Evaluating the Deployment of Microclimate Prediction Models On Embedded Systems

# **Abstract**

This paper explores the implementation of a machine learning model on a low-power embedded system, for the purpose of microclimate monitoring in an outdoor Caribbean environment. The goal is to evaluate the feasibility, challenges, techniques and trade-offs of deploying such models in extremely resource-constrained settings. The paper considers model accuracy alongside hardware and communication limitations, power budgets and environmental conditions.

# **Keywords**

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

# **Introduction**

Accurate weather forecasting is critical for the majority of tasks that we undertake in a wide variety of areas of modern society. In the context of an individual, weather forecasts determine the type of clothing worn, the type of outdoor activities planned and the mode(s) of transportation chosen for travel for work or leisure.

On a larger scale, however, these forecasts become even more crucial. In the field of power demand forecasting, for example, weather is closely tied to energy consumption patterns. For example, weather variables significantly impact the need for heating, cooling and lighting (Diaz-Iglesias et al., 2025, p. 4). 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, p. ,y). 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).

As highlighted by Kumar et al. (2021), the commercial weather stations which capture the data for these NWP may be located a significant distance away from regions of interest, leading to degradations in climatic patterns and poor forecasting performance. Microclimates describe the climatic parameters (such as temperature, humidity and precipitation) in and around a homogenous, smaller zone of interest. Improving prediction for these microclimates could result in substantial economic gains in an assortment of fields such as offshore-platform activity planning, commercial fishing scheduling, and large-scale agricultural crop rotations.

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. The figure below illustrates the basic architecture of a neural network. 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.

Neural Network 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 (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 servers commonly referred to as 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.

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). By running these models on these embedded devices at the edge, 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 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

Sze et al. presented a thorough background and efficiency analysis of DNNs (2017). They first discussed the basic computations involved in NNs, which we will now summarize.



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

In the above figure, they 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. The true output Y for several input data points are usually labelled prior, and the accuracy of the NN is computed by measuring the difference between the predicted output Ŷ and the true output Y, using a loss function.

Note that these multiply and accumulate (MAC) operations between X, w and b 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.

To evaluate the performance of the model, two main loss functions are used:

Mean Absolute Error (MAE):

and Mean Squared Error (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. 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).

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 can be performed on embedded devices. Since this is the stage at which TinyML can truly be leveraged, we will focus our discussions mainly on this second stage.

## 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 shows the typical requirements of some popular architectures.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Metric** | **LeNet5** | **AlexNet** | **Overfeat fast** | **VGG 16** | **GoogLeNet v1** | **ResNet 50** |
| Top-5 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).

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

The field of TinyML has created no single framework or library to implement 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 interference 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 normalized data from the entire collection of files.

## Sensor Data Preprocessing

## ML Model Building

## ML Model Pruning, Quantization and Evaluation

## TinyML Model Deployment

# **Results and Discussion**

Placeholder text...

# **Conclusion**

Placeholder text...

# **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

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

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

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

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.