Evaluating the Deployment of Microclimate Prediction Models On Embedded Systems

# **Abstract**

# This paper explores the feasibility of utilizing powerful machine learning models for microclimate modelling andprediction in extremely remote environments. The experiments outlined in this paper simulate rigid constraints on economic cost, computing power, and internet connectivity in order to draw comparisons to real-world settings such as isolated farms or offshore environments, where weather prediction is often critical, but resource constraints make it exceedingly difficult. Cost and computing constraints are considered through the use of an inexpensive STM32F767 board which hosts 2MB of flash memory, 512KB SRAM, and operates within a supply voltage range of 1.7 V to 3.6 V. The paper examines the trade-offs between model accuracy, computing power, and inference latency. Multiple custom neural network architectures are designed, built, trained, **pruned** and **quantized** to minimize memory usage, and deployed to the board, situatedin a rural area within the city of Couva, Trinidad and Tobago. The various model performances are then evaluated and compared. Finally, as we aim to demonstrate the feasibility of deploying sufficiently accurate models within these resource-constrained settings, the performances of the pruned and quantized models are evaluated against full-sized models deployed in the cloud, highlighting that stellar performance can be achieved despite rigorous hardware constraints.

# **Keywords**

TinyML, Embedded Systems, Microclimate, Machine Learning, EdgeAI, Caribbean Climates

# **Introduction**

## Background to Weather Forecasting

Accurate weather forecasting plays a critical role in numerous tasks across various aspects of modern society. Weather forecasts heavily influence individual decision-making: the type of clothing worn, outdoor activities planned, and the mode(s) of transportation chosen for travel.

On a larger scale, these forecasts become even more crucial. In power demand forecasting, for example, Diaz-Iglesias et al. demonstrated that weather variables significantly impact the need for heating, cooling and lighting, thus highlighting the close correlation between weather and energy consumption patterns. In the field of agriculture, Shen et al. found a positive correlation between the frequency at which agricultural production entities pay attention to meteorological forecast information, and a subsequent increase in their income (2024). In the field of offshore installations and maintenance as well, Døskeland et al. highlighted how work performed close to offshore platforms or subsea infrastructure is often very sensitive to weather, and how decisions made around weather forecasts could have serious Health and Safety (HSE) and economic ramifications (2023).

Historically, weather forecasting has been achieved primarily through the use of Numerical Weather Prediction (NWP). Predictions are obtained by numerical integration of partial differential equations to determine the current state of the Earth’s atmosphere (Ben Bouallègue et al., 2024). These equations to model the physical laws of fluid dynamics and thermodynamics were first introduced in the early 20th century. Since then, there have been periodic, incremental improvements in NWP through advancements in supercomputers, numerical modelling and other techniques. Forecast accuracy can be increased by raising the resolution of the model (through reducing time steps or decreasing grid spacing), however, we quickly run into limitations imposed by high computational and timeline costs (de Burgh-Day and Leeuwenburg, 2023).

Additionally, NWP models generally work best at larger spatial scales rather than very localized microclimates. Microclimates describe climatic parameters (such as temperature, humidity and precipitation) in and around a homogenous, smaller zone of interest. As highlighted by Kumar et al. (2021), commercial weather stations which capture the data used by these NWP systems are often located a significant distance away from regions of interest, leading to degradations in climatic patterns and poor forecasting performance.

Improving forecasting capabilities within these microclimates could result in substantial economic gains in an assortment of fields such as those highlighted above, and more specifically: offshore-platform activity planning, commercial fishing scheduling, and large-scale agricultural crop rotations.

## The Impact of Machine Learning

Data-driven Machine Learning (ML) solutions are being increasingly recognised for their potential in this space, as they are able to deliver these predictions with much lower computational costs. As the amount of publicly-available weather datasets grows, so too does the viability of weather forecasting ML solutions. For example, the European Centre for Medium-Range Weather Forecasts (ECMWF) publishes and maintains a massive open dataset of hourly estimates for global weather data (Copernicus Climate Change Service and Climate Data Store, 2018). Dueben and Bauer were able to utilize this dataset to create several Neural Networks (NNs, one type of ML model available), and were able to forecast weather in a 24 hour timeframe reasonably well (2018).

The recent explosion of research into NNs has given rise to a myriad of model architectures which propagate input data through multiple layers, with each layer progressively extracting higher level features from the data.

## General Architecture of Neural Networks

The figure below illustrates the basic architecture of a NN. In this example, we have 3 input nodes (which represent 3 features of input data). These 3 input nodes are all connected to 4 hidden nodes (which perform computations on the data to learn its features and patterns), and finally, these are connected to 2 output nodes (which will produce a label or prediction for our input data).



***Figure X:*** *Basic feedforward neutral network architecture (adapted from Goodfellow et al., 2016)*

To further illustrate, if we were to utilize an architecture such as this to predict weather, you might imagine the 3 input nodes as temperature, humidity and pressure data that we collect from sensors, and the output nodes as temperature and humidity that we are trying to predict using our model.

NN architectures with several hidden layers are commonly termed Deep Learning (DL) or Deep Neural Networks (DNNs), and have been shown to be excellent at learning features of the input data, which enables better eventual predictions or classifications than traditional ML approaches (Shiri et al., 2023). Within this broad class of DNNs, there exist subclasses such as Multi-Layer Perceptrons (MLPs), Convolution Neural Networks (CNNs), Recurrent Neural Networks (RNNs) and Long Short Term Models (LSTMs), with each subclass having its own set of particular strengths and being well suited to its own set of tasks.



***Figure X.*** *The hierarchical relationship between several major ML classes, adapted from (Shiri et al., 2023).*

Typically, these ML models are run in large datacenters, on cloud servers. These cloud servers often include powerful Central Processing Units (CPUs) alongside multiple Graphical Processing Units (GPUs), each equipped with several gigabytes (GBs) of memory. This is necessary because these models require large amounts of memory, memory bandwidth, and parallel computing power to handle speedy, high-dimension matrix operations. This, in turn, translates to high power demand by these servers (commonly several hundreds of Watts,W) alongside high operational and maintenance costs. Additionally, it means that all systems that incorporate these models must configure and maintain one (or more) Internet connections (which can be pricey, depending on the location), and must accept the high latency and privacy risks incurred during network communication with these cloud servers.

## ML on Embedded Devices

However, in recent times there has been growing interest in deploying these models on low-power embedded devices on the edge (i.e close to where we produce and utilize the data). An embedded device is a computer system with a dedicated function within a larger electrical or mechanical system. These devices often take the form of Microcontroller Units (MCUs). They are often optimized for low power consumption and cost, and small sizes (Kelemen et al., 2020).

By running these models on these devices, we can provide better privacy and lower our bandwidth requirements (since we don’t need to transmit all of our data over the Internet), lower our power requirements (to several milliwatts, mW), and provide more real-time data processing (Han et al., 2015). This alternative way of running ML algorithms by using these low-profile devices is termed TinyML (Abadade et al., 2023). More specifically, TinyML refers to ML inference performed on edge devices that typically consume power in the milliwatt range, while devices that consume more than this fall into the broader class of Edge AI devices (Warden and Situnayake, 2019).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ML Model Type (and example use case)** | **Platform Example** | **Memory Footprint** | **Power Consumption** | **Internet Connection Required** |
| CNN (Image Classification) | CPU & GPU Server (eg. Intel Xeon & NVIDIA V100) | 200–500 MB | 200–300 W | Yes |
| LSTM (Weather Forecasting) | CPU & GPU Server | 100–300 MB | 50–150 W | Yes |
| Shallow Neural Network (MLP) | Laptop or CPU-only Server | 50–200 MB | 30–80 W | Yes |
| **Quantized CNN (Keyword Spotting)** | Embedded Microcontrollers (e.g. Arduino Nano 33 BLE Sense) | 20–150 KB | ~5–20 mW | No |
| **Tiny LSTM (Temperature Prediction)** | Embedded Microcontrollers (e.g. STM32 / Raspberry Pi Pico) | 100–300 KB | ~50–150 mW | No |
| **Compressed DNN (Gesture Recognition)** | Embedded Microcontrollers (e.g. STM32 / Raspberry Pi Pico) | 100–500 KB | ~1–10 mW | No |

*Table X. Comparison of resource requirements for conventional ML and TinyML models (adapted from Warden and Situnayake, 2019; Codeluppi et al., 2021; Morales-García et al., 2023; Heydari et al., 2025).*

This deployment, though, requires these ML models to be highly memory-optimized and compressed so that they can fit on the on-chip SRAM (Static Random Access Memory) and flash memory, which are both ordinarily only a few megabytes (MB) or hundreds of kilobytes (KB) in size. In order to achieve this, researchers utilize techniques such as pruning and quantization, which are able to produce ML models at much lower sizes, but similar levels of accuracy. Pruning involves removing less important neurons and connections from the models, while quantization involves reducing the precision of weights and activations stored by the model. These techniques can enable compression ratios of up to 49x and memory footprint reductions greater than 90%, while maintaining similar levels of accuracy as the full models (Heydari and Mahmoud, 2025).

In the following section, we further examine research and analysis performed into deep neural network architectures, compression techniques employed on these architectures to enable TinyML, and specific use cases where these TinyML model architectures have been deployed.

# **Related Work**

## Background to Neural Networks and Dense Neural Networks

In this section, we will first review the foundational ideas and computations underpinning NNs, then further expand these ideas to understand the more sophisticated class of DNNs. We will focus solely on explaining these ideas in the domain of regression (i.e. the prediction of numerical values). NNs and DNNs can be used in many other domains such as classification, but that is outside the scope of this work.



***Figure X.*** *The basic computations involved in a single neuron, adapted from (Sze et al., 2017).*

In the above figure, Sze et al. (2017) illustrate a series of *n* input features of X, fed into a neuron. The *n* features represent different aspects of the input dataset, X, that were captured. For example, a weather dataset might have 3 input features: temperature, pressure and humidity (*n*=3 in this case).

The weights, *w*,represent multipliers that allow the NN to assign lower or higher importance to linear combinations of the input features.

The biases, *b*, are offsets that allow the NN to shift the output so that the function does not always need to pass through the origin.

The functions, *f*, are non-linear activation functions that transform the weighted sum. This allows NNs to closely approximate and model real-world functions. Common examples include the Rectified Linear Unit (ReLU) and hyperbolic tangent (tanh) functions.

The predicted output *Ŷ*, is then taken as the output of this activation function.

In supervised ML tasks, the true output *Y* for several input data points is labelled prior, and the accuracy of the NN is computed by measuring the difference between the predicted output *Ŷ* and the true labelled output *Y*, using a loss function. There also exists another category called unsupervised ML, but that is outside the scope of this work.

Two main classes of loss functions are typically used, the Mean Absolute Error (MAE), and the Mean Squared Error (MSE).

MAE is mathematically defined as:

and for MSE:

Where ​ = true value, = predicted value, = total number of samples

MAE computes the aggregation of the L1 losses, while MSE computes the aggregate L2 losses. In simple terms, the MAE gives an average of how far off the predictions are from the true values, while the MSE squares the prediction errors so that the differences are further amplified.

MSE is the most widely used for time series forecasting tasks, but both functions have their comparative advantages. For example, MSE is much more sensitive to outliers than MAE, and MAE is much quicker to compute (Jadon et al., 2024).

It can be observed that the computation of the predicted output for a neuron and the loss for that output involves several multiply and accumulate (MAC) operations. These MAC operations for the data points may occur millions of times across the entire NN. These operations typically dominate the computation cost of the NN architecture. The number of MACs is thus a common measure to benchmark the performance of an architecture, and becomes even more significant in the field of TinyML, where compute power is often severely limited.

In practice, *X*, *Y* (and *Ŷ*) are commonly vectors (i.e data represented by rows and columns), which allow the NNs to process large datasets with multiple dimensions using highly-parallelized computations.

These individual neurons can then be scaled to form *layers*. A single layer of a NN can be composed of hundreds or thousands of neurons, and DNNs can often have tens or hundreds of these layers. Due to the non-linearity of the activation functions, these large NN architectures have been found to be capable of learning high-level features of input data with significant complexity and abstraction, which allows them to perform advanced predictions of numerical values, recognise composite objects and even interpret entire scenes in images.



*Figure X: An example of an image classification task using a DNN, adapted from (Sze et al., 2017). Image is open-sourced from* [*Unsplash.com*](http://unsplash.com)

Sze et al. then go on to discuss the two high-level stages typically involved in all ML tasks. The first stage is termed the *training* stage, and involves randomly initializing the parameters (weights and biases) of the model, then splitting the input dataset into three parts to optimize these parameters using techniques such as gradient descent:

1. The training subset is used to improve these parameters by computing a loss (the difference between the estimated output from the model and the true output of the dataset).
2. The validation subset is used to tune hyperparameters for the model (which control things such as the speed at which the model updates its parameters) and prevent overfitting (ensure that the model is generalizable to all future data, and doesn’t just perform well on this input dataset).
3. The testing subset is used to provide an unbiased estimate of the performance of the model, using data that the model has never seen before, so that it can be compared fairly to other models.

This training stage is usually computationally-heavy and time-consuming, and is thus usually performed on the cloud. The output of this first stage is our trained model, which consists of a series of weights and biases along with the activation functions, all arranged in the particular NN architecture (measured primarily by its total MACs).

The second stage is called the inference stage, where the trained model is utilized to predict new datapoints, and it is on this stage that optimized computation is essential, to ensure that the models run smoothly on embedded devices.

## Efficient Neural Networks and TinyML

Typically, the size and complexity of DNN architectures increase as the capability of the model is increased. The table below illustrates the computation costs of some popular DNN architectures, when used for a particular image classification challenge called ImageNet.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Metric** | **LeNet5** | **AlexNet** | **Overfeat fast** | **VGG 16** | **GoogLeNet v1** | **ResNet 50** |
| Error rate | n/a | 16.4% | 14.2% | 7.4% | 6.7% | 5.3% |
| Total Weights | 60k | 61M | 146M | 138M | 7M | 25.5M |
| Total MACs | 341k | 741M | 2.8G | 15.5G | 1.43G | 3.9G |

Table X: An illustration of some typical DNN architectures with their metrics, adapted from Sze et al. (2017)

In this table, we can see that the capability and complexity increase from left to right, and the error rate decreases (i.e. the accuracy increases) in the same manner. The higher number of weights translate directly to higher memory requirements, and the heightened MACs translate to greater processing time. Both translate to a much greater energy requirement for inference in the model.

However, researchers have found several techniques to reduce these weight and MAC requirements, while keeping the accuracy of the model approximately constant. The two main techniques we will delve into are pruning and quantization.

As presented by Abadade et al., pruning a NN involves setting a particular threshold for weights and eliminating those weights below that chosen level. In this manner, only the most significant weight contributors (most important neurons) are kept in the model. There is an initial loss of accuracy, but this can be easily restored through retraining of the sparse NN (as opposed to the initial dense NN) (2023).

Cheng et. al expanded on this by first explaining that pruning could be broadly divided into two classes: unstructured and structured pruning (2024). Unstructured pruning refers to the removal of weights anywhere in the network (in individual neurons), while structured pruning conducts pruning across entire filters, channels or layers. They advised that without special hardware or software libraries, structured pruning was more suitable in most cases because of the complexity in implementing unstructured pruning.

Quantization is also presented as a technique to reduce the precision of weights and activations in the NN architecture. Instead of representing these as 64-bit or 32-bit floating-point numbers, quantized models represent these parameters as 8-bit fixed-point numbers or other, more-efficient schemes.



Figure X: A simple visual illustration of pruning and quantization of a subset of neurons

Abadade et al. also discussed several other techniques for compressing these DNNs such as huffman coding and knowledge distillation, but we chose to focus mainly on pruning and quantization as they were the best supported in standard application and tooling environments (2023).

In the next section, we explore some of these popular toolsets, and how well they’re able to implement these techniques.

## TinyML Framework and Library Analysis

In the field of TinyML, no single framework or library has emerged yet which implements all of the optimization techniques discussed so far. As such, various vendors involved in the embedded system ecosystem have created their own tools to assist in implementing these techniques. Immonen and Hämäläinen presented an overview of several of these tools, and we highlight the three most applicable here (2022).

The first framework discussed was TensorFlow Lite for Microcontrollers (TFLM), which is an open-source framework for running ML inference on embedded devices. It can be used with an assortment of ARM Cortex-M microcontrollers, and is widely supported by vendors such as Espressif and Arduino.

They then discuss the Cortex Microcontroller Software Interface Standard-NN or CMSIS-NN, developed by ARM. This library allows NN inference on ARM Cortex-M processors, and is specifically built to support low-level optimized versions of functions and networks such as Convolution layers and Softmax layers. This library is not typically used by itself, but rather integrated with other tools such as TFLM.

Finally, they discuss STM32Cube.AI, which is a NN and ML toolkit for the ST family of microcontrollers. This toolkit supports several advanced features such as automatic conversion of pretrained models and support for most other tools such as TFlite and keras, but is only supported for STM32 microcontrollers, leaving a large subset of ARM boards without support.

Several studies note that STM32Cube.AI produces reduced memory usage and faster execution times than TFLM, indicating that it might be preferable for STM32 devices (Hasanpour et al.,2025;Osman et al., 2022). ST also provides a cloud-based AI developer environment, which supports similar optimizations and additionally facilitates remote benchmarking of inference performance across a multitude of their boards using a server board-farm (“ST Edge AI Developer Cloud,” 2023).

The figure below illustrates a typical toolchain for a TinyML workflow.



Figure X: Example of a typical TinyML toolchain for training/inference on an embedded device.

In the next section, we briefly explore some past studies where TinyML devices have been deployed in the field.

## TinyML and EdgeAI Case Studies

Codeluppi et al. performed an extensive experiment on running ML models on edge AI devices, by creating and analysing Long Short-Term Memory (LSTM) networks, Recurrent NN (RNNs) and Artificial Neural Networks (ANNs) (i.e. Multilayer Perceptrons, MLPs) deployed on a Raspberry Pi device (2021). These NNs were used to predict air temperature inside a greenhouse (in its microclimate), and were chosen as they were found in the literature to be particularly good at time-series forecasting.

The authors highlighted how climate variables inside a greenhouse are critical to its commercial operation, as the growth rate of the products being grown depend on variables such as soil moisture, air humidity and temperature. They mentioned that this makes greenhouses one of the best use-cases for such a ML model, as the outputs of the model can be directly connected to actuators to regulate the temperature, which in turn directly improves revenue for the greenhouse operation.

For their experiment, the authors collected air temperature data from sensor nodes deployed within the greenhouse at ten minute intervals, over a period of sixteen months. They then engineered this large dataset to create seventy subsets of data, which were used to train seventeen models using the Keras ML framework.

They found that the RNN and LSTM models resulted in the highest performance, with the lowest root mean squared errors (RMSE) of 0.289 and 0.294 Celsius respectively. However, these models required more processing power than the regular MLP. For the overall balance of performance and power, they concluded that the MLP model was the best candidate.



Figure X: Possible application scenario for DNNs in a greenhouse microclimate, adapted from (Codeluppi et al., 2021)

It should be noted that these authors did not employ any pruning or quantization techniques, likely because they were working with a larger, higher-specification single board computer (SBC), with RAM in the range of hundreds of megabytes, and not a true TinyML embedded device.

Deutel et al. studied DNN implementation on ARM Cortex-M-based systems, utilizing pruning and quantization to compress the models for these TinyML devices (2022). They mainly focused on a proprietary compression pipeline that they created, but illustrated that the performance was comparable to a standard pipeline involving Keras, TensorFlow and TFLM.

They utilized a Raspberry Pi Pico and an Arduino Nano 33 BLE Sense, which both come equipped with only 256 Kilobytes of SRAM and one or two megabytes of flash memory. They were able to compress popular DNN architectures such as LeNet, AlexNet and ResNet to enable them to be run on these miniscule amounts of RAM.

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Compression Ratio** | **Equivalent Memory Reduction** | **Relative Accuracy** |
| LeNet | > 99 % | Model size reduced by > 100× | > 99 % |
| AlexNet | > 98 % | Model size reduced by ~50× | ≈ 95 % |
| ResNet | ≈ 93 % | Model size reduced by ~15× | ≈ 96 % |

Table X: Summary of DNN model compression, adapted from (Deutel et al., 2022)

The table above summarizes the results that the authors were able to achieve. In all cases, they found that increasing the compression ratio decreased the execution time required for inference, but also decreased the relative accuracy. Thus, there was a tradeoff between maximum tolerable execution/inference time and minimum tolerable accuracy.

They found that the combination of pruning and quantization produced the best results in terms of lowest memory requirements and lowest execution times. They highlighted the difference between element-wise pruning (removal of individual weights within a layer) and structural pruning (removal of entire structures of weights such as filters, channels, layers etc.), and concluded that structural pruning resulted in better savings of memory and execution time. Additionally, they found that weights quantized to integers resulted in better system performance than when they remained as floating point numbers.

## Comparison of Neural Network Architecture Performance

Lara-Benítez et al. compared the performance of seven popular NN architectures, including MLPs, CNNs and LSTMs, in time-series forecasting tasks across twelve datasets (2021). These time series forecasting tasks were found to have a critical component in a plethora of research areas such as energy consumption, retail sales and, most importantly for us, weather.

For evaluating the performance, instead of using the Mean Absolute Error (MAE), the most commonly used metric for evaluation, they chose instead to use a weighted error (termed the Mean Absolute Percentage Error, MAPE) by dividing the MAE by the mean of the dataset. This allowed them to obtain an error that could be compared across multiple time series datasets.

After statistical analyses of the MAPE of each architecture in each dataset, they found that the MLPs generally performed the worst, and the LSTMs and CNNs performed the best (had the lowest MAPEs).

When considering efficiency of the architecture (length of time required for both training and inference), MLPs were found to perform the best, with CNNs following right after. This led them to conclude that CNNs resulted in the best speed/accuracy tradeoff.

They also suggested that CNNs generally performed best for time series forecasting with four layers, and no max-pooling layers. There was no conclusive evidence on the best number of filters to utilize. This provided a good starting point for our own implementation.

In the next section, we draw on all of the knowledge we reviewed thus far to set up our own experiment.

# **Materials and Methods**

For our own research, we set out to build a system capable of forecasting outdoor air temperature in a rural Caribbean location, using ML on a simple, low-power embedded system.

As illustrated below, we first set up several sensors to record and log ambient environmental data at the location, after which we preprocessed the data and set out building the full ML model.

Once the model was built, we pruned, quantized and evaluated the model, before deploying it on our embedded board. We then analyzed the performance of the deployed model while performing inference, and stepped back to ML model building to fine-tune the architecture and hyperparameters of the model as required to improve our results.



Figure X: The overall process that we used to guide our experiment

We dive into more detail of each step in the following sections.

## Sensor Data Collection

We utilized three sensors to collect the local environmental data that we needed for the initial training:

* A BME280: To measure temperature, humidity and pressure
* A VEML7700: To measure light intensity
* A LM393: To measure whether it rained or not

Besides these sensors, two other modules were required in order to build the dataset accurately:

* A DS3231 Realtime Clock (RTC) module: To maintain an accurate internal clock for timestamps
* A MicroSD Breakout Board and MicroSD card: To store the periodic measurements from the sensors

An STM32 F767 board was used as the core of the system. The following figure illustrates the physical connections between these modules and the board:

A diagram of a computer

AI-generated content may be incorrect.

Figure X: Physical connections between the modules discussed and the STM32 board

Several small breadboards and jumper wires were used to make the physical connections.

The board was configured through STM32 Cube IDE with the FreeRTOS operating system to coordinate tasks for each module.

A diagram of a diagram

AI-generated content may be incorrect.

Figure X: Simplified View of the State Machines of Modules Coordinating Data Collection

The above figure illustrates a simplified version of the state machines configured for the main modules of the system. Each sensor ran in its own task (or thread), and put its valid results onto the measurement queue every ten minutes. The measurement task initialized the file system, directory and daily log file on the MicroSD card, then serviced the queue and flushed the recorded data (with their timestamps) to the daily csv log file.

A screen shot of a computer

AI-generated content may be incorrect.

Figure X: An illustration of the csv data persisted to disk

The figure above shows a snippet of the data that was collected in a single CSV file. In the following section, we discuss how we aggregated, cleaned and standardized data from the entire collection of files.

## Data Preprocessing

Several data pre-processing steps needed to be implemented on the raw CSV sensor data before it could be used to train our ML models. Figure X illustrates these steps.

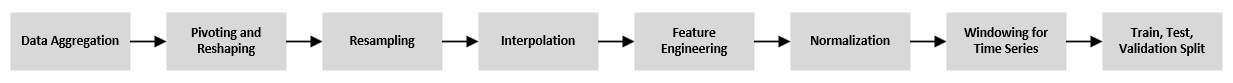


Figure X: An overview of the data preprocessing stage

### Data Aggregation

Since each CSV represents one 24-hour period of environmental sensor data, the foundational step in preparing the data for our ML models involved concatenating all CSVs into a single, unified dataset. This concatenation made it far simpler to perform downstream data processing steps, and to pass the full dataset to the ML models.

A diagram of a data flow

AI-generated content may be incorrect.

Figure X: Aggregating daily CSV files into a unified dataset

### Pivoting and Reshaping

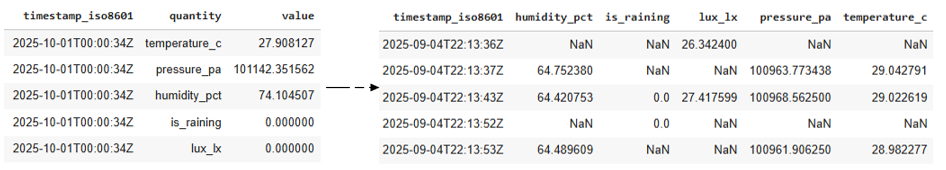
****

Figure X: Pivoting the unified dataset – ensuring each feature is a column

ML models typically require one row of data to represent one instance of the dataset. For our timeseries forecasting task, this means that each row should indicate the state of each feature at that timestamp. Pivoting the unified dataset ensures that for each timestamp, the value of each feature is represented as a column (Figure X RHS), as opposed to one row representing one feature’s value at each timestamp (Figure X LHS).

### Resampling

Resampling involves converting a sequence of data points from one sampling rate to another. The right-hand side of Figure X above indicates why resampling was a necessary data preprocessing step in our methodology. Since our initial timestamps go down to the granularity of seconds, if there is a slight difference (a few seconds) between the time the BME280 sensor captures humidity, pressure and temperature readings, and when the VML7700 sensor captures light intensity, this will be represented as two separate rows (e.g. 2025-09-04T22:13:36Z and 2025-09-04T22:13:37Z). In practice, we intend for these readings to represent values for the same time period, thus, we must resample to a less granular interval. We resampled the data from 1-second intervals to 30-minute intervals. Within each 30-minute interval, there are up to 1800 readings from each sensor, so to get one representative value for the resampled 30-minute interval, we take the **mean** of all the 1-second interval values within a given 30-minute interval.

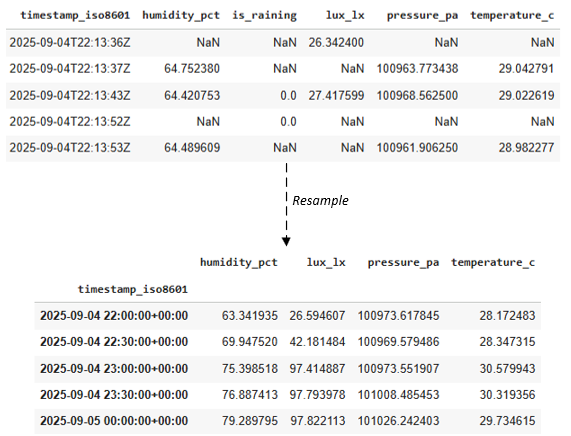


Figure X - Resampling from a 1-second interval to a 30-minute interval

### Interpolation

During the data collection period, the sensors did not always capture data continuously. Periodic interruptions, such as those required for additional configuration or maintenance, necessitated temporarily disconnecting the board and sensors from power. As a result, some 30-minute intervals contain no recorded measurements. To address these gaps and maintain a consistent time series for input into the ML models, we applied interpolation techniques to estimate missing values within these intervals. To achieve this, we used simple linear interpolation; for each missing value, we find the nearest valid data point before and after the gap, then estimate the missing value and a point on the straight line connecting the two known values. This process is done independently for each feature.

A graph of weather forecast

AI-generated content may be incorrect.

Figure X - Temperature and Humidity Dataset Before and After Interpolation

#### Feature Engineering

Several custom features were engineered from the raw sensor data to enhance the predictive performance of the ML models.

* Cyclical Time Features:  
  The sine and cosine of the hour of the day were computed to capture daily periodicity and represent time as cyclical variables. This ensures that times such as 23:00 and 0:00 are recognized as much more temporally adjacent than, say, 23:00 and 15:00, thus capturing relevant diurnal cycles.   
  The sine of the hour of day is calculated as:  
  The cosine of the hour of day is calculated as:
* Temperature Delta:   
  For each timestamp, the difference between the current temperature and the temperature at the previous timestamp was calculated as where:  
   is the temperature delta at time T. is the temperature at the current timestamp is the temperature at the previous timestamp  
    
  This provides the model with information about shorter-term temperature changes, enabling prediction of rapid fluctuations in weather conditions.
* Rolling means:   
  6-hour rolling averages were computed for temperature and humidity to provide context on recent trends.  
  where:  
   is the rolling average at time t is the value of the variable at time t - i is the number of time steps in 6 hours (for 30-minute intervals, N = 12)

#### Normalization

We used Z-score normalization (standardization) to scale every feature except the target feature (temperature) into a common scale, without distorting differences in value ranges.

The Z-Score for each feature is calculated as:

.

In the context of our experiments, the initial features in the raw data had significantly different scales. Normalization ensured that each feature contributed equally to the learning processes of the ML models, i.e. this prevented larger ranges from dominating. Additionally, Cabello-Solorzano et. al demonstrated that Z-score normalization improves accuracy across multiple ML algorithms, including neural networks (2023). Figure X below highlights the difference in the scale of the humidity feature before and after Z-score normalization.

A screenshot of a graph

AI-generated content may be incorrect.

Figure X: Illustration of input feature normalization

#### Windowing for Time Series

For this time series forecasting task, the data was structured into fixed-length input and output sequences. Each input sequence was comprised of a 24-hour sequence of historical observations, while the corresponding output sequence represented the subsequent 12 hours of temperature forecasts. (i.e. we used the previous 24 hours to predict the next 12 hours of temperature). Since we resampled to 30-minute intervals, this meant that 48 instances were used to compose each input window, and 24 instances were used to compose each output window.

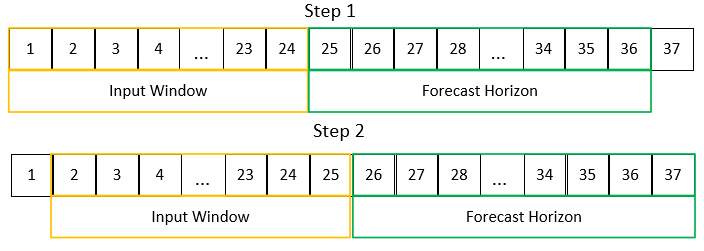


Figure X: Examples of input and output interval slots used for windowing

## API Data Collection

Unfortunately, we were only able to collect approximately 2 months of valid sensor data from local sensors in the rural location in Couva. We therefore used the OpenMeteo Historical API to augment our sensor data to create a full year of overall training data. In order to match the sensor data, we used the following features available from the API:

* **Temperature** (Air temperature at a 2M elevation above the ground)
* **Relative Humidity** (Relative humidity at a 2M elevation above the ground)
* **Shortwave Radiation** (Shortwave solar radiation)
* **Surface Pressure** (Atmospheric air pressure reduced to pressure at surface)

Once the input dataset was downloaded, the same preprocessing steps discussed above were performed. The two datasets were then merged to form the complete training dataset.

## ML Model Building

### Convolution Neural Network

Since our literature review showed that CNNs were a good choice for a balance between model forecasting performance and hardware resource constraints, we decided that it would be a suitable first architecture for our use case.

A diagram of a layer

AI-generated content may be incorrect.

Figure X: The CNN architecture we created to forecast temperature for a 12-hour window

Our input layer consisted of a feature vector of size [1x48x7], with the first dimension (1) representing a single batch of time sequences (in 30min batches), the second dimension (48) representing the historical size of the time slots in the window, and the third dimension (7) representing the number of features of our input data used in each time step.

The input layer feeds into the first convolution block, which consists of 2 convolution layers followed by a max pooling layer. The first convolution layer consists of 24 filters with a kernel size of 5 and a ReLU activation function, allowing the model to learn low-level temporal patterns across all the input features over a series of 5 time-steps, while keeping the length of the time-axis constant.

The second convolution layer consists of 24 filters with a kernel size of 3 and a ReLU activation function, enabling the model to learn higher-level patterns about our data by extending the first layer’s representations. The max pooling layer then computes the maximum across every 2 time-steps, reducing the temporal size of the feature maps produced by the convolution layers, thereby ensuring that the computational cost of the following layers remains manageable.

The data then flows into a second convolution block, beginning with two separable convolution layers. Both of these consist of 64 filters with a kernel of size 3 and ReLU activation layers as well. These layers break standard convolution (as performed in the first layers) into depthwise and pointwise operations, examining features along each channel independently, then mixing the channels together in a linear combination. This allows us to create more channels for richer representations of patterns across the training data, while keeping the number of parameters and MACs lower than a standard convolution layer. These are very applicable at this stage in the CNN because each convolution layer produces a large number of output channels to represent the features learned.

The data then flows to another max pooling layer (to compute the max across every 2 time steps again), and then a global average pooling layer, which computes averages of each feature map learnt (i.e. the new channels learnt in the previous steps) across the temporal axis, allowing us to find significant features while keeping the number of parameters comparatively low.

We then introduce a dense projection layer with 96 units, which combines the features learnt in the previous steps to find patterns and relationships among the past feature data to determine the most appropriate forecast values for the future.

The dropout layer prevents overfitting (a phenomenon where models become too closely tied to their training dataset and thus perform poorly on other, newer datasets) by randomly assigning 10% of the weights in the model to 0.

Finally, the dense output layer produces our forecasted temperature values, with the number of output units equal to 24 for our final testing, allowing us to forecast 12 hours of data (24 forecast slots, each at 30 min time intervals).

### Recurrent Neural Network

Much of the deep learning literature suggests that RNNs are a strong candidate model for the use-case presented in this paper, as RNNs are a class of neural networks specifically designed to process sequential data (Rumelhart et al. 1986). Unlike a regular feedforward neuron or a convolutional neuron, a recurrent neuron receives inputs, produces an output, and sends that output back to itself. At each time step , the recurrent neuron takes the current input , and its own output from the previous time step . To determine the RNN architecture, we employed a randomized search methodology which searched for the combination of hyperparameters which minimized the MAE of the model. To perform this random search, we specified the search spaces for the hyperparameters in Table Y. Table Y also shows the results of this random search.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Hyperparameter** | **Hyperparameter Description** | **Purpose** | **Search Space** | **Optimal Value** |
| n\_hidden | Specifies the number of additional recurrent layers stacked after the first RNN layer | Controls depth of temporal feature extraction |  | 1 |
| n\_neurons | Defines the number of neurons in each additional recurrent layer | Determines model’s capacity to learn complex patterns | (integer) | 27 |
| optimizer | The algorithm used to update model weights during training | Influences convergence speed | {sgd, adam} | adam |
| learning\_rate | A scalar value that controls the step size during gradient descent | Balances convergence speed against the risk of overshooting loss function minimum | (log) | 0.00905 |

Figure X illustrates the architecture of the final model.

The input layer accepts a sequence length of 48 instances (*48 time steps = 24 hours at 30-min sampling rate*). Layer 1 is a SimpleRNN layer with 32 neurons. This layer extracts temporal features, then passes the full sequence to Layer 2. Layer 2 is another SimpleRNN with 27 neurons, as determined via the optimization performed by the randomized search. Finally, the output layer provides a linear projection to the 24-time-step temperature forecast horizon (*24 time steps = 12 hours at 30-min sampling rate*).

A diagram of a algorithm

AI-generated content may be incorrect.

Figure X: RNN architecture

## ML Model Pruning & Quantization

A diagram of a diagram

AI-generated content may be incorrect.

Figure X: Our pruning process to reduce the space and computations of the initial full model

To compress our model, we then implemented structured pruning as described in the figure above and our literature review.

Instead of trying to prune individual weights, entire channels were pruned by computing their L1 norms. The L1 norm allows us to calculate a simple scalar number that represents the relative important of a channel in a CNN layer, and is defined as:

Where is the L1 norm and is the number of weights in a particular channel

Channels/filters with lower L1 norms can be interpreted as being less important to the overall model, and are thus good candidates for pruning.

For each convolution and separable convolution layer, we defined a keep ratio, specifying what percentage of channels we wanted to keep in each layer. The earlier layers were given higher keep ratios as they were found to be more crucial for feature mapping.

Several candidates of keep ratios were then configured, with progressively more aggressive pruning for each candidate. Each candidate was then assessed in terms of both MAE increase as well as size and computation decrease, to determine which one produced the best overall result.

Once the pruning was completed, we realigned the layers to ensure that the dimensions were correct within and between each layer.

At this point, the model consisted of weights and activations represented by 32-bit floating point numbers. To reduce its size and computational complexity even further, a large subset of the training and validation data were assessed to determine their full dynamic range and thus calculate appropriate step intervals. These steps were then utilized to quantize the weights and activations to 8-bit integer values, which are much easier for the majority of microcontrollers to manage. Additional functions were also added to normalize and quantize the input and output data values passed to and from the model, so that all MAC operations could be done with solely integers.

## ML Model Deployment

The quantized model was then loaded into STM32 Cube.AI package for conversion into a C library, so that it could be integrated with the rest of the FreeRTOS code.

Two Tasks were created to perform inference on the system, and subsequently log the results to the MicroSD card for later analysis.

A diagram of a diagram

AI-generated content may be incorrect.

Figure X: The Forecast Temp Task which handled the bulk of the inference work on the board

The first task, named the **Forecast Temp Task**, is illustrated above.

At startup, pointers are initialized to the input and output buffers of the model. The SD card is then mounted in the filesystem, and past measurements are used to partially reconstruct the 24-hour window, so that we don’t need to wait 24 hours to get an inference reading on each reboot of the board.

The latest data from the sensors are then fetched every 1 minute, until we have a full 30-min window of sensor samples. Once this 30-min window is filled, the extra engineered features are also computed. Then all 9 features are normalized and pushed onto a new slot in the 24-hour inference window.

This occurs repeatedly until the 24-hour window is filled. At that point, the entire window is quantized and exported, then fed into the model produced by STM32 Cube.AI. The output of the model is then denormalized and dequantized to get the true inferred value for temperatures.

The array of 30-min inferred values, for the upcoming 12 hours of temperature predictions are then stored in a context variable for reading by other tasks.

A diagram of a system

AI-generated content may be incorrect.

Figure X: The Inference Logger Task which handled the periodic logging to the MicroSD card

The second task, called the **Inference Logger task** handles the reading of the latest predicted/inferred values from the Forecast Temp Task and its writing into the SD card. Upon initialization, the task ensures that the SD card is mounted in the filesystem and the expected directory is created. It then creates an inference CSV file to store all inference values recorded in a single day.

Every 5 mins, this task then polls the Forecast Temp Task for the current cached inference value, and writes it to the current daily CSV. The task also manages error conditions and daily file rotations.

## Power and Inference Time Measurement

We implemented a simple system to compute an approximation for the inference time for each model, by recording the time before the Forecast Temp Task is called, then the time after the function call is completed. The inference time (in ms) is taken as this difference in time.

We also implemented a simple system to approximate the total power draw of the embedded system by utilizing an in-wall multi-meter, and sampling the AC voltage, current and power factor in 5-min intervals.

# **Results and Discussion**

Our models produced relatively good results, given the time and hardware constraints imposed.

## Model Compression

The following table summarizes the size and computation reductions of each model through each stage.

|  |  |  |
| --- | --- | --- |
|  | **CNN** | **RNN** |
| Initial model size (kBs) | 49.72 | 105.91 |
| Initial # of parameters | 12,728 | 19,032 |
| Initial # of MACs | 200,064 | 644,652 |
| Initial test MAE (°C) | 1.38 | 1.63 |
| Model size after pruning (kBs) | 46.53 | 102.05 |
| # of parameters after pruning | 11,911 | 18,943 |
| # of MACs after pruning | 169,272 | 644,563 |
| Test MAE after pruning (°C) | 1.34 | 1.68 |
| Model size after quantization (kBs) | 30.77 | 80.88 |
| # of parameters after quantization | 11,911 | 18,943 |
| # of MACs after quantization | 169,272 | 644,563 |
| Test MAE after quantization (°C) | 1.44 | 1.70 |

Table X: Summary of model compression after each stage

For the **CNN**, we saw that the model size was reduced by **6.4%** after pruning, and a further **38.1%** after quantization. The number of parameters and number of MACs were both also reduced by **6.4%** after pruning, and stayed consistent after quantization. We actually saw a slight decrease in MAE after pruning **(-2.9%),** but then this was followed by a **4.3%** increase in MAE after quantization.

For the **RNN**, we saw that the model size was reduced by **3.6%** after pruning, and a further **23.6%** after quantization. The number of parameters and number of MACs were both also reduced by **0.5%** after pruning, and stayed consistent after quantization. We saw an increase in MAE after pruning of **3.1%,** followed by a **1.2%** increase in MAE after quantization.

Comparing the RNN to the CNN, we saw that the final model size ~2.6x higher, the number of parameters was ~1.6x higher, and the # of MACs was ~3.8x higher.

## Model Performance

## Convolutional Neural Network

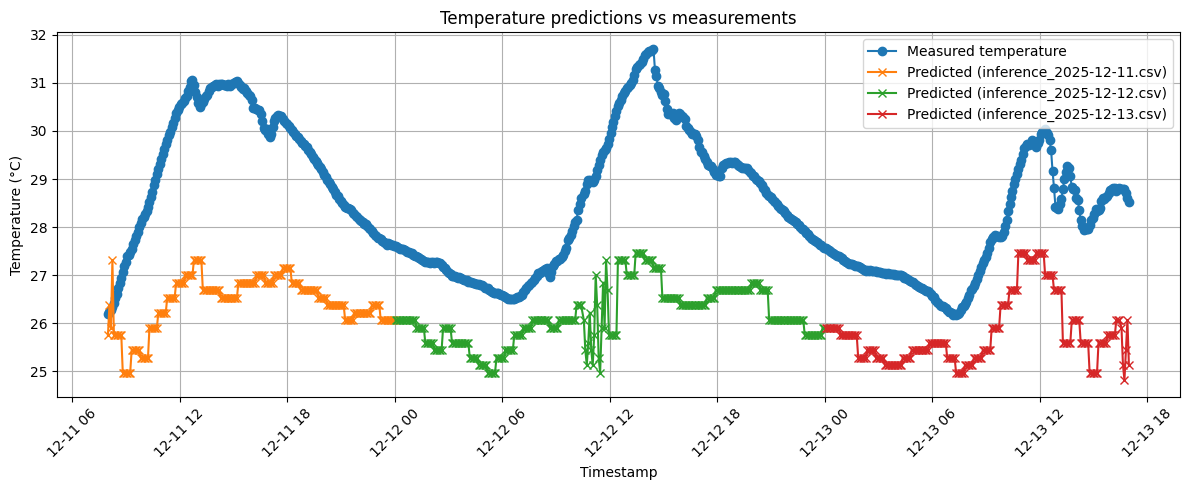


Figure X: CNN model predictions vs actual measurements

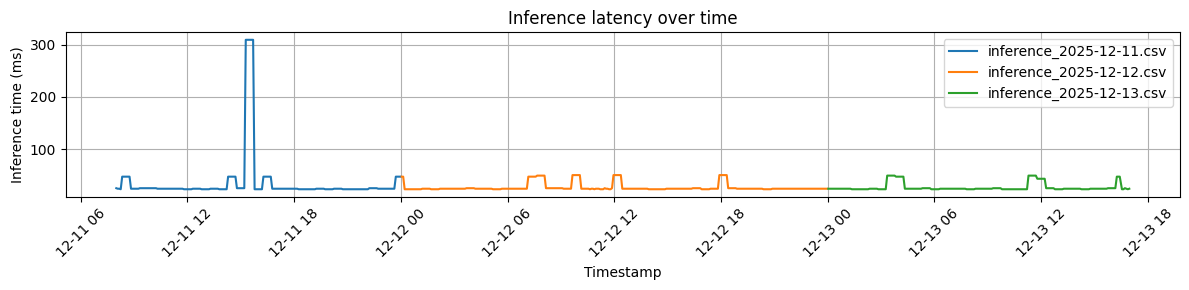


Figure X: CNN model inference latency

The mean inference latency was found to be 30.1 ms, and the median was found to be 25.0ms.

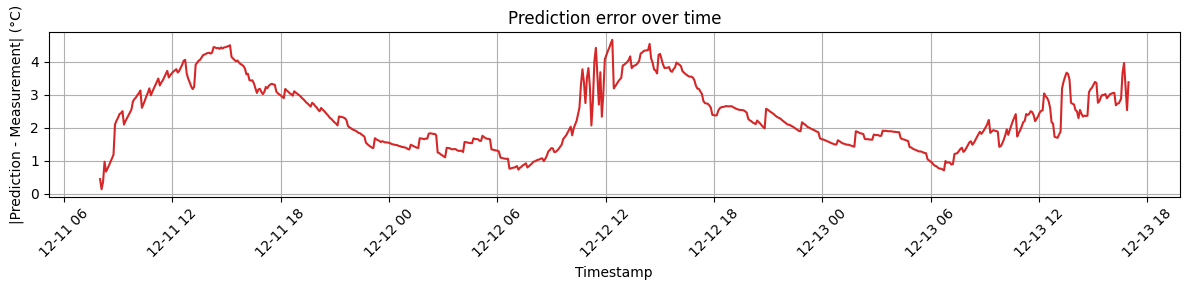


Figure X: CNN model prediction error

The mean prediction error was found to be 2.4°C, while the median was found to be 2.2°C.

## Recurrent Neural Network

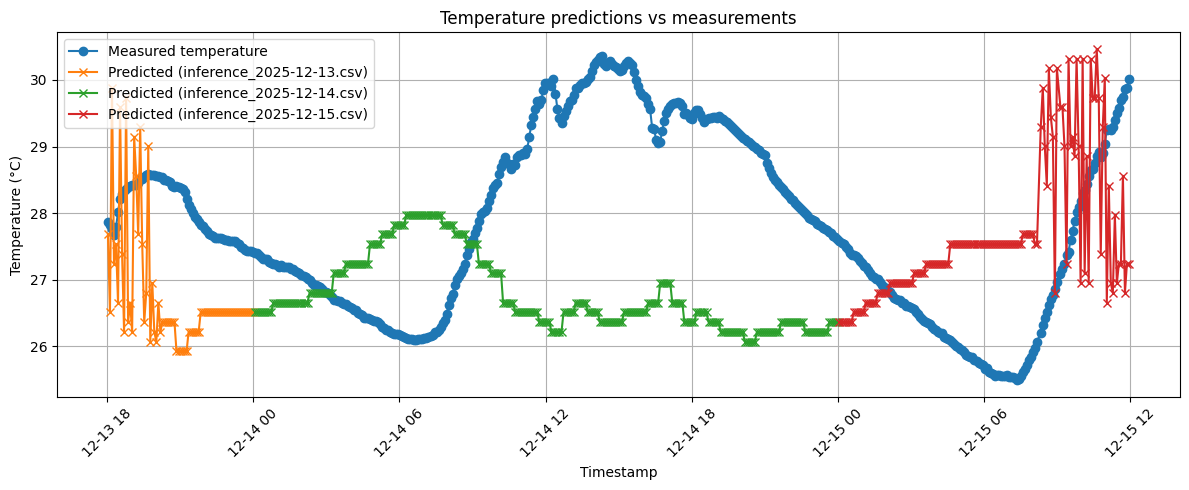


Figure X: RNN model predictions vs actual measurements

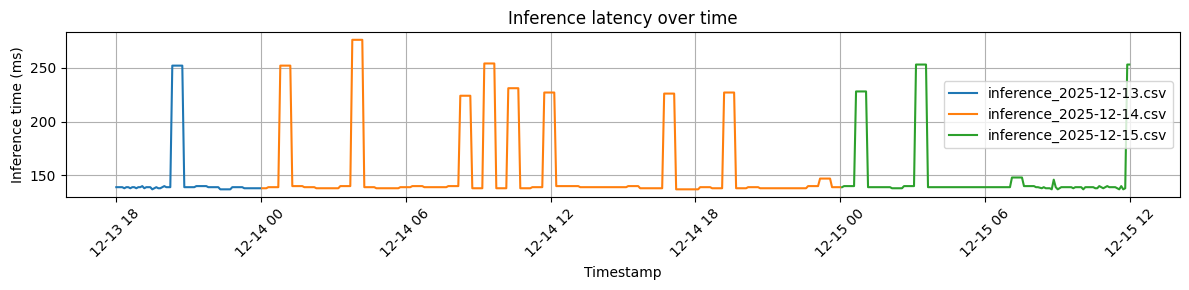


Figure X: RNN model inference latency

The mean inference latency was found to be 152.8 ms, and the median was found to be 139.0ms.

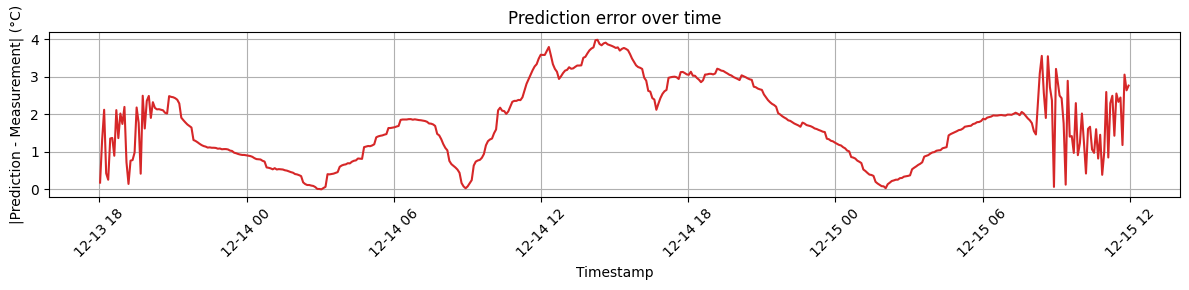


Figure X: RNN model prediction error

The mean prediction error was found to be 1.8°C, while the median was found to be 1.7°C.

## Power Measurements

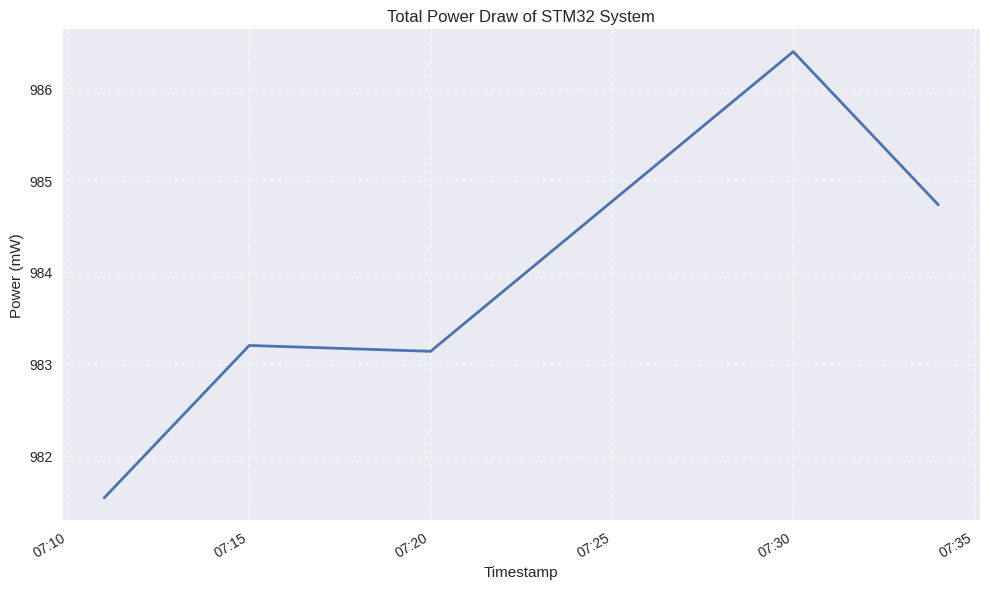


Figure X: Total system power draw during inference

We saw no significant statistical variation in total power draw for our embedded system during inference with either model. As illustrated in the graph above, power consumption seemed fairly constant across all sampling intervals. Across all samples, we measured a mean of 984mW, and a standard deviation of 1.8mW, meaning that we saw less than a 0.2% variation in total power consumption across all measurements. It is possible that a multi-meter with additional measurement capabilities (with automatic, more-frequent sampling and a current scale in the range of mA) would uncover more variations which could be tied to inference.

## Discussion

# **Conclusion And Future Work**

Given the time and hardware limitations required, our final CNN and RNN models performed relatively well and fit easily in the SRAM of the STM32 board. The power required to perform inference on the board was not statistically significant over the baseline power required to operate the regular functions.

We were able to deploy a CNN with a final model size of 30.8 kBs. an average test MAE of 1.4 °C and an average on-device deployment MAE of 2.4°C. We were also able to deploy an RNN with a final model size of 80.9 kBs an average test MAE of 1.7°C and an average on-device deployment MAE of 1.8°C.

We found that the RNN was able to achieve a lower MAE after deloyment, but required more space in SRAM than the CNN. Therefore, either model could be suitable, depending on the specific hardware requirements of the project.

In order to improve this work further, we would recommend capturing training data over an entire year or more at the intended deployment location of the system, for a much richer and more localised training set.

We would also recommend spending additional time iterating through different model architectures (with extra layers, different quantization ranges etc.) and evaluating the results after deployment.

Additionally, we would recommend including more input sensors to enable predictions of additional features. For example, an anemometer and a puviometer could be added to record input features for wind speed and rainfall amounts respectively.

# **References**

<https://www.bibcitation.com/s/S4Cgz2uDtx>

Abadade, Y., Temouden, A., Bamoumen, H., Benamar, N., Chtouki, Y., Hafid, A.S., 2023. A Comprehensive Survey on TinyML. IEEE Access 11, 96892–96922. https://doi.org/10.1109/access.2023.3294111

Ben Bouallègue, Z., Clare, M.C.A., Magnusson, L., Gascón, E., Maier-Gerber, M., Janoušek, M., Rodwell, M., Pinault, F., Dramsch, J.S., Lang, S.T.K., Raoult, B., Rabier, F., Chevallier, M., Sandu, I., Dueben, P., Chantry, M., Pappenberger, F., 2024. The Rise of Data-Driven Weather Forecasting: A First Statistical Assessment of Machine Learning–Based Weather Forecasts in an Operational-Like Context. Bulletin of the American Meteorological Society 105, E864–E883. https://doi.org/10.1175/bams-d-23-0162.1

Cabello-Solorzano, K., Ortigosa de Araujo, I., Peña, M., Correia, L., J. Tallón-Ballesteros, A., 2023. The Impact of Data Normalization on the Accuracy of Machine Learning Algorithms: A Comparative Analysis, in: Lecture Notes in Networks and Systems. Springer Nature Switzerland, Cham, pp. 344–353.

Codeluppi, G., Davoli, L., Ferrari, G., 2021. Forecasting Air Temperature on Edge Devices with Embedded AI. Sensors 21, 3973. https://doi.org/10.3390/s21123973

Copernicus Climate Change Service, Climate Data Store, 2018. ERA5 hourly data on single levels from 1940 to present [WWW Document]. Climate Data Store. URL https://cds.climate.copernicus.eu/doi/10.24381/cds.adbb2d47 (accessed 9.7.25).

de Burgh-Day, C.O., Leeuwenburg, T., 2023. Machine learning for numerical weather and  climate modelling: a review. Geoscientific Model Development 16, 6433–6477. https://doi.org/10.5194/gmd-16-6433-2023

Deutel, M., Woller, P., Mutschler, C., Teich, J., 2022. Energy-efficient Deployment of Deep Learning Applications on Cortex-M based Microcontrollers using Deep Compression [WWW Document]. arXiv.org. URL https://arxiv.org/abs/2205.10369

Diaz-Iglesias, A., Belaunzaran, X., Florez-Tapia, A.M., 2025. Short-Term Power Demand Forecasting for Diverse Consumer Types to Enhance Grid Planning and Synchronisation [WWW Document]. arXiv.org. URL https://arxiv.org/abs/2506.04294

Døskeland, Ø., Gudmestad, O.T., Moen, P., 2023. Use of response forecasting in decision making for weather sensitive offshore construction work. Ocean Engineering 287, 115896. https://doi.org/10.1016/j.oceaneng.2023.115896

Dueben, P.D., Bauer, P., 2018. Challenges and design choices for global weather and climate models based on machine learning. Geoscientific Model Development 11, 3999–4009. https://doi.org/10.5194/gmd-11-3999-2018

Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep Learning. MIT Press.

Guo, Q., He, Z., Wang, Z., 2024. Monthly climate prediction using deep convolutional neural network and long short-term memory. Scientific Reports 14. https://doi.org/10.1038/s41598-024-68906-6

Han, S., Mao, H., Dally, W.J., 2015. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding [WWW Document]. arXiv.org. URL https://arxiv.org/abs/1510.00149

Hasanpour, M.A., Kirkegaard, M., Fafoutis, X., 2025. EdgeMark: An automation and benchmarking system for embedded artificial intelligence tools. Journal of Systems Architecture 167, 103488. https://doi.org/10.1016/j.sysarc.2025.103488

Heydari, S., Mahmoud, Q.H., 2025. Tiny Machine Learning and On-Device Inference: A Survey of Applications, Challenges, and Future Directions. Sensors 25, 3191. https://doi.org/10.3390/s25103191

Immonen, R., Hämäläinen, T., 2022. Tiny Machine Learning for Resource-Constrained Microcontrollers. Journal of Sensors 2022, 1–11. https://doi.org/10.1155/2022/7437023

Jadon, A., Patil, A., Jadon, S., 2024. A Comprehensive Survey of Regression-Based Loss Functions for Time Series Forecasting, in: Lecture Notes in Networks and Systems. Springer Nature Singapore, Singapore, pp. 117–147.

Kelemen, M., Miková, Ľ., Hroncová, D., Filakovský, F., Sinčák, P.J., 2020. EMBEDDED SYSTEMS – CONTROL OF POWER SUBSYSTEMS. Acta Mechatronica 5, 23–28. https://doi.org/10.22306/am.v5i2.64

Kumar, P., Chandra, R., Bansal, C., Kalyanaraman, S., Ganu, T., Grant, M., 2021. Micro-climate Prediction - Multi Scale Encoder-decoder based Deep Learning Framework, in: Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery &amp; Data Mining. ACM, New York, NY, USA, pp. 3128–3138.

Lara-Benítez, P., Carranza-García, M., Riquelme, J.C., 2021. An Experimental Review on Deep Learning Architectures for Time Series Forecasting. International Journal of Neural Systems 31, 2130001. https://doi.org/10.1142/s0129065721300011

Morales-García, J., Bueno-Crespo, A., Martínez-España, R., Posadas, J.-L., Manzoni, P., Cecilia, J.M., 2023a. Evaluation of low-power devices for smart greenhouse development. The Journal of Supercomputing 79, 10277–10299. https://doi.org/10.1007/s11227-023-05076-8

Morales-García, J., Bueno-Crespo, A., Martínez-España, R., Posadas, J.-L., Manzoni, P., Cecilia, J.M., 2023b. Evaluation of low-power devices for smart greenhouse development. The Journal of Supercomputing 79, 10277–10299. https://doi.org/10.1007/s11227-023-05076-8

Naseer, I., Akram, S., Masood, T., Jaffar, A., Khan, M.A., Mosavi, A., 2022. Performance Analysis of State-of-the-Art CNN Architectures for LUNA16. Sensors 22, 4426. https://doi.org/10.3390/s22124426

OpenMeteo API [WWW Document], 2025. . Open-Meteo.com. URL https://open-meteo.com/ (accessed 12.19.25).

Osman, A., Abid, U., Gemma, L., Perotto, M., Brunelli, D., 2022. TinyML Platforms Benchmarking, in: Lecture Notes in Electrical Engineering. Springer International Publishing, Cham, pp. 139–148.

Perumal, T., Mustapha, N., Mohamed, R., Shiri, F.M., 2024. A Comprehensive Overview and Comparative Analysis on Deep Learning Models. Journal on Artificial Intelligence 6, 301–360. https://doi.org/10.32604/jai.2024.054314

Prudden, R., Adams, S., Kangin, D., Robinson, N., Ravuri, S., Mohamed, S., Arribas, A., 2020. A review of radar-based nowcasting of precipitation and applicable machine learning techniques [WWW Document]. arXiv.org. URL https://arxiv.org/abs/2005.04988?utm\_source=chatgpt.com

Rumelhart, D.E., Hinton, G.E., Williams, R.J., 1986. Learning representations by back-propagating errors. Nature 323, 533–536. https://doi.org/10.1038/323533a0

Shen, D., Zhao, X., Chai, L., Guo, Z., Leng, C., 2024. Analysis of the agricultural economic value of a weather forecasting service based on a survey of peasant households in Chinese provinces. Humanities and Social Sciences Communications 11. https://doi.org/10.1057/s41599-024-02685-3

Shiri, F.M., Perumal, T., Mustapha, N., Mohamed, R., 2023. A Comprehensive Overview and Comparative Analysis on Deep Learning Models: CNN, RNN, LSTM, GRU [WWW Document]. arXiv.org. URL https://arxiv.org/abs/2305.17473

ST Edge AI Developer Cloud [WWW Document], 2023. . STMicroelectronics - STM32 AI. URL https://stm32ai.st.com/st-edge-ai-developer-cloud/ (accessed 10.7.25).

Sze, V., Chen, Y.-H., Yang, T.-J., Emer, J.S., 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. Proceedings of the IEEE 105, 2295–2329. https://doi.org/10.1109/jproc.2017.2761740

Warden, P., Situnayake, D., 2019. TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers. O’Reilly Media.