

Implementation of On-Device Convolutional Neural Network Inference for Indoor Occupancy Detection on the ESP32-S3

Ade Aditya Ramadha¹, Felix Corputty², Naufal Hanan Lutfianto³

^{1,2,3} Telecommunication Engineering Study Program, School of Electrical Engineering, Telkom University, Jl. Telekomunikasi no. 1, Bandung 40257, West Java, Indonesia

ARTICLE INFO

Article history:

Received : 08/07/2026

Revised : 21/08/2026

Accepted : 29/08/2026

Keywords:

Convolutional Neural Network; Edge Computing; ESP32-S3; INT8 Quantization; Occupancy Detection

ABSTRACT

Indoor occupancy detection is a key enabling technology for energy-efficient smart building management, as heating, ventilation, air conditioning, and lighting systems waste significant energy when conditioned spaces are unoccupied. This paper implements a compact Convolutional Neural Network (CNN) trained on the UCI Occupancy Detection Dataset and deployed entirely on the ESP32-S3 microcontroller using TensorFlow Lite Micro with INT8 post-training quantization, eliminating any cloud dependency. Models are trained under a leakage-controlled protocol on the designated training file, selected on the first test file, and evaluated on the held-out second test file; the INT8 quantizer is calibrated from an evenly spaced spread of training windows to avoid class bias. Twelve configurations spanning four filter capacities, $(F_1, F_2) \in \{(4,8), (8,16), (16,32), (32,64)\}$, and three temporal window lengths, $W \in \{15, 30, 60\}$ minutes, are quantized, flashed, and measured on the device for tensor arena, flash footprint, inference latency, and held-out F1. All twelve configurations execute successfully: the best compact model, CNN(8,16) at a fifteen-minute window, reaches a held-out F1 of 0.959 using 4.2 KB of tensor arena and 1.24 ms per inference, with more than 320 KB of heap remaining free; the tensor arena across the full grid stays between 2.6 and 30 KB; and INT8 quantization changes F1 by at most ± 0.021 relative to the floating-point baseline. These results demonstrate that on-device CNN occupancy detection is practical on a low-cost microcontroller, that a compact two-block architecture is sufficient to reach near-optimal accuracy, and that the full pipeline from leak-free training through representative INT8 calibration to on-device measurement is reproducible.



This work is licensed under a [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/)

Corresponding Author:

Ade Aditya Ramadha

Telecommunication Engineering Study Program, School of Electrical Engineering, Telkom University, Main Campus (Bandung Campus), Jl. Telekomunikasi no. 1, Bandung 40257, West Java, Indonesia

Email: adeadityaramadha@telkomuniversity.ac.id

1. INTRODUCTION

Smart buildings are among the most productive applications of the Internet of Things (IoT) paradigm because the energy savings they enable are large and the sensing infrastructure they require is genuinely low-cost. Heating, ventilation, air conditioning, and lighting together account for a substantial fraction of commercial building energy consumption, and much of that energy is wasted on spaces that no one is using [1],[2]. Occupancy-driven control addresses this problem directly: instead of running systems on fixed schedules, a building management system responds to actual demand. Candanedo and

Feldheim [1] showed that ambient measurements of temperature, humidity, CO₂, and light are sufficient for reliable binary occupancy classification, and that result remains the foundation for much of the subsequent work in this area.

The challenge in this field has shifted from whether ML-based occupancy detection can work to where inference should happen. Most published solutions route classification to a server or cloud backend, treating the edge microcontroller as a data relay. This design is functional in well-controlled deployments, but it introduces latency that scales with network conditions, a dependency on continuous connectivity that many installations cannot guarantee, and the transmission of environmental data that some operators prefer to keep local [3],[4],[5]. Camera-based systems offer higher spatial resolution and suit multi-occupant scenarios [6], but they carry substantial compute requirements and surveillance concerns that exclude them from many buildings [2]. Inference performed entirely on-chip eliminates each of these drawbacks.

Tiny Machine Learning (TinyML) provides this capability, and the field has matured rapidly with more capable microcontrollers and improved compression techniques [7],[8]. Post-training INT8 quantization is the most practically important of these techniques: Ray [9] documents how it reduces memory footprint to a fraction of the floating-point original, and Elhanashi et al. [10] confirm that accuracy loss under a well-calibrated INT8 scheme is typically small across sensing tasks. Alajlan and Ibrahim [11] survey TinyML on ultra-low-power devices and show that environmental classification is well within current hardware limits, while Abadade et al. [12] identify occupancy sensing as one of the targets best aligned with TinyML. Sanchez-Iborra and Skarmeta [8] argue that such microcontrollers should be treated as autonomous decision nodes rather than data endpoints.

The ESP32-S3 is a strong candidate for this role. Its dual Xtensa LX7 cores include vector instructions suited to the multiply-accumulate operations that dominate convolution [13], it provides 512 KB of internal SRAM, and it runs TensorFlow Lite Micro, the embedded inference framework maintained for microcontroller-class devices [14]. The convergence of edge computing and artificial intelligence that Deng et al. [5] call edge intelligence frames why this matters at a system level. The MLPerf Tiny benchmark [15] standardizes INT8 TFLM inference at this scale, Osman et al. [16] benchmark TinyML frameworks across platforms, Rodriguez-Conde et al. [17] show that distributed DNN inference across IoT nodes is feasible, and Tsoukas et al. [18] identify on-device smart-environment sensing among the most promising near-term applications. Recent work has extended this to human sensing, including convolutional activity recognition on the ESP32-S3 from Wi-Fi channel state information [19].

This paper reports a complete, working implementation of CNN-based occupancy detection running entirely on the ESP32-S3 and documents the full pipeline from split-clean training to on-device measurement. The work makes three contributions. First, it provides a leakage-controlled, end-to-end pipeline: models are trained on the UCI training file, selected on the first test file, and evaluated on the second test file, with no window crossing a split boundary and the INT8 quantizer calibrated from a representative spread of training data. Second, quantized models deploy and execute on the ESP32-S3 under TensorFlow Lite Micro; measured tensor arena, flash footprint, inference latency, and accuracy are reported for each of twelve configurations. Third, it shows that a compact two-block CNN already reaches an F1 near 0.95 and that INT8 quantization is near-lossless, making a small network sufficient for practical deployment.

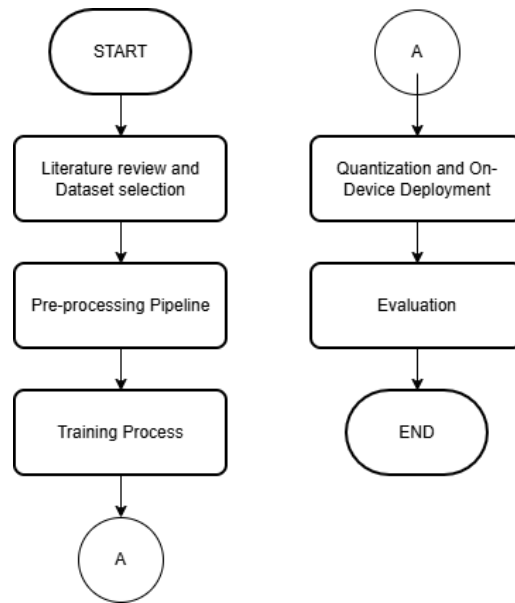


Figure 1. Research workflow

Table 1. Number of sliding windows generated per file at each window length

W (min)	Train windows	Validation windows	Test windows
15	8,128	2,650	9,737
30	8,113	2,635	9,722
60	8,083	2,605	9,692

2. RESEARCH METHOD

The research workflow adopted in this study is presented in Figure 1. The details of each stage in the research process are described in this section.

2.1. Dataset

This study uses the UCI Occupancy Detection Dataset [1], a public benchmark for binary room occupancy from ambient sensing. A single office room was instrumented for temperature, relative humidity, light intensity, CO₂ concentration, and humidity ratio, sampled once per minute, with ground truth from time-stamped photographs. The data is distributed as three separate files: a training file of 8,143 samples and two test files of 2,665 and 9,752 samples. This natural separation is used directly. Models are trained on the training file, hyperparameters are selected on the first test file, and all reported accuracy is measured on the second test file, which no model sees during training or selection. The datetime column is discarded because it carries information a deployed node could not use. Humidity ratio is retained, since Candanedo and Feldheim [1] showed it encodes a combined thermodynamic signal that aids classification. For on-device evaluation an evenly spaced subsample of windows from the held-out file was compiled into firmware as static C arrays, while the full file was used for the offline figures.

2.2. Preprocessing Pipeline

Five numerical features remain after the datetime column is removed. Each is scaled to the unit interval by min-max normalization whose minimum and maximum are computed on the training file alone and applied unchanged to both test files, so no test statistic leaks into preprocessing. A sliding window of length W is then formed with a step of one sample and reshaped to $(W, 5, 1)$ for a two-dimensional convolution, with the occupancy of the sample immediately following the window as its label. Three window lengths are studied, $W \in \{15, 30, 60\}$, corresponding to fifteen, thirty, and sixty minutes of history. Windows are built within each file independently, so none spans a split boundary. Because each window is labeled by the sample that follows it, the number of windows per file is its length minus W ; the resulting counts are listed in Table 1.

2.3. Model Architectures

The implementation uses a single compact CNN template and varies only its capacity. Each model takes the (W, 5, 1) window through a first convolutional block of F_1 filters of size 3×3 with same padding and ReLU activation, followed by 2×2 max pooling, then a second block of F_2 filters of the same size and activation. Global average pooling reduces the resulting feature map to a vector of length F_2 , which feeds a single sigmoid unit. The full data path is shown in Figure 2. Four capacities are implemented: (F_1 , F_2) of (4, 8), (8, 16), (16, 32), and (32, 64), corresponding to 345, 1,265, 4,833, and 18,881 parameters. Global average pooling is used in place of flattening so the classifier stays independent of window length [19]. Models were trained with Adam and binary cross-entropy for up to 100 epochs with early stopping on the validation file, restoring the weights of the lowest validation loss with a patience of fifteen epochs. Learning rate was tuned over {0.001, 0.010}, with the better of the two by validation F1 retained for each configuration.

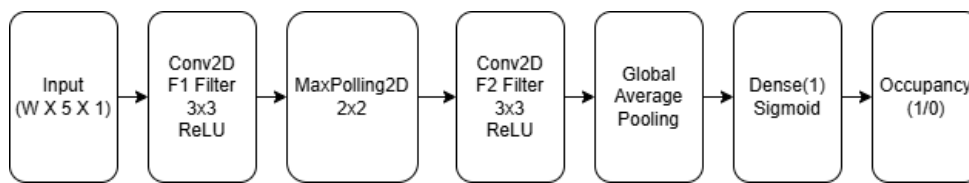


Figure 2. The two-block CNN used in all configurations.

2.4. Model Training

All training was performed offline on a host machine, and only the resulting quantized models were transferred to the device. The three dataset files were kept in their distributed roles throughout: the training file fitted the model weights, the first test file guided model selection, and the second test file was reserved for the final held-out measurement and never influenced any training decision. The min-max scaler was fitted on the training file alone, and its fixed minimum and maximum were applied to the validation and test files, so no statistic from the evaluation data entered the pipeline.

Each configuration was trained with the Adam optimizer against a binary cross-entropy loss, in mini-batches of thirty-two windows, for a maximum of one hundred epochs. Rather than train for a fixed length, the procedure monitored the loss on the validation file after every epoch and stopped early once it had not improved for fifteen consecutive epochs, restoring the weights from the epoch of lowest validation loss. This early-stopping rule served two purposes: it prevented the larger-capacity models from overfitting the training file, and it removed epoch count as a manual hyperparameter, since each configuration trained only as long as its own validation curve justified.

Two learning rates, 0.001 and 0.010, were tried for every configuration, and the one giving the higher validation F1 was retained. This was the only hyperparameter search performed; no tuning was carried out against the held-out test file. The same procedure was applied identically to all twelve capacity and window configurations, so that any difference in their reported accuracy reflects the model rather than a difference in how it was trained.

2.5. Quantization and On-Device Deployment

Each configuration was converted to an INT8 TFLite Micro model by post-training quantization under the TFLITE_BUILTINS_INT8 scheme [20]. The calibration set was two hundred windows drawn at even spacing across the entire training file rather than from its start, because the recording begins with a long unoccupied period and a contiguous early slice would expose the quantizer to a single class and collapse the model toward it. To check the quantization penalty, each integer model was also evaluated offline on the full held-out file using the same arithmetic the device applies. Quantized binaries were embedded as C arrays and flashed to the ESP32-S3 under ESP-IDF v5.5 and TensorFlow Lite Micro, one model per flash cycle, using a six-kernel resolver (Conv2D, MaxPool2D, Mean for the pooling stage, FullyConnected, Reshape, and Logistic); the Mean operator must be registered explicitly or inference fails at the first call. On the device, the tensor arena reported by the interpreter, the inference latency from `esp_timer_get_time()`, the free heap from `esp_get_free_heap_size()`, and the task stack high-water mark were recorded, and the device figures were gathered on an evenly spaced subsample of the held-out file and validated against the offline integer results.

Table 2. Implementation results measured on the ESP32-S3

Filters (F1,F2)	W	Params	Flash (B)	Arena (B)	Latency (μ s)	Held-out F1	Δ F1 (INT8)
4,8	15	345	3,832	2,660	680	0.953	+0.002
4,8	30	345	3,832	2,980	1,232	0.812	-0.001
4,8	60	345	3,832	3,700	2,303	0.895	+0.001
8,16	15	1,265	5,072	4,244	1,237	0.959	+0.000
8,16	30	1,265	5,072	4,884	2,314	0.920	+0.021
8,16	60	1,265	5,072	6,084	4,423	0.865	+0.002
16,32	15	4,833	9,288	7,428	2,376	0.943	+0.009
16,32	30	4,833	9,288	8,708	4,569	0.943	+0.002
16,32	60	4,833	9,288	11,108	8,862	0.868	+0.000
32,64	15	18,881	24,632	23,012	5,040	0.951	+0.005
32,64	30	18,881	24,632	25,572	9,606	0.936	+0.005
32,64	60	18,881	24,632	30,372	18,539	0.950	-0.006

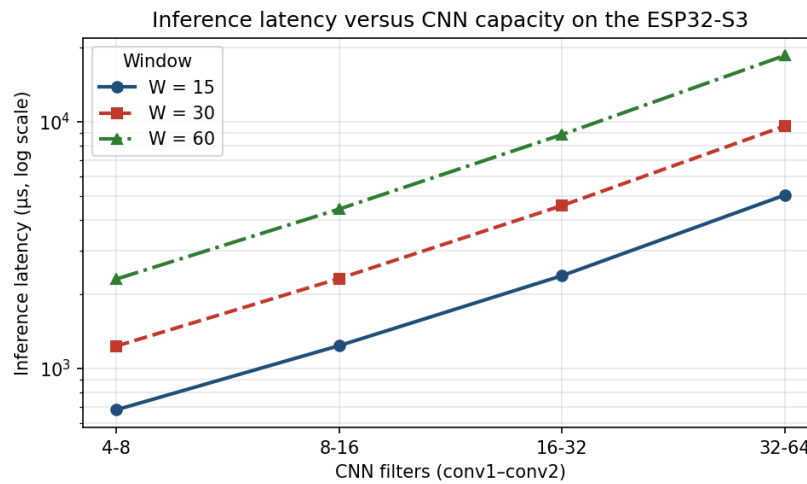


Figure 3. Measured on-device inference latency versus CNN filter capacity

3. RESULTS AND DISCUSSION

3.1. Implementation Results

All twelve configurations were successfully quantized, flashed, and executed on the ESP32-S3. Table 2 reports, for each configuration, the parameter count, flash footprint, on-device tensor arena, measured inference latency, held-out INT8 F1, and the float-to-INT8 F1 change. These measurements show that the leakage-controlled training pipeline produces deployable models and that TensorFlow Lite Micro executes all twelve configurations on the device without modification. The compact models reach a held-out F1 near 0.95, the tensor arena stays between 2,660 and 30,372 bytes across the full grid, and inference completes in 0.68 to 18.5 milliseconds. Every configuration allocated memory correctly and produced valid output.

3.2. Resource Usage on the Device

The models occupy a negligible share of the device's resources. The tensor arena ranges from 2,660 bytes for CNN(4,8) at W=15 to 30,372 bytes for CNN(32,64) at W=60, against the chip's 512 KB of internal SRAM; the largest configuration therefore consumes under six percent of available memory. More than 320 KB of heap remained free after model initialization in every case, and the worst-case task stack high-water mark was 1,608 bytes. This headroom is sufficient to simultaneously run a BLE stack, sensor drivers, FreeRTOS, and a modest application layer on the same chip. The tensor arena does not grow with window length for a given filter configuration: Global Average Pooling collapses the temporal dimension before the dense layer, so arena size is determined by filter count alone, not by W.

This property makes GAP-based CNNs substantially more memory-efficient than recurrent alternatives such as LSTM.

3.3. Inference Latency

Inference latency is reported for every configuration in Table 2 and plotted against filter capacity in Figure 3. It rises from 680 microseconds for CNN(4,8) at W=15 to 18,539 microseconds for CNN(32,64) at W=60, a span of approximately $27\times$. The increase is steep because convolutional layer cost scales with both filter count and input length, and the two factors compound: doubling the filter counts while quadrupling the window length raises the multiply-accumulate (MAC) count by roughly $8\times$ in the first convolutional block alone, before accounting for the second block. Despite this, even the heaviest model consumes only 0.031 percent of a sixty-second sensor interval. The ESP32-S3 can therefore perform occupancy inference continuously at the sensor's native cadence while leaving more than 99.97 percent of each cycle available for sensor polling, wireless communication, and other application tasks. Latency would become a binding design factor only at model sizes or sampling rates far beyond those evaluated here.

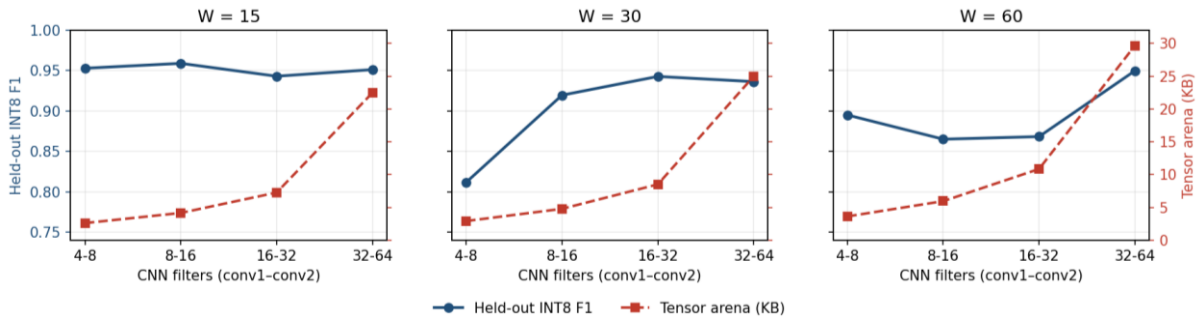


Figure 4. Held-out INT8 F1 (left axis) and on-device tensor arena (right axis) versus CNN filter size

Table 3. Held-out F1 across five random seeds for the two smallest capacities

Model	W	Mean F1	Std. dev.
CNN(4,8)	15	0.916	0.048
CNN(4,8)	30	0.914	0.034
CNN(4,8)	60	0.878	0.059
CNN(8,16)	15	0.933	0.059
CNN(8,16)	30	0.942	0.025
CNN(8,16)	60	0.886	0.013

3.4. Accuracy and the Effect of Window Size

The held-out INT8 F1 sits near 0.94 to 0.96 for the fifteen-minute window across all four filter configurations, and tends to decline at the sixty-minute window, while increasing model capacity does not raise accuracy. Figure 4 plots held-out F1 on the left axis and tensor arena on the right axis, and shows this flat accuracy profile against monotonically rising memory cost, which directly motivates the recommendation of the smallest sufficient model. Two observations are necessary to interpret the per-cell figures correctly.

First, the individual figures carry training variance. Retraining the two smallest filter configurations across five independent random seeds produced the mean F1 scores and standard deviations listed in Table 3. Standard deviations reach approximately 0.06 for CNN(4,8), and the single anomalous result in Table 2, CNN(4,8) at W=30 with F1=0.812, shifts to a mean of 0.914 across seeds. This indicates that the low value is within run-to-run noise rather than a stable effect of the window length. The seed study was restricted to the two smallest configurations because they are the deployment candidates of practical interest and because they exhibit the largest single-run spread; the larger-capacity models are more expensive to retrain and are not recommended given that their accuracy does not exceed the compact baseline, so a variance qualification on the small models is sufficient.

Second, the one pattern that survives training variance is a consistent, modest decline at the sixty-minute window relative to fifteen minutes. A plausible mechanism is the behavior of Global Average Pooling under long sequences: GAP averages activations across all W time steps, and at $W=60$ this averaging dilutes the short, recent transitions that most reliably signal a change in occupancy state. A longer window also yields fewer and more strongly auto-correlated training samples from a fixed-length recording, reducing effective training set diversity. Together these effects give the fifteen-minute window a consistent accuracy advantage. A controlled ablation that separates temporal dilution from reduced effective sample count would further clarify this mechanism and is a natural extension of the work.

3.5. INT8 Quantization

INT8 quantization is effectively lossless for this model class. Across all twelve configurations the held-out F1 changes by between -0.006 and $+0.021$ relative to the corresponding floating-point model evaluated with the same integer arithmetic the device applies. This near-lossless behavior arises because the convolutional filters here are small enough that the representational redundancy of a 32-filter bank averages out individual rounding errors, while the compact (4,8) configuration loses at most half a percent because each of its four filters must encode a more distinct pattern and is therefore more sensitive to per-weight rounding. The memory and latency reductions of INT8 inference therefore arrive at no meaningful accuracy cost, which is the key property that makes on-device deployment practical. This outcome is consistent with post-training quantization penalties reported for convolutional networks in the general literature [22],[23].

Table 4. Deployment ranking of the short-window ($W = 15$) configurations

Model ($W = 15$)	Held-out F1	Arena (B)	Latency (μ s)
CNN(4,8)	0.953	2,660	680
CNN(8,16)	0.959	4,244	1,237
CNN(16,32)	0.943	7,428	2,376
CNN(32,64)	0.951	23,012	5,040

Table 5. Confusion matrix of the recommended model.

	Predicted: Occupied	Predicted: Empty
Actual: Occupied	TP = 642	FN = 18
Actual: Empty	FP = 37	TN = 2303

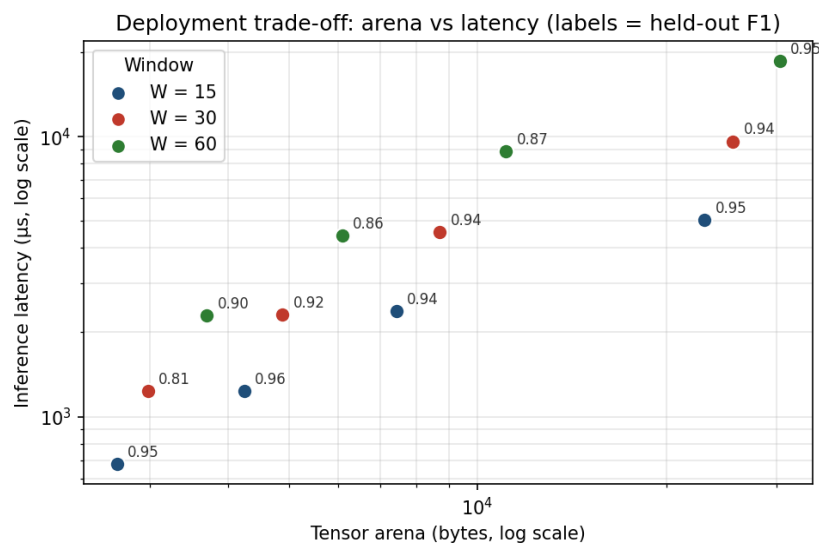


Figure 5. Tensor arena versus inference latency for all twelve configurations

3.6. Deployment Trade-offs and Recommendation

For deployment decisions, Table 4 ranks the four short-window ($W=15$) configurations by held-out F1 against their on-device arena and latency, and Figure 5 plots arena against latency for the full twelve-configuration grid with F1 annotated. CNN(4,8) and CNN(8,16) are statistically tied in accuracy at 0.953 and 0.959 respectively, so the choice between them depends on the relative importance of cost and recall margin. CNN(8,16) spends 4,244 bytes of arena and 1,237 microseconds to achieve a slightly higher recall, as the confusion matrix in Table 5 shows: on the 3,000-window device evaluation subsample it produces 37 false positives and 18 false negatives, giving a precision of 0.946 and a recall of 0.973. CNN(4,8) achieves nearly identical aggregate accuracy in 2,660 bytes and 680 microseconds. CNN(8,16) is the recommendation when a small recall margin matters and memory is available; CNN(4,8) is the recommendation when leaving the largest possible share of the chip free is the priority. The two larger configurations, CNN(16,32) and CNN(32,64), consume substantially more arena and latency without any accuracy benefit, making them suboptimal for this task under any cost constraint.

The latency figures become meaningful when set against the sensing cadence. With one sensor reading per minute, each inference has a sixty-second budget of sixty million microseconds. The recommended CNN(8,16) configuration consumes 1,237 microseconds of this budget, approximately 0.002 percent, leaving the processor idle for more than 99.99 percent of every cycle. Even the largest configuration, CNN(32,64) at $W=60$ with 18,539 microseconds, uses only 0.031 percent of the interval. A single node could therefore evaluate several models sequentially, maintain a BLE or Wi-Fi stack, poll multiple sensors, or run additional on-device analytics within each cycle without any timing conflict.

The memory picture supports the same conclusion. The recommended model requires 4,244 bytes of tensor arena, under one percent of the 512 KB of internal SRAM, and more than 320 KB of heap remained free after initialization in every configuration tested. These margins confirm that the ESP32-S3 can function as an autonomous sensing-and-decision node, rather than a data relay, with substantial capacity remaining for wireless communication stacks, sensor drivers, and application logic. The implementation therefore satisfies the design goal stated in the introduction: occupancy inference on the device is practical, the hardware is not the bottleneck, and the node can act locally without any cloud dependency.

4. CONCLUSION

This paper demonstrated a complete, working implementation of quantized CNN occupancy detection running entirely on the ESP32-S3, without any cloud dependency. All twelve configurations were trained under a leak-free protocol, quantized to INT8, deployed through TensorFlow Lite Micro, and executed on the device. The recommended configuration, CNN(8,16) at a fifteen-minute window, reaches a held-out F1 of 0.959 using 4.2 KB of tensor arena and 1.24 ms per inference, with over 320 KB of heap free, and INT8 quantization is essentially lossless. On-device CNN occupancy detection is therefore practical on a low-cost microcontroller, and a small network is sufficient. The time and memory it leaves unused, more than 99.99 percent of each sensing cycle and over 320 KB of heap, indicate that the chip can serve as an autonomous sensing-and-decision node rather than a data relay, with room for further on-device tasks.

The study has limits that point to the next steps. The benchmark is a single office room, so the accuracy obtained may not transfer directly to harder multi-room or noisier settings. Accuracy was measured on an evenly spaced subsample of the held-out file and validated against the full offline integer evaluation, and the smallest models were repeated across seeds while the others were trained once. Power and energy were not measured. Extending the implementation to live streaming sensors, to multi-occupant and multi-room occupancy [21], and to a measured energy budget under a full application workload would build on the working system demonstrated here.

REFERENCES

- [1] L. M. Candanedo and V. Feldheim, "Accurate occupancy detection of an office room from light, temperature, humidity and CO₂ measurements using statistical learning models," *Energy Build.*, vol. 112, pp. 28–39, 2016. doi: 10.1016/j.enbuild.2015.11.071

- [2] P. Chaudhari, Y. Xiao, M. M.-C. Cheng, and T. Li, "Fundamentals, algorithms, and technologies of occupancy detection for smart buildings using IoT sensors," *Sensors*, vol. 24, no. 7, p. 2123, 2024. doi: 10.3390/s24072123
- [3] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, 2016. doi: 10.1109/JIOT.2016.2579198
- [4] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, 2016. doi: 10.1109/MC.2016.145
- [5] S. Deng, H. Zhao, W. Fang, J. Yin, S. Dustdar, and A. Y. Zomaya, "Edge intelligence: The confluence of edge computing and artificial intelligence," *IEEE Internet Things J.*, vol. 7, no. 8, pp. 7457–7469, 2020. doi: 10.1109/JIOT.2020.2984887
- [6] H. Choi, C. Y. Um, K. Kang, H. Kim, and T. Kim, "Application of vision-based occupancy counting method using deep learning and performance analysis," *Energy Build.*, vol. 252, art. no. 111389, 2021. doi: 10.1016/j.enbuild.2021.111389
- [7] P. Warden and D. Situnayake, *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. Sebastopol, CA: O'Reilly Media, 2019.
- [8] R. Sanchez-Iborra and A. F. Skarmeta, "TinyML-enabled frugal smart objects: Challenges and opportunities," *IEEE Circuits Syst. Mag.*, vol. 20, no. 3, pp. 4–18, 2020. doi: 10.1109/MCAS.2020.3005467
- [9] P. P. Ray, "A review on TinyML: State-of-the-art and prospects," *J. King Saud Univ. Comput. Inf. Sci.*, vol. 34, no. 4, pp. 1595–1623, 2022. doi: 10.1016/j.jksuci.2021.11.019
- [10] A. Elhanashi, P. Dini, S. Saponara, and Q. Zheng, "Advancements in TinyML: Applications, limitations, and impact on IoT devices," *Electronics*, vol. 13, no. 17, p. 3562, 2024. doi: 10.3390/electronics13173562
- [11] N. N. Alajlan and D. M. Ibrahim, "TinyML: Enabling of inference deep learning models on ultra-low-power IoT edge devices for AI applications," *Micromachines*, vol. 13, no. 6, p. 851, 2022. doi: 10.3390/mi13060851
- [12] Y. Abadade, A. Temouden, H. Bamoumen, N. Benamar, Y. Chtouki, and A. S. Hafid, "A comprehensive survey on TinyML," *IEEE Access*, vol. 11, pp. 96892–96922, 2023. doi: 10.1109/ACCESS.2023.3294111
- [13] Espressif Systems, *ESP32-S3 Technical Reference Manual*, ver. 0.4, 2023.
- [14] R. David et al., "TensorFlow Lite Micro: Embedded machine learning on TinyML systems," in *Proc. 4th MLSys Conf.*, pp. 800–811, 2021.
- [15] C. Banbury et al., "MLPerf Tiny benchmark," in *Proc. NeurIPS 2021 Track Datasets Benchmarks*, 2021.
- [16] A. Osman, U. Abid, L. Gemma, M. Perotto, and D. Brunelli, "TinyML platforms benchmarking," in *ApplePies 2021*, LNEE vol. 866. Cham: Springer, 2022. doi: 10.1007/978-3-030-95498-7_20
- [17] I. Rodriguez-Conde, C. Campos, and F. Fdez-Riverola, "Horizontally distributed inference of deep neural networks for AI-enabled IoT," *Sensors*, vol. 23, no. 4, p. 1911, 2023. doi: 10.3390/s23041911
- [18] V. Tsoukas, A. Gkogkidis, E. Boumpa, and A. Kakarountas, "A review on the emerging technology of TinyML," *ACM Comput. Surv.*, vol. 56, no. 10, pp. 1–37, 2024. doi: 10.1145/3661820
- [19] Z. Wang, W. Yan, and T. Oates, "Time series classification from scratch with deep neural networks: A strong baseline," in *Proc. IJCNN*, 2017, pp. 1578–1585. doi: 10.1109/IJCNN.2017.7966039

- [20] B. Jacob et al., “Quantization and training of neural networks for efficient integer-arithmetic-only inference,” in Proc. IEEE/CVF CVPR, 2018, pp. 2704–2713. doi: 10.1109/CVPR.2018.00286
- [21] D. Fährmann, F. Boutros, P. Kubon, and N. Damer, “Ubiquitous multi-occupant detection in smart environments,” Neural Comput. Appl., vol. 36, no. 6, pp. 2941–2960, 2024. doi: 10.1007/s00521-023-09162-z
- [22] R. Banner, Y. Nahshan, and D. Soudry, “Post training 4-bit quantization of convolutional networks for rapid-deployment,” in Proc. NeurIPS, 2019.
- [23] K. Zhao et al., “Distribution adaptive INT8 quantization for training CNNs,” in Proc. AAAI, vol. 35, no. 4, pp. 3483–3491, 2021. doi: 10.1609/aaai.v35i4.16462
- [24] J. A. Armenta-Garcia, F. F. Gonzalez-Navarro, J. Caro-Gutierrez, and C. I. Garcia-Reyes, “Tools and Methods for Achieving Wi-Fi Sensing in Embedded Devices,” Sensors, vol. 25, no. 19, art. no. 6220, 2025. doi: 10.3390/s25196220