatory steps taken with the data. Additionally, it reveals the underlying process of classifying room occupancy status based on the collected metrics.

In Chapter 6, we evaluate the prediction neural network models by conducting a regression analysis, employing key error metrics such as Mean Absolute Error (MAE), Mean Squared Error (MSE) and the Huber loss function. Additionally, we delve into the limitations affecting the overall system, highlighting potential constraints and challenges that may impact the system's performance and reliability. Moreover, we discuss potential threats to the validity of our findings, addressing factors that could potentially influence the scalability and generalizability of our system.

In Chapter 7, we draw conclusions based on our findings. We present the results of the prediction model evaluation and address the discovered limitations. Furthermore, we discuss potential future improvements aimed at enhancing the system.

2

Related Work

There are several existing systems that use similar approaches and share analogous features with our proposed design.

ViMo, a system presented by Louisville (8), is a real-time monitoring application designed to enhance room occupancy visualization and present historical statistics of CO2, noise, and light levels through charts and graphs. For that purpose, it utilizes a subset of the dataset we use for creating our system. Real-time incoming data is communicated with the frontend via REST API, employing a polling mechanism that is performed every two seconds. Since ViMo is designed to produce statistics over the last 7-day period, the incoming MQTT data is stored in memory rather than being saved in the database or logged in any format. While ViMo prioritizes improving the user experience, our proposed design is centered around the creation of a more powerful and robust tool for monitoring and analyzing sensor data in terms of scalability and usability. Moreover, we suggest several predictive models to make environmental data forecasts as well as providing a classification algorithm to determine room occupancy status. In contrast to ViMo's multi-threaded message queueing design, our solution advocates for an asynchronous, non-blocking solution. We provide an additional medium to communicate real-time data by establishing a socket connection instead of solely relying on REST API requests.

A similar system suggested by Atmoko et al. (9) is designed to monitor humidity and temperature sensor readings in real-time. Rather than delivering a comprehensive system, it primarily focuses on conducting a comparative analysis of HTML and MQTT protocols regarding their performance in handling sensor data. The system operates on a small scale, designed to observe real-time values from a single sensor. Its sole function is to display how humidity and temperature levels have changed over time in the form of a line chart.

A study conducted by Jara Ochora et al. (10) is aimed at comparing MQTT and HTML protocols in terms of the underlying platform's power consumption. They introduce a digital dashboard that provides monitoring of temperature and humidity values from multiple sensors along with their spatial mapping. The primary objective of this study is to provide a comparative analysis of HTML and MQTT protocols in terms of power consumption under different loads. Additionally, they present a predictive model for battery power usage and its lifecycle based on network traffic. Although this research generally focuses on optimizing hardware, it offers a wider context, illustrating potential enhancements for the overall system and communication among its integral components, and provides an alternative perspective to predictive analysis predicting power consumption rather than metric values.

Khedkar et al. present a paper that seeks to discover the most effective predictive model used for forecasting IoT traffic (11). It explores several architectures involving machine learning, deep learning, and statistical time series-based prediction methods such as LSTM, ARIMA, VARMA, and feed-forward neural networks. The experiments conducted in this study show that LSTM and feed-forward networks provide superior accuracy in terms of predicting future time series values compared to statistical models. While the presented prediction models are tailored towards forecasting IoT traffic, a similar approach could be used to predict IoT sensor reading values.

3

Background

3.1 Hardware

At the core of the system is a network comprising interlinked sensors positioned across multiple floors (7th, 10th, 11th and 12th) within the NU building at Vrije Universiteit Amsterdam. A total of 43 Nordic Thingy:52 devices form the backbone of the project.

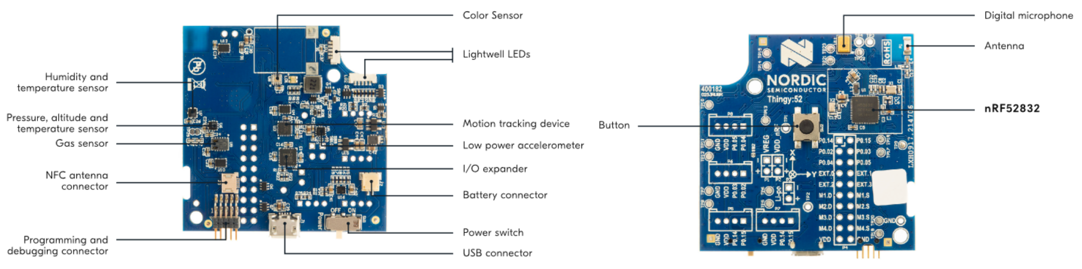


Figure 3.1: Printed Circuit Board (PCB) of the Nordic Thingy:52 device, showcasing sensor placements, connectors, switches, and other key components.(12)

There are various environment sensors present within the Nordic Thingy:52 platform that track miscellaneous metrics relevant to our system:

- CCS811 air quality sensor

  Measures total volatile organic compounds (TVOC) and carbon dioxide concentrations in ppm units.

- HTTS221 humidity and temperature sensor

  A combined sensor recording the relative humidity ranging from 0 to 100 percent and the temperature in Celsius.

- BH1745 color sensor

  Determines light intensity levels for red, green and blue colors in lux units.

Additionally, for noise level detection, the Nordic Thingy:52 devices are equipped with a built-in microphone utilizing pulse density modulation to indicate the current noise level (12).

To transmit the sensor recordings to the MQTT broker, a Bluetooth Low Energy (BLE) protocol is used. Two BLE channel types are available: the advertising channel, used for broadcast transmission, and the data channel, which supports two-way communication between linked devices (13). Sensors that are used to collect the data for our framework are configured to work with the advertising channel. The BLE packet's advertising unit carries a 37-byte payload, with 6 bytes reserved for the address of the advertising device. The remaining space is allocated for transmitting advertisement data, which includes the sensor's recordings. Due to the limited packet's size, which is insufficient to transfer the entire variety of sensor data, light intensity values are sent in a separate message to the MQTT server. This creates a need for processing two message types carrying different data and merging them afterwards to restore the full packet.

3.2 MQTT Protocol

Message Queuing Telemetry Transport (MQTT) is a network protocol particularly designed for communicating data between IoT devices and is built on top of the TCP/IP protocol (14). It follows the publisher-subscriber model, where after establishing a connection, clients are able to subscribe to specific topics (or all at once) and transmit or receive messages associated with that topic from or to the MQTT broker. The MQTT broker serves as an intermediary between publishers and the subscribed devices they communicate with.

The MQTT protocol can be configured to operate using different Quality of Service (QoS) levels that vary in their approach to message transfer and retrieval. For example, QoS 0 ensures that the message is sent at most once without the need for the acknowledgement packet. This provides the fastest possible communication and results in the minimum overhead. QoS 1, on the contrary, verifies that the packets are acknowledged upon arrival, so that each packet is sent at least once. QoS 2 guarantees that the packet is transmitted exactly once by performing a two-way handshake.

The MQTT broker can also be set up to store the most recent published messages within each topic. These messages, which are usually referred to as "retained" messages, are sent to the clients upon connection establishment so that they receive the latest data without waiting for the next message in each topic to be published.

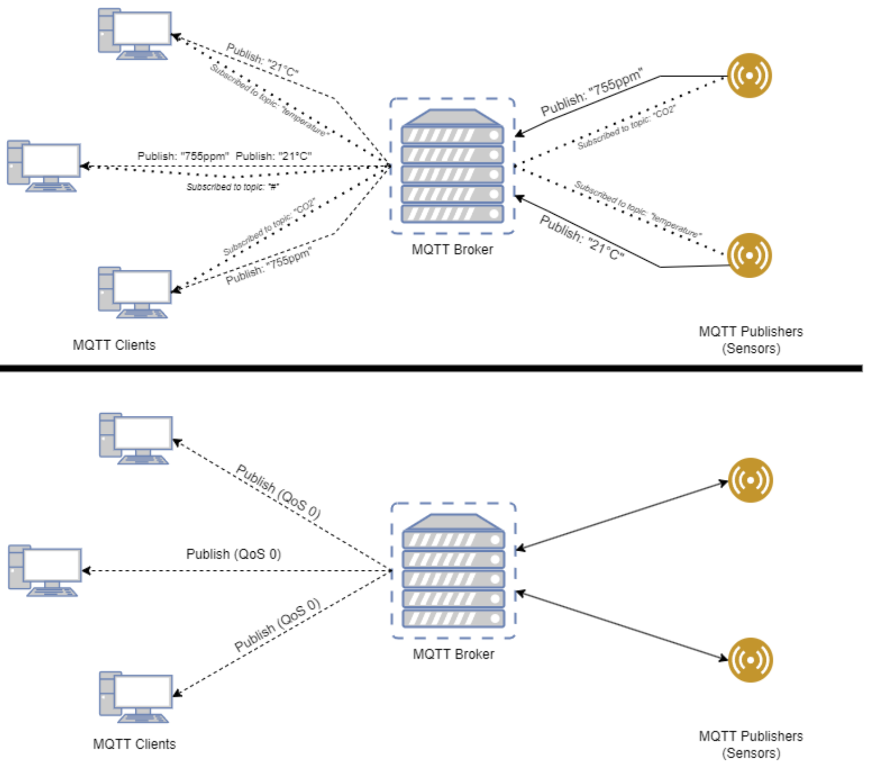


Figure 3.2: MQTT protocol illustrating the publisher-subscriber methodology.

Our system benefits from the MQTT protocol, which operates on the QoS 0 level (meaning clients do not acknowledge the incoming messages). Additionally, the MQTT broker does not preserve the retained messages, implying that connecting clients know nothing about the current state of the sensors. In particular, our system's backend monitors one specific topic, which delivers two types of combined packets containing various metrics belonging to individual sensors.

4

Design Overview

4.1 System Architecture

The system supports two distinct data transfer methods: one through REST API requests and the other via a socket connection. On the initial application launch, the latest stored state for each sensor or room is loaded through the API call. Live updates are intended to be transmitted through the socket connection. This approach offers advantages over using

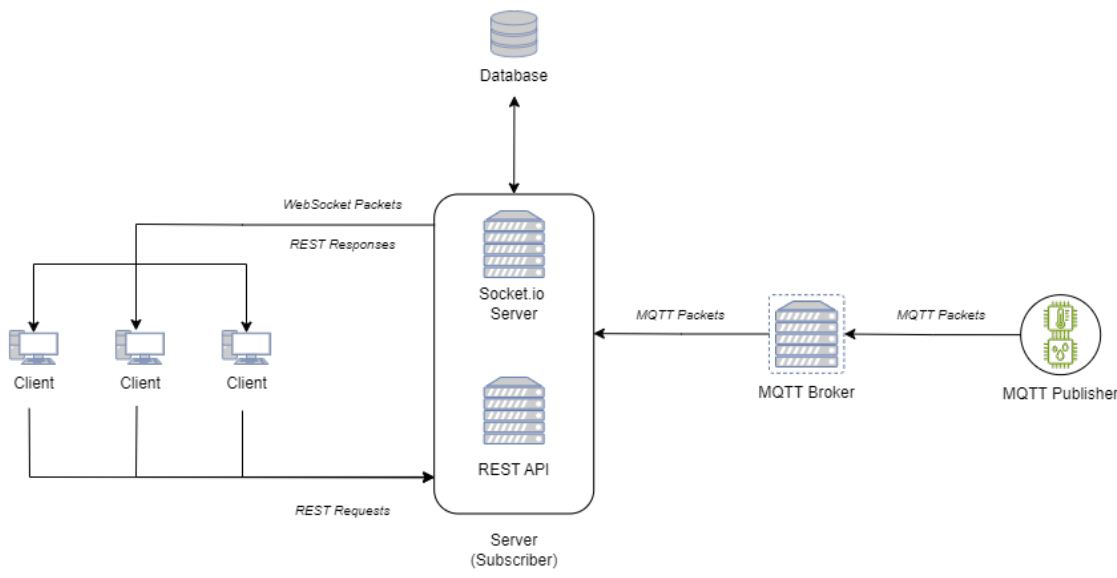
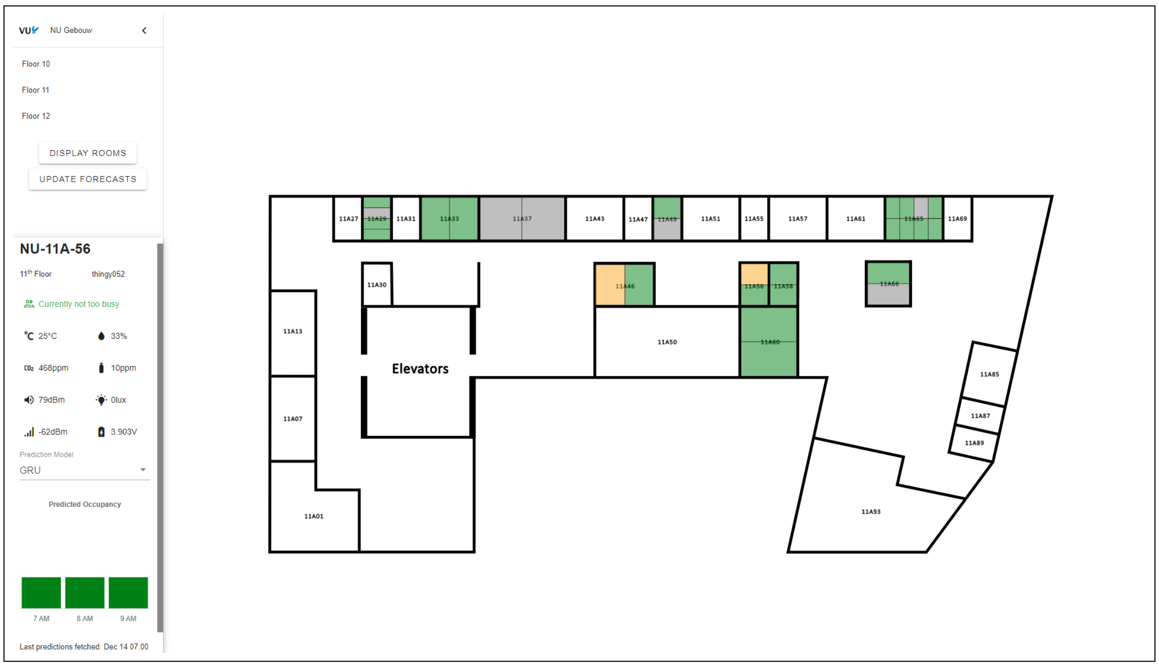


Figure 4.1: Overview of a system architecture displaying communication between IoT devices, MQTT broker, server, database, and clients.

a polling mechanism when data is requested at equal time intervals. It minimizes overhead by updating specific sensor states instead of requesting data for all sensors. Additionally, in instances where no updates occur, continuous polling would repeatedly request the same data. Implementing a socket-based connection ensures that data transfer is entirely managed from the system's backend.

The client's user interface (UI) is designed to allow for floor selection. Since there are several cases of multiple sensors being present in a single room, users have the option to toggle between displaying either rooms or areas covered by particular sensors. To improve visibility, it is possible to zoom in or out on the floor plan within the UI.



NU-11A-56

11th Floor thingy052

Currently not too busy

25°C 33%

468ppm 10ppm

79dBm 0lux

-62dBm 3.903V

Prediction Model: GRU

Predicted Occupancy

7 AM 8 AM 9 AM

Last predictions fetched: Dec 14 07:00

Figure 4.2: Desktop web application user interface.

The UI also offers a side bar displaying various metrics based on the selected room or sensor, along with the timestamp of the last data retrieval. Hovering over metric values makes the pop-up explanation of what the metric represents show up. Moreover, the component showcases a chart presenting predictions for the upcoming 3 hours, depending on the chosen room or sensor and the availability of predictions.

In addition, the system includes an alternative design customized for mobile devices, mirroring the features available in the desktop version.

4.2 Prediction Models Selection

Several models were considered for time-series prediction. Studies highlighted the effectiveness of recurrent neural networks (RNN) in addressing such challenges and their precedence over statistical models such as ARIMA in terms of predicting capabilities (15). As a result, the LSTM and the GRU models were selected.

These RNN models were particularly suitable for our scenario due to their ability to capture long-standing temporal dependencies in data (16), enabling predictions of unseen time-series data. LSTM and GRU models are widely used and applied in various contexts demanding regression analysis, such as predicting stock market trends (17) (18), power consumption levels (19) or SARS-CoV-2 fatalities (20).

There are additional model types that are applicable for time-series forecasting other than RNNs. Convolutional neural network (CNN) models possess the capacity to independently learn and extract features from raw input data, making them suitable for addressing time-series forecasting problems. CNN models are able to make sense of the temporal data by treating a sequence of lagged observations in a format similar to how one-dimensional images are represented. Guessoum et al. illustrate an example application of CNNs for time-series forecasting (21), where the 1D CNN model is used for predicting geodesic and astrometric parameters.

Initially, to keep the codebase consistent, a JavaScript library for creating neural networks called Brain.js was used. After preprocessing the data, the LSTM model was constructed and trained. However, due to the absence of GPU-accelerated training of LSTM networks in Brain.js, the training process was exceptionally time-consuming. Consequently, a shift towards a Python-based solution was made, integrating the prediction script into the server's workflow.

5

System-wide Implementation Overview

This chapter provides an in-depth overview of the implementation of our system, offering insights into the system's workflow from a technical standpoint. The project's codebase is available in the corresponding GitHub repository (22).

5.1 Frontend Setup and Implementation

The frontend refers to the incorporation of different elements and the interactions between them within the UI. This was achieved with the help of the Vue.js library, which provides a way to create user-friendly single-page web applications.

Upon the application's initial launch in the browser, multiple GET requests are dispatched to the server to retrieve the most recent sensor and room data, along with the predictions associated with them. This retrieval occurs asynchronously, enabling concurrent operations of other components. Utilizing the Socket.io library, we establish a WebSocket client to listen for incoming messages carrying real-time updates through a socket. Once such messages arrive, we update the corresponding slot within the data structure containing sensor data.

If a user initiates a manual update request for predictions, we send a dedicated GET request and await the response. Upon receiving a successful response from the server, acknowledging the construction of new predictions, we proceed to re-fetch predictions for both rooms and sensors.

To maintain a clean, modular, and reusable codebase, the UI is organized into multiple components, each designated to perform a specific task. As the sidebar and the component showcasing the floor plan are siblings within the DOM structure, the data communication between them is performed by using a system of props (arguments available within the components) passing and custom emits.

5.2 Backend Setup and Implementation

The application's backend refers to the server setup. Our server infrastructure is implemented using a Node.js environment and the Express.js web framework. Node.js operates on top of an event-driven architecture and, in principle, follows the asynchronous I/O design (23). This allows us to create a scalable solution capable of concurrently managing incoming MQTT sensor data streams, processing API requests, handling socket communication, and interacting with the database.

5.2.1 MQTT Messages Handling

Within the system's backend, we maintain a connection to the MQTT broker, subscribing to a designated topic. This subscription allows to receive real-time messages transmitted from the Nordic Thingy:52 devices.

The incoming messages arrive in two distinct formats: one includes recordings from various sensors that measure temperature, pressure, humidity, carbon dioxide, volatile organic compounds (VOC), noise, and voltage levels, while the other solely contains detected light intensity levels for red, green and blue colors.

Before storing the incoming messages in the database and delivering real-time updates through a socket connection to the client, it is necessary to merge two separate messages of different type to create a complete packet. To achieve this, a buffer is formed. It is structured as a sensor to data mapping, where sensor names are mapped against type I message data. As first type messages are stored in the buffer, the system awaits the arrival of the second type message associated with that specific sensor and combines them. Once both types of messages are combined, the buffer for that sensor is cleared, and the full packet is stored in the database and thereafter transmitted over the socket.

The messages received from the MQTT broker arrive as JSON-like objects, simplifying the parsing of incoming payloads. Because we work extensively with JSON objects, utilizing a database that supports JSON-like documents has been beneficial.

5.2.2 Database

The database serves as a critical component in managing the constant stream of incoming JSON-like messages received from the MQTT broker.

To address this need, a NoSQL document-oriented database was considered for its efficiency when managing JSON-like objects. Ultimately, the database of choice was MongoDB, complemented by the use of the mongoose Object Document Mapping (ODM) library to elegantly define schemas for our document collections. In total we define schemas for 5 different collections:

- SensorData

  Used for storing incoming MQTT messages. As the new messages are processed by the backend of the application, a new document is created to store the containing data. Moreover, each document contains a timestamp that indicates when the document was created.

- LatestSensorData

  Uses the same structure as the SensorData collection. This collection was introduced to improve the performance of resource-heavy queries. Instead of creating a new document for every single incoming message, a single document associated with a specific sensor is updated with the latest data. Similar to the SensorData collection, each document contains a timestamp that indicates when the document was last updated.

- Room

  A collection that contains information about the rooms that have sensors installed. The collection is initialized at the server start. The "colorCode" field is the only one that gets updated, depending on the current room occupancy status. Since there are multiple sensors installed in each room, the rooms' occupation status is updated based on the data collected by any of these sensors. Additionally, it provides coordinates that are used by the frontend to display rooms.

- Sensor

  A collection similar to the Room collection and contains information about the installed sensors. The collection is also initialized at the server start. The "colorCode" field is the only one that gets updated, depending on the current room occupancy status, which is calculated based on the data collected by this specific sensor. Additionally, it provides coordinates that are used by the frontend to display distinct sensors.

- Prediction

  Serves as storage for predictions generated by LSTM, GRU, and CNN models. Documents comprise a "modelType" field to distinguish predictions made by certain models as well as an array of constant size (3 elements corresponding to the next 3 hours) containing predicted sensor data.

To improve the execution times of complex, resource-intensive queries, several indexes were introduced in the SensorData collection. Due to the substantial volume of documents within this collection, the indexing was applied to the fields utilized for sorting operations (e.g. timestamp), aiming to enhance the execution times of such processes.

5.2.3 REST API & WebSockets

The server hosts a range of API endpoints, each adjusted to provide specific data based on the requested information. We define distinct endpoints to retrieve up-to-date sensor data for all rooms and sensors. These endpoints process the given floor ID, delivering exclusively the information relevant to the associated floor in the response. Different endpoints are used to return forecasted data and function in a similar way. All API endpoints designed to supply requested data are prefixed with the '/api' string, while the root endpoint is reserved to serve frontend-built distributions containing static files.

Additionally, we establish a socket server to facilitate real-time communication. Upon the arrival of new data from the MQTT broker, we broadcast the processed packet via the socket, enabling connected clients to access the latest data in real time.

5.2.4 Scheduled Processes

There are several tasks that require execution at regular intervals. The server runs a primitive scheduler that spawns child processes each hour. These processes include: one executing a Python prediction script; another identifying and flagging outdated entries in the database in case no updates occurred within the last hour. There are also two supplementary child processes available: one is responsible for removing entries older than a specified timestamp, while the other performs database backups. These auxiliary processes are available for optimization and maintenance purposes within the system and can be used upon necessity.

5.2.5 Manual Forecasting

While scheduled predictions rely on historical data from the past 6 hours, recordings made in the current hour do not influence the prediction. To accommodate potential trend changes within the current hour, we introduce an alternative, manual-triggered workflow that augments the data collected in the last hour with the recordings made in the current hour.

5.3 Historical Data Preprocessing

In order to format the historical data to align with the input requirements of the employed neural network models, raw data first had to be preprocessed in a number of ways. The raw data, gathered from December 1, 2022, to October 9, 2023, was organized into individual CSV files, with each file representing the data collected on a single day from the MQTT broker. The data utilized was sourced from logs generated by an MQTT broker, rather than being directly collected by our system's backend. The processing involved utilizing multiple script files, executed sequentially, to generate intermediate datasets.

5.3.1 Dataset Construction

As the messages from the MQTT broker resulted in two distinct types of packets, where one type lacked color levels recordings, it was necessary to merge these two message types associated with a particular sensor to form a complete packet, similar to how incoming real-time messages are handled. This was achieved by iterating through every CSV file and creating a buffer (in a form of a map), mapping sensor names to type I message contents for every entry within the file. When a corresponding second type II message was found, it cleared the associated slot in the buffer and merged two types of messages together. Different message types were distinguished based on the presence of carbon dioxide metric.

The subsequent dataset creation phase involved sampling all entries by the hour and averaging the metric values within each hourly interval, which added a new column called 'hour' with values ranging from 0 to 23. Since the filenames included a date, it allowed for the extraction of day values. Consequently, an extra column was added to indicate the day of the recordings, ranging from 0 to 6 (0 for Sunday, 1 for Monday, and so forth up to 6 for Saturday).

Due to instances where certain sensors were intermittently disconnected or experienced weak signal, there were periods when recordings were absent from the dataset. Consequently, this absence resulted in missing entries sampled hourly, disrupting the temporal pattern.

In order to facilitate the capture of temporal relationships by neural network models, it is vital for the data to form a continuous, uninterrupted sequence. Hence, these gaps, indicative of missing hours in the daily recordings, had to be removed from the dataset. If any hours were absent from each day's records, all entries for that day were subsequently removed. That reintroduced a similar issue, causing days with missing data rather than missing hours, thereby compromising the temporal integrity of the dataset.

To simplify the creation of a dataset compatible with neural network models, the recordings were reorganized by sensor name and stored in distinct, linked files (each file dedicated to a specific sensor) contrasting the previous approach of separate files for each day's recordings. This restructuring also enabled the assessment of whether the datasets for individual sensors met the required size criteria.

It is important to note that after removing incomplete days, the temporal consistency of the datasets remains compromised. To address this, an imputation technique was considered, aiming to fill in missing hours with mean values. For each day-hour combination, averages were computed to capture seasonal patterns. Recall our approach of hourly sampling, computing averages for all entries. Mean imputation becomes a viable choice in that case, as it operates under the assumption that the mean of a variable serves as the optimal estimator for missing values on that variable.(24) However, it was crucial to delay the imputation process as outliers are yet to be addressed; imputing at this stage could add unwanted noise by influencing the calculations of average values with outliers.

To mitigate the influence of outliers on prediction models, mean and standard deviation were computed for each sensor-day-hour combination (meaning different distributions per each sensor-day-hour group) regarding CO2, light, and noise levels, aimed at capturing sensor-specific trends and preserve seasonality. For every dataset entry's metric, a z-index was calculated based on the sensor, day, and hour. If, for any metric, the calculated z-score surpassed a threshold of 2.5, the entry was flagged as an outlier and subsequently removed. Z-score was calculated as follows:

$$z = \frac{x - \mu}{\sigma} \tag{5.1}$$

where x is the raw metric score, μ is the dataset mean, and σ is the dataset standard deviation.

After handling outliers, the further step involved imputing missing values. We verified if every 24th entry (representing the initial hour of the subsequent day) aligned with the expected sequence of days. If not, all absent days were imputed to ensure the temporal integrity, maintaining a continuous sequence of data. This resulted in the dataset containing entries of the following format:

```
day, hour, sensor, eCO2, sound, light, roomtype
```

To enable neural network models to discern patterns specific to particular sensors, the "roomtype" feature denotes the type of room where the sensors are positioned. This approach aims to provide the models with additional context, enhancing their understanding of various patterns and trends, given that our models are generic and accept input sequences from any sensor.

The resulting datasets, categorized by sensor name, varied in the resulting number of entries. For instance, the "thingy016" sensor had 6360 entries, while sensors like "thingy058" had only 48 entries. To avoid bias towards overrepresented sensors with larger datasets, those datasets smaller than 100KB were excluded. Even the largest dataset below this threshold, measuring 53KB, portrayed a significant gap compared to sensors with file sizes above the 100KB threshold.

5.3.2 Data Normalization

In order to enhance the models' comprehension of the data, a z-score based normalization method was implemented for the environmental data (metrics). Similar to the data preprocessing phase, z-scores were computed for each distinct sensor-day-hour group to maintain seasonality. The "day" column underwent one-hot encoding, generating seven columns to represent days (replacing the original single column). Using the LabelEncoder class from the sklearn library, sensor names and room types were label encoded. Each sensor in the sensor list was associated with unique ID, encoding the sensor names based on these IDs. The same procedure was applied to room types, associating each type with a distinct ID.

Following the normalization of all the values in the dataset, each entry contained 13 features (columns).

5.3.3 Input Sequences Creation

Given the original design of the predictive models, intended to forecast metric values for the subsequent 3 hours based on the data corresponding to 6 preceding hours, the dataset required segmentation into 6-hour blocks for use as inputs and 3-hour blocks for use as output labels. As the dataset is traversed, every 6-hour block constitutes the input window, while the following 3 hours form the output window. By processing the entire dataset, a sequence of windows was generated. This method is known as a Sliding (Shifting) Window approach.

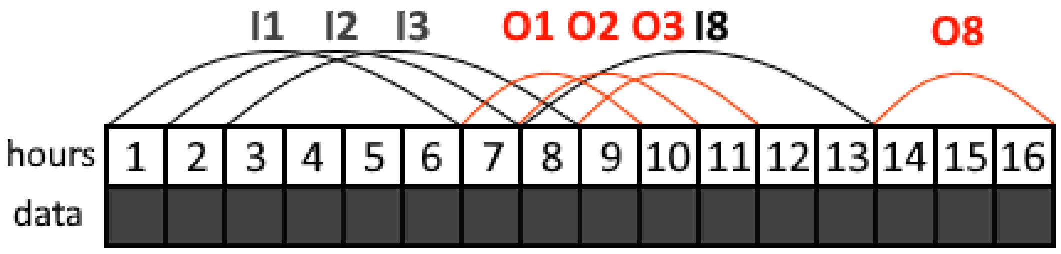


Figure 5.1: Sliding Window approach used to create input and output sequences used for training and validation. I1, I2, I3, and I8 denote the input windows, whereas O1, O2, O3, and O8 represent the output windows corresponding to the input sequences.

For model training and evaluation, 20% of the dataset was designated as a validation set. This subset served as an independent sample used to assess the model's performance during training, ensuring that the model is able to generalize well to unseen data beyond the training set.

5.4 Training Neural Network Models

At first, we define the input and output sequence formats for all our utilized models. The inputs represent sequences of six entries with 13 features, each entry representing one hour of observations. As for the outputs, the sequences follow the (3,13) format, as we generate predictions for the upcoming 3 hours. The Keras library is used to construct neural network models.

To optimize the model's training performance, we implement two callbacks designed to intervene during training. The first, EarlyStopping, uses a patience level of 5, enabling the training to stop in case the performance on the validation set degrades. Hence, we set a sufficiently large number of epochs, allowing this callback to terminate the training process once performance starts to deteriorate. The second callback, ReduceLROnPlateau, is set off when the observed metric (e.g., validation loss) stops improving. In instances where performance stagnates for three consecutive epochs, this callback reduces the learning rate by a factor of 0.2 (the old learning rate is multiplied by this factor). Doing so facilitates the model by optimizing convergence without overshooting potential local minima.

We employ the Adam optimization algorithm, a method based on stochastic gradient descent (SGD).

5.4.1 Baseline Models

To allow for model evaluation, we first outline the baseline model architectures. The LSTM and GRU models' structures draw inspiration from existing solutions, such as those proposed by Zhu and Laptev (25). Both the LSTM and GRU models adopt a two-layer stacked configuration, featuring 64 and 32 hidden states, respectively, followed by a fully connected dense layer with ReLU activation responsible for producing the final output.

The baseline CNN model uses two convolutional layers: the initial convolutional layer in the baseline CNN model employs 64 filters, while the subsequent layer utilizes 32 filters. Both layers have a kernel size of 3. After the convolutional layers, the output is flattened and forwarded to a fully connected dense layer, responsible for generating the final output. The ReLU activation function is applied to the output dense layer.

5.4.2 Hyperparameter Optimization

To improve our models' performance, we acquire the optimal set of hyperparameters to enhance each model's performance on the given dataset. Due to the inferior performance of the 1D CNN model in contrast to the RNN models, the CNN model was abandoned and did not undergo the tuning process.

Because of the exhaustive nature of the grid search algorithm, the scale of our dataset, and the large number of hyperparameter sets, an alternative hyperparameter tuning process was employed to discover the best-performing model. The Hyperband algorithm provided by the KerasTuner framework was used to find these optimal hyperparameters.

The Hyperband algorithm utilizes an approach known as "successive halving" that traverses the predetermined search space and assigns more computational budget to the models with the most promising set of hyperparameters (26). The underperforming models are

Hyperparameter	**Default value**	**Value Range**
Number of Units in the Input Layer	64	$\{x \in \mathbb{Z} \,\vert\, 32 \leq x \leq 256, x \equiv 0 \pmod{32}\}$
Number of Units in the Last/Intermediate LSTM/GRU Layer	32	$\{x \in \mathbb{Z} \,\vert\, 16 \leq x \leq 128, x \equiv 0 \pmod{16}\}$
Number of Intermediate LSTM/GRU Layers	0	$\{x \in \mathbb{Z} \,\vert\, 0 \leq x \leq 3\}$
Recurrent dropout rate	0	$\{x \in \mathbb{R} \,\vert\, 0 \leq x \leq 0.5, x = k/10 \quad \exists : k \in \mathbb{Z}, 0 \leq k \leq 5\}$
Output Dense layer activation function	"relu"	{"relu","sigmoid","tanh"}
Batch size	32	$\{x \in \mathbb{Z} \,\vert\, 2 \leq x \leq 256, x = 2^n \quad \exists : n \in \mathbb{Z}, 1 \leq n \leq 8\}$

Table 5.1: Hyperband hyperparameter tuning algorithm search space.

discarded, essentially halving the number of remaining model configurations. Ultimately, this process selects a singular model with the most optimal set of hyperparameters.

As a result of a Hyperband hyperparameter tuning algorithm, the following hyperparameter set was acquired:

- Number of Units in Input Layer — 128
- Number of Intermediate LSTM/GRU Layers - 0
- Number of Units in Last LSTM/GRU Layer — 64
- Recurrent dropout rate - 0.2
- Output Dense layer activation function - ReLU
- Batch size - 32

It is important to mention that the best hyperparameter set found shares identical values for both the LSTM and GRU models, mostly due to the similar nature of these neural networks. As a result, we assess the models using this set of hyperparameters instead of the baseline models.

5.5 Room Occupancy Status Classification

In order to provide users an ability to monitor occupancy in a more comprehensive manner, specific sensors were assigned color codes for each room or area (in case of multiple sensors in the same room) they covered. Different color codes signified different occupancy statuses: green indicated low occupancy, orange indicated limited capacity, while red suggested the room being at full capacity. The occupancy status was determined based on thresholds derived from three metrics: CO2 levels, light intensity, and noise levels.

Instead of relying on hardcoded thresholds, a more resilient approach was adopted. Utilizing a KMeans classifier and an existing dataset used for training neural network models, three distinct value clusters are formed. To exhibit temporal patterns in the data, classification is performed on data grouped by sensor, day, and hour. The center value within each cluster serves as a threshold: the center of a cluster containing moderate values indicated a threshold from low ("green") to medium ("orange") occupancy, while the center of the cluster with the highest values indicated a threshold from medium ("orange") to high ("red"). These thresholds, specific to each sensor, day, hour, and metric combination, are serialized for future utilization in determining the color-coded occupancy status of both real-time and predicted data.

Given the automated selection of the K parameter by the KMeans classifier, the algorithm attempts to determine how to construct the three clusters of values by itself. However, there is no guarantee that three distinct clusters will be consistently identified. This inconsistency leads to instances where two calculated thresholds do not show significant differences.

6

Evaluation

6.1 Experimental Setup

To evaluate the performance of both the baseline and the tuned models, we perform a regression analysis. Given the nature of the regression problem, traditional evaluation metrics used for classification problems cannot be applied to evaluate the accuracy of predicted values. Therefore, we evaluate the models' performance based on the mean squared error (MAE), mean absolute error (MAE), and Huber loss error metrics.

In order to analyze how loss changed during training on the validation set, separate models need to be trained using the different above-mentioned loss functions. This approach guarantees that the models comprehend the temporal data patterns by leveraging these loss functions during training.

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

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

$$\text{Huber} = \frac{1}{n}\sum_{i=1}^{n}\begin{cases} \frac{1}{2}(y_i - \hat{y}_i)^2 & \text{if } |y_i - \hat{y}_i| \leq \delta \\ \delta(|y_i - \hat{y}_i| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

MAE, MSE, and Huber vary in their treatment of outliers. MSE, due to its quadratic nature, significantly amplifies the impact of outliers. On the contrary, MAE tends to be more lenient towards the outliers, while Huber loss offers a middle ground between MSE and MAE. The Huber loss introduces the δ parameter, which determines a threshold to decide on the method to apply. A larger delta value makes it more robust to outliers, while a smaller delta value makes the loss function more sensitive to minor errors. Our experimental setup uses a default δ value of 1.0 provided by the Keras library.

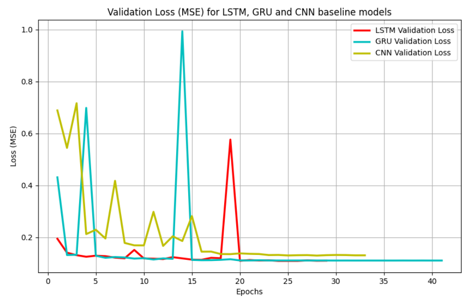


Figure 6.1: Validation loss (MSE) evolution across epochs for LSTM, GRU and CNN baseline models.

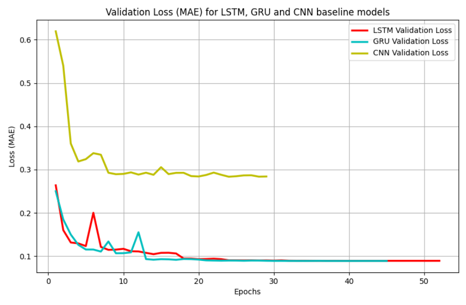


Figure 6.2: Validation loss (MAE) evolution across epochs for LSTM, GRU and CNN baseline models.

Once the models employing various loss functions finish training, we extract the training history and serialize it so that it can be reused. We illustrate the resulting validation loss for different loss functions across epochs by generating plots using these statistics. Due to the implementation of an EarlyStopping callback, certain models terminate training earlier, resulting in varying lengths of plotted lines for different models.

Figures 6.1 through 6.6 illustrate the evolution of validation loss across epochs when utilizing different loss functions.

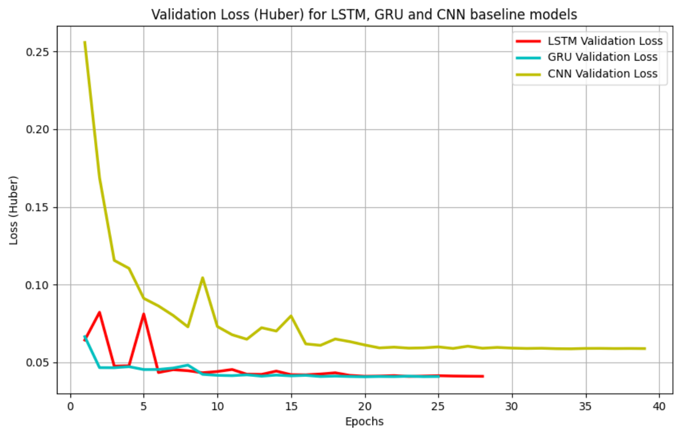


Figure 6.3: Validation loss (Huber) evolution across epochs for LSTM, GRU and CNN baseline models.

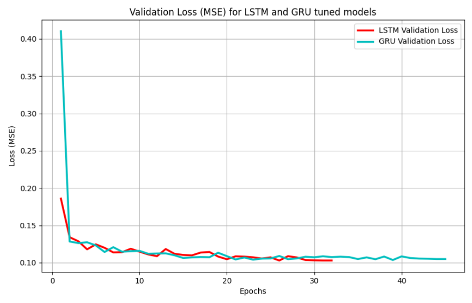


Figure 6.4: Validation loss (MSE) evolution across epochs for tuned LSTM and GRU models.

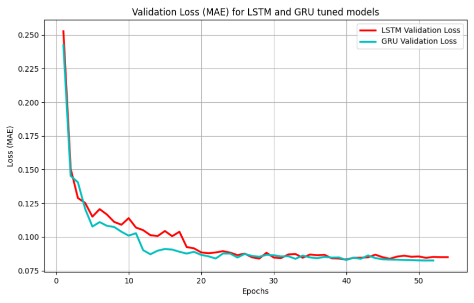


Figure 6.5: Validation loss (MAE) evolution across epochs for tuned LSTM and GRU models.

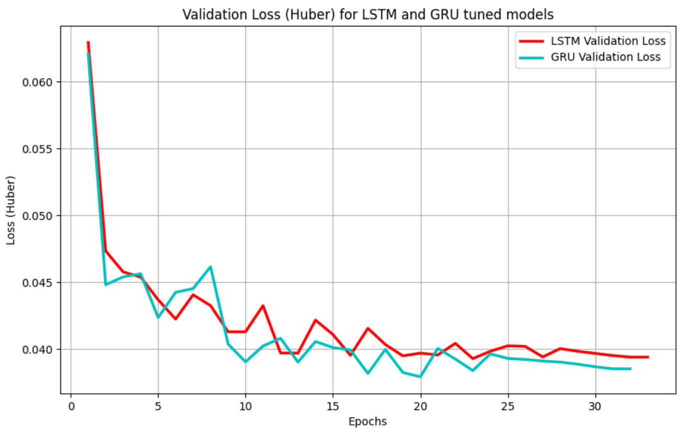


Figure 6.6: Validation loss (Huber) evolution across epochs for tuned LSTM and GRU models.

6.2 Limitations and Threat to Validity

There are various limitations that have been identified concerning the evaluation and implementation of the system, with varying degrees of impact—some having a more pronounced influence than others.

To start with, there is some bias present towards the overrepresented sensor datasets, with some datasets being three times as large as the smallest ones. This affects the models' performance in terms of producing accurate prediction results for every sensor. It is possible that underrepresented sensors might be identified as another sensor if there is no clear distinction in patterns between them.

A potential solution could be to impute the underrepresented datasets or downsample the largest ones. However, doing so might introduce additional unwanted noise, making it even harder for the neural networks to distinguish between patterns innate to different sensors. RNNs tend to make the most accurate predictions in instances where patterns observed in datasets associated with different sensors are discrete. Therefore, in the case of a generic model used to make predictions for a set of different sources, excessive uniformity might have a negative impact on the model's ability to make accurate forecasts.

To overcome that limitation, creating a system of models tailored specifically for each sensor might be beneficial. Our system design suggests a generic model in which it learns to distinguish between separate sensors on the basis of two features: the encoded sensor name and the encoded room type. To reduce computational load, one option could be to design a collection of models grouping the sensors placed in rooms of the same type together. This adjustment would produce a system of room-specific models.

The approach above, however, would only work under perfect conditions when no sensors produce incorrect or inaccurate sensor readings at any point. In practice, sensors generate incorrect data, and it takes time to recalibrate them. It is important that any part of the system is not impacted by the erroneous patterns..

Moreover, we rely on static historical data for training and calculating thresholds for classification, and as highlighted earlier, patterns can evolve over time. Hence, it is crucial to continuously update the dataset with the latest data rather than being exclusively dependent on historical records.

Finally, our models' capabilities are limited to producing predictions for the upcoming 3 hours based on the 6-hour data input. There are two aspects to that limitation. Firstly, we specify a constant input window size. The input window has to be large enough to encapsulate discernible patterns in environmental data. Additional research is required to determine the most suitable input window size for our domain. Secondly, the prediction mechanism for the upcoming hours functions by forecasting the values of the next hour and then utilizing that hour's predicted values alongside the preceding 5 hours to construct a fresh input sequence for predicting the following hour's value. This process leads to subsequent hour predictions being influenced by predicted values for preceding hours, which introduces an additional layer of inaccuracies.

6.3 Discussion on Evaluation

The trends observed in Figures 6.1 through 6.3 and the resulting regression error metrics suggest that RNN models perform better in the context of our problem compared to CNN models. The CNN model results in a 0.2840 MAE value, while the MAE values for the LSTM and GRU models do not exceed 0.0893. Although the difference is less pronounced when examining other metrics like MAE and Huber, it is still significant enough to exclude the CNN model from the evaluation. Hence, our analysis primarily focuses on two models: LSTM and GRU.

As training progresses, certain spikes in the validation loss become more noticeable, particularly in baseline models compared to tuned models. The implementation of a recurrent dropout rate of 20% aimed to mitigate overfitting problems, contributing to the reduction of overfitting.

Moreover, significant spikes in the MSE validation loss are noticeable across all models. Since we utilize the Adam optimizer and a relatively small batch size, these spikes are an inherent outcome of mini-batch gradient descent (27), since some batches may contain flawed data that negatively affect optimization.

	Baseline			Tuned	
	LSTM	GRU	CNN	LSTM	GRU
MSE	0.1103	0.1110	0.1310	0.1029	0.1048
MAE	0.0893	0.0881	0.2840	0.0850	0.0825
Huber	0.0410	0.0409	0.0588	0.0394	0.0385

Table 6.1: Comparison of the baseline and tuned models based on the resulting regression error metrics.

Across all models, training with Huber loss as the loss function yields the lowest resulting loss on the validation data. Consequently, the models trained using Huber as the loss function will be regarded as the most optimal solution, resulting in the lowest error in terms of predicting environmental data.

LSTM and GRU models showcase similar performance when compared using different regression error metrics. On average, a tuned LSTM model exhibits a 5.14% improvement in error metrics compared to its baseline version. Similarly, the tuned GRU model, on average, shows a 5.93% improvement compared to its baseline version.

7

Conclusion

7.1 Summary of Research Outcomes

We have developed an extensive real-time monitoring system integrating LSTM, GRU, and 1D CNN predictive models for environmental parameter forecasting. Our discussion encompasses the limitations and challenges encountered during system implementation, including the addressing of dataset bias, tailoring models for sensor- and room-specific predictions, managing intricate patterns in sensor readings, and tackling limitations in model training.

Furthermore, we enhance the system's scalability and adaptability when compared to existing solutions by introducing a document-oriented database structure capable of handling extensive data volumes and providing an asynchronous, non-blocking, and event-driven approach to handling communications between the system's integral components.

To assess the models' performance, we conducted a comparative analysis between various model architectures using diverse regression error metrics, including MSE, MAE, and Huber. The results indicate that, on average, the tuned LSTM and GRU models exhibit an improvement of 5.14% and 5.93%, respectively, in comparison to their baseline versions. Specifically, the tuned LSTM achieved a Huber loss of 0.0394, and the tuned GRU scored 0.0385 on the validation set. Moreover, we conclude that the CNN model does not perform as effectively as recurrent neural network models on the given dataset.

7.2 Future Work

Based on the limitations discussed in this paper, several areas of improvement emerge, including:

- Integration of a retrainable model into the current system

  The current implementation involves offline model training, utilizing static historical data. Integrating real-time data with the existing dataset enables the model to capture potential changes in trends.

- Integration with the reservation systems

  The system presented in this paper infers occupancy statuses through statistical models and observed data trends due to the absence of actual information about people's presence. Integrating the reservation system could offer additional insights into determining real-time room occupancy as well as enhance the accuracy of environmental data predictions.

- Application in diverse benvironments

  By extending the system beyond usage within the NU building to other building structures and environments, we could assess how the system's performance translates to other settings that have different environmental factors.

- Usability and UI refinement

  Conduct a user experience study involving potential stakeholders and end-to-end users to improve the usability of the system based on the provided feedback. Implementing additional features could help to gain a better understanding of the presented data and observed trends.

Each of the presented suggestions contributes to enhancing the system's functionality, adaptability, and user-centric design, offering promising prospects for future improvements and refinements.

References

[1] Muhammad Irsyad Abdullah, Lilysuriazna Raya, Mohamad Adib Akmal Norazman, and Untung Suprihadi. **Covid-19 Patient Health Monitoring System Using IoT**. In *2022 IEEE 13th Control and System Graduate Research Colloquium (ICSGRC)*, pages 155–158, 2022. 1

[2] Galina Ivankova, Ekaterina Mochalina, and Natalia Goncharova. **Internet of Things (IoT) in logistics**. *IOP Conference Series: Materials Science and Engineering*, **940**:012033, 10 2020. 1

[3] Tahera Kalsoom, Shehzad Ahmed, Piyya Muhammad Rafi-Ul-Shan, Muhammad Azmat, Pervaiz Akhtar, Zeeshan Pervez, Muhammad Imran, and Masood Ur Rehman. **Impact of IoT on Manufacturing Industry 4.0: A New Triangular Systematic Review**. *Sustainability*, **13**:12506, 11 2021. 1

[4] Cor Verdouw, Harald Sundmaeker, Bedir Tekinerdogan, Davide Conzon, and Teodoro Montanaro. **Architecture framework of IoT-based food and farm systems: A multiple case study**. *Computers and Electronics in Agriculture*, **165**:104939, 2019. 1

[5] Rodger Lea and Michael Blackstock. **City Hub: A Cloud-Based IoT Platform for Smart Cities**. *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, **2015**:799–804, 02 2015. 1

[6] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. **Internet of Things for Smart Cities**. *IEEE Internet of Things Journal*, **1**(1):22–32, 2014. 1

[7] Davit Marikyan, Savvas Papagiannidis, R. Ranjan, and Omer Rana. **Working in a Smart Home-office: Exploring the Impacts on Productivity and**

Wellbeing. In *17th International Conference on Web Information Systems and Technologies*, pages 275–282, 01 2021. 2

[8] Jazzley Louisville. *ViMo: A real-time ambience based monitoring system.* Bachelor thesis, Vrije Universiteit Amsterdam, 2023. 2, 6

[9] R A Atmoko, R Riantini, and M K Hasin. **IoT real time data acquisition using MQTT protocol**. *Journal of Physics: Conference Series*, **853**(1):012003, may 2017. 6

[10] Heriberto J. Jara Ochoa, Raul Peña, Yoel Ledo Mezquita, Enrique Gonzalez, and Sergio Camacho-Leon. **Comparative Analysis of Power Consumption between MQTT and HTTP Protocols in an IoT Platform Designed and Implemented for Remote Real-Time Monitoring of Long-Term Cold Chain Transport Operations**. *Sensors*, **23**(10), 2023. 7

[11] Shilpa P. Khedkar, R. Aroul Canessane, and Moslem Lari Najafi. **Prediction of Traffic Generated by IoT Devices Using Statistical Learning Time Series Algorithms**. *Wireless Communications and Mobile Computing*, **2021**:5366222, Aug 2021. 7

[12] Nordic Semiconductor ASA. *Nordic Thingy:52 User Guide.* Nordic Semiconductor ASA, 2019. 8, 9

[13] Carles Gomez, Joaquim Oller Bosch, and Josep Paradells. **Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology**. *Sensors (Basel, Switzerland)*, **12**:11734–53, 12 2012. 9

[14] Gastón C. Hillar. *MQTT Essentials - A Lightweight IoT Protocol.* Packt Publishing, 2017. 9

[15] Peter T. Yamak, Li Yujian, and Pius K. Gadosey. **A Comparison between ARIMA, LSTM, and GRU for Time Series Forecasting**. In *Proceedings of the 2019 2nd International Conference on Algorithms, Computing and Artificial Intelligence*, ACAI '19, page 49–55, New York, NY, USA, 2020. Association for Computing Machinery. 13

[16] Gissel Velarde, Pedro Brañez, Alejandro Bueno, Rodrigo Heredia, and Mateo Lopez-Ledezma. **An Open Source and Reproducible Implementation of LSTM and GRU Networks for Time Series Forecasting**. *Engineering Proceedings*, **18**(1), 2022. 13

[17] Chi Chen, Lei Xue, and Wanqi Xing. **Research on Improved GRU-Based Stock Price Prediction Method**. *Applied Sciences*, **13**(15), 2023. 13

[18] Ya Gao, Rong Wang, and Enmin Zhou. **Stock Prediction Based on Optimized LSTM and GRU Models**. *Scientific Programming*, **2021**:4055281, Sep 2021. 13

[19] Sameh Mahjoub, Larbi Chrifi-Alaoui, Bruno Marhic, and Laurent Delahoche. **Predicting Energy Consumption Using LSTM, Multi-Layer GRU and Drop-GRU Neural Networks**. *Sensors (Basel)*, **22**(11), May 2022. 13

[20] Farah Shahid, Aneela Zameer, and Muhammad Muneeb. **Predictions for COVID-19 with deep learning models of LSTM, GRU and Bi-LSTM**. *Chaos, Solitons & Fractals*, **140**(C), 2020. 13

[21] Sonia Guessoum, Santiago Belda, Jose M. Ferrandiz, Sadegh Modiri, Shrishail Raut, Sujata Dhar, Robert Heinkelmann, and Harald Schuh. **The Short-Term Prediction of Length of Day Using 1D Convolutional Neural Networks (1D CNN)**. *Sensors*, **22**(23), 2022. 13

[22] Mikita Volakh. **Real-time Monitoring and Predictive Modeling for Space Occupancy**. GitHub Repository. 14

[23] M. Casciaro. *Node.js Design Patterns.* Community experience distilled. Packt Publishing, 2014. 15

[24] Qinbao Song and Martin Shepperd. **Missing Data Imputation Techniques**. *IJBIDM*, **2**:261–291, 10 2007. 19

[25] Lingxue Zhu and Nikolay Pavlovich Laptev. **Deep and Confident Prediction for Time Series at Uber**. *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 103–110, 2017. 22

[26] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. **Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization**, 2018. 22

[27] Diederik P. Kingma and Jimmy Ba. **Adam: A Method for Stochastic Optimization**, 2017. 29

Appendix A: Reproducibility

Start by cloning the project available in this GitHub repository. There are several requirements to run the project:

- Node.js v16.14.2+
- npm packet manager (included with a Node.js installation)
- Python v3.10.11+
- pip packet manager

Server Setup

First install the dependencies:

```
cd server && npm install
```

Then the server could be run by executing the following script and endpoints will be accessible on the port specified in the local environmental variables:

```
npm run start
```

Client (Development Server) Setup

In case you want to change the frontend source code, you will need to run the development server and build the distribution after you make any changes. First install the dependencies:

```
cd client && npm install
```

Then the client (dev server) could be run by executing the following script:

```
npm run dev
```

The application could be accessed at **localhost:5173**. To build the distribution containing static files run:

```
npm run build
```

Authorisation

Project uses environmental variables through **dotenv** npm module. You have to be authorized both to access the MQTT data the and to be able to connect to the database. Source code uses the following environmental variables (case-sensitive, exact spelling):

- PORT

  Port the server runs on.

- MQTT_USERNAME

  Username of a client connecting to the MQTT server (must be configured on the server).

- MQTT_PASSWORD

  Password of a client connecting to the MQTT server (must be configured on the server).

- MQTT_HOST

  MQTT host URI.

- DB_URI

  MongoDB connection string containing the username and the password. Any options params must be URL encoded.

- PROCESSED_BYSENSOR_REM_DATA_PATH

  Path containing preprocessed raw dataset per each sensor in the CSV format.

- ROOM_DATA_PATH

  Path containing the xlsx file with the room information.