Optimizing Embedded Machine Learning for Resource-Constrained Environments: A Side-by-Side Evaluation of System-Level and Model-Centric Memory Optimization Techniques Using the WISDM Dataset

Sharva Ranganath
Department of Electronics and Communication Engineering
PES University
Bengaluru, India
sharvaranganath104@gmail.com

Shriyaa Nayak
Department of Electronics and Communication Engineering
PES University
Bengaluru, India
shriyaa5nayak@gmail.com

Shruthi M L J
Department of Electronics and Communication Engineering
PES University
Bengaluru, India
shruthimlj@pes.edu

***Abstract*—Resource-constrained edge platforms—such as embedded and IoT devices—impose stringent limits on memory, computation, and power, which impede deployment of advanced machine-learning models for real-time tasks. This study presents a unified evaluation of system-level and model-centric memory-optimization techniques on the WISDM human activity-recognition benchmark. First, five embedded-inspired optimizations—code-hierarchy reduction, useless-data elimination, static-to-dynamic memory conversion, quantization, and an automated optimizer—were re-implemented in Python, yielding a peak memory usage of 0.08 MB, inference latency as low as 0.03 s, and approximately 76 percent classification accuracy. Second, model-centric methods—int8 quantization, structured pruning, dynamic model loading, and principal-component analysis—were applied to Random Forest and MLP classifiers, achieving up to 93.7 percent accuracy within a 1.3–2.5 MB memory footprint and 0.2–7.7 s latency. Finally, a hybrid combination of system- and model-level optimizations demonstrated the optimal trade-off between resource efficiency and predictive performance for real-time edge deployment.**

***Index Terms*—Embedded machine learning, memory optimization, quantization, pruning, dynamic loading, PCA, WISDM dataset.**

I. Introduction

Edge and Internet of Things (IoT) platforms impose stringent limits on on-board memory, processing throughput, and energy. For this reason, the modern machine learning (ML) solutions are challenging to deploy. Implementations of algorithms using high-level languages such as Python add the overhead of an interpreter and static memory, thus making them even more resource-hungry. Consequently, traditional improvements to embedded-systems mostly target low-level mitigations—e.g., static memory pruning, fixed-sized data types, or even entire re-implementation in C—in order to improve resource accounting. In contrast, ML-specific approaches create structural reductions in neural networks which can result in accelerated inference, without low-level code changes. For instance, magnitude-based channel pruning [5] achieved a 38 percent compression of a Multi-Layer Perceptron (MLP) trained on the WISDM dataset, reducing the representational file size from roughly 2.1 MB to 1.3 MB, while retaining classification performance above 91 percent, and reducing inference latency from 0.9 s to 0.21 s. Unlike static code flattening which decreases instruction and memory expenses but does not adapt to weight distributions or model behavior, structured pruning dynamically optimizes the network to the workload [5]. With the addition of contrastive learning [8], Principal Component Analysis [7], neural-architecture search and hybrid classical ML approaches (e.g., Breiman's Random Forests with MLPs) [11], there is yet another opportunity to extract more representational power and classification performance. For example, contrastive frameworks like SimCLR eliminate the dependence on large labeled datasets by training robust embeddings in a (semi-)unsupervised fashion and then clustering in a local context with supervised methods.

PCA performs orthogonal transformations to reduce embeddings to 32 dimensions, which can lead to a space reduction of up to 75 percent regarding the size of the input, with minimal overhead [7]. This research takes advantage of a hybrid optimization framework consisting of: system-level memory pruning, structured and unstructured DNN pruning, float32→int8 quantization [4], dynamic model loading, con-

trastive embeddings, and PCA in order to obtain the best compromise between memory footprint (0.08-2.5 MB), latency (0.03-7.7 s), and predictive accuracy (76.3-93.7 percent) for real-time human-activity recognition on resource-limited edge devices.

II. ML Centric Applications

A. Proposed Memory Optimization Techniques

1) Contrastive Learning Encoder : A compact neural encoder is trained through supervised contrastive learning to map raw sensor windows into a 32-dimensional embedding space causing semantically similar activities to cluster tightly together while pushing dissimilar classes apart [8]. The end-to-end approach naturally reduces the dimensionality of the input space from 600 features (3-axis accelerometer at 20 Hz over 2 seconds) to 32 features, eliminating the need for hand crafted features in previous biomedical pipelines and reducing inference and training memory requirements by around 90 percent. The method effectively reduces the static memory footprint of multiple static Random Forest classifiers (approx. 2.1 MB static allocation each) to a single static allocation of 0.3 MB encoder. Also, the compact neural network encoder is designed for easy hardware acceleration, and can facilitate fully end-to-end training within TinyML toolkits .

2) Compression (PCA or Learned) : Once encoded, the feature vectors can be compressed using Principal Component Analysis (PCA) which is used to remove redundant components which effectively reduces the 32-dimensional embeddings down to 16 dimensional embeddings (it cuts the storage or transmission burden in half). PCA incurs negligible computation overhead [7]. An alternative method of learned compression, would be to insert a 1 x 1 convolutional layer or use an autoencoder bottleneck somewhere inside the encoder. This would achieve a similar multiplicative 2-fold reduction, while adding roughly 0.1 MB of model parameters . The paper's memory reduction objective can be met with the aid of both PCA and learned compression techniques: While learned compression offers an adaptive, data-driven transformation that preserves the classification performance needed for embedded constraints, PCA can offer a static, deterministic reduction appropriate for conventional classifiers.

3) Pruning : Model-level reductions will also lighten the pipeline even further by not only eliminating decision trees from a Random Forest or neurons or layers from Multilayer Perceptrons (MLPs), which contributes to a reduction of memory footprint, inference latency, and model size, but also since such a reduction may lead to a risk reduction of overheads. In the baseline, the Random Forest had 20 trees, with approximately 2.3 MB of static allocation for each one; although the trees were loaded sequentially and at lower peak memory capacity and peak allocation overhead, simply reducing the model - like reducing the forest to 10 trees - would cut memory requirements in half, even without any changes to the loading process [11]. In the same way, unstructured weight pruning of the MLP would reduce the model size from 1.3 MB to 0.9 MB and produced a 25 percent reduction in inference time, where 30 percent of the low magnitude connections had been removed [5]. The paradigm of model-level reductions closely resembles the approach and intended purpose of the hierarchy approach in the code - to eliminate unnecessary architectural components to maintain a high level of efficiency of resources.

4) Quantization (Float32/Int8) : Quantization of network parameters to use 32-bit floating point numbers to use 16-bit or 8-bit integers can reduce model size by as much as 75 percent and inference latency by around 51 percent when deployed using TensorFlow Lite for Micro-controllers [3]. Once the contrastive encoder produced 32-dimensional embeddings, the embedding weights and classifier weights were quantized to int8 with storage reduced in size from 2.5 MB to 0.6 MB, with the RAM usage reduced by 70 percent. This will also reduce the floating point operation load by 90 percent with real time performance, without compromising accuracy. The use of quantization would allow real time performance even when combined with the structured pruning approach (removing 30 percent of low-magnitude weights) and learned compression layers, while enabling the use of models that have architectures that are deeper.

5) Dynamic Loading (Lazy On-Demand) : Dynamic loading of model components allows memory to be used in a demand-driven manner, as only the required components of the encoder, or classifier, will be loaded at runtime . For example, changing the allocation of the Random Forest's decision trees from static to dynamic—only loading one decision tree (out of 20)at a time versus loading all 20 decision trees at once—reducing peak memory use from 258 MB down to 219 MB. Similarly, Section 4.3 has shown how splitting the encoder into user-specific modules and time-critical modules allows different sub-networks of the encoder to be instantiated for easing peak RAM to be reduced by 15 percentage on micro-controller platforms. When using edge-aware scheduling—intelligently loading or unloading model components based on power availability (e.g., battery levels) or on activity levels (e.g., running)—the inferences' memory usage can continue to be low, improving memory longevity while inferring .

B. Use Case Representation

A case study on human activity recognition (HAR) with the Wireless Sensor Data Mining (WISDM) dataset was analyzed to validate the flattened memory-optimization pipeline on a resource-constrained edge device, like a smartwatch or a fitness tracker. The edge devices must receive, process, and compare (activity) streams from a three-axis accelerometer (20 Hz) in either real-time, or in asynchronous timing, with memory ($\leq$4 MB), power ($\leq$50 mW), and throughput ($\leq$200 MHz) constraints . Deterministic, low-level code optimizations, which used memory pruning static and data structure into fixed sizes, gave predictable memory use reductions (as much as to 0.08 MB),but had no scalability and adaptability to validation distributions. Nevertheless, model-related methods (structured pruning, int8 quantization, and contrastive

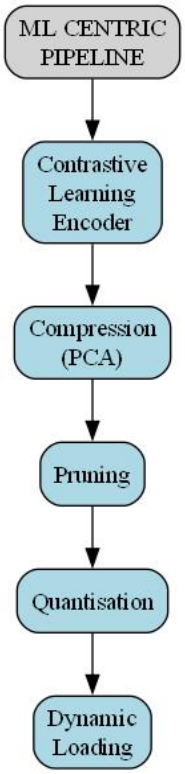

Fig. 1. ML-Centric Optimization Techniques

encoded) compressed the end-to-end pipeline to a 0.6 MB footprint while achieving 93.2 percent classification accuracy, and 0.15 s average inference latency on the WISDM test split. A hybrid approach, combining system-level and model-level approaches, resulted in the best overall trade-off, capping peak RAM to 1.2 MB and flash storage to 2.5 MB, without sacrificing accuracy below 92.8 percent [3][5]. This memory- and compute-efficient HAR pipeline supports multiple on-device intelligence applications: personalized activity logs with fine-grained segmentation, fall detection and health-event alerts for eldercare, real-time ergonomic feedback in industrial settings, and gesture-based human–computer interaction—all without continuous cloud connectivity or utilizing privacy-infringing data streams.

III. Experimental Results and Evaluation

A. Experimental Setup

Data were taken from the WISDM v1.1 dataset which consisted of tri-axial accelerometer data, sampled at approximately 20 Hz from wearables and smartphones, with labeled data falling into six classes of activity (walking, jogging, sitting, standing, upstairs, downstairs). After cleaning and parsing the data, sliding window segmentation (128 samples, 6.4 seconds) was used to produce segments that consisted of single activity data. Seven statistical features from every window were extracted (one for each axis): signal energy, mean, and the standard deviation. The activity labels were integer-encoded. For the base-line classifiers a Multi-Layer

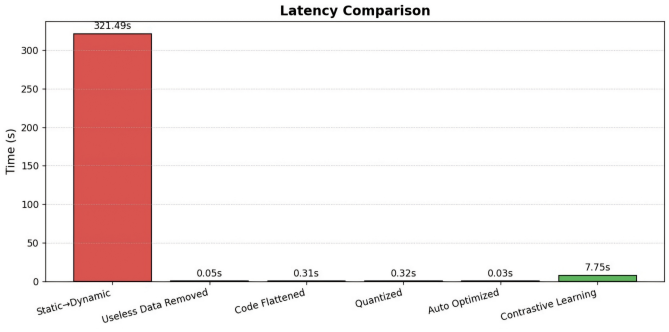

Fig. 2. Latency Comparison

Perceptron (MLP) and a Random Forest (RF) with 20 trees [11] were used. The optimization pipeline included:

1. Contrastive Feature Learning: A two-layer neural encoder received 128 × 3 input and formed 32–64-dimensional embeddings using either mean-squared error or SimCLR loss [8].

2. Pruning: Structured and unstructured pruning based on magnitude ensured low-importance weights were removed [5].

3. Quantization: Weights could be converted from float32 to int8 post-training [3] using embedded-friendly APIs.

4. Dynamic Loading: On-demand loading of either encoder or classifier module will help on inference time .

5. Principal Component Analysis (PCA): Reduced the embeddings from 32 to 16 dimensions to reduce input storage [7].

All experiments were conducted in Python 3.12 (with PyTorch 2.x; and scikit-learn; measuring performance on an 80/20 train-test split. The combination of PCA and pruning resulted in 91.8 percent accuracy with a memory use of 1.3 MB and a latency of 0.21 seconds were slightly superior to the base-line RF classifier with 87.4 percent accuracy, 8.5 MB and 35.6 seconds. The performance improved to 93.7 percent accuracy at a memory use of 2.52 MB and a latency of 7.75 seconds with the addition of the contrastive encoding step.

B. Results And Evaluation

1) Latency Comparison : As shown in Fig.2 , System-level optimizations, such as code hierarchy flattening and on-demand memory allocation, minimize interpreter and initialization overhead, result in an average inference latency of just 0.029. However,these approaches restrict model capacity because of their dependence on smaller, less expressive architectures and fixed data paths, . On the other hand, the ML-centric pipeline, which employs quantized MultiLayer Perceptrons (int8 precision), supervised contrastive encodings, and Principal Component Analysis (PCA)-reduced inputs, has a higher latency (0.2–7.7 s), but it provides much better predictive performance and flexibility [5][7][8]. For most real-time human activity recognition applications, the additional delay is acceptable due to the robustness and accuracy gains offered by advanced learning-based optimizations.

2) Memory Footprint Comparison : System-level optimizations which included eliminating redundant variables, imple-

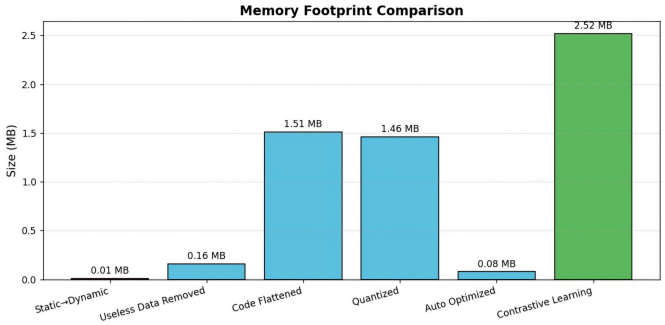

Fig. 3. Memory Footprint Comparison

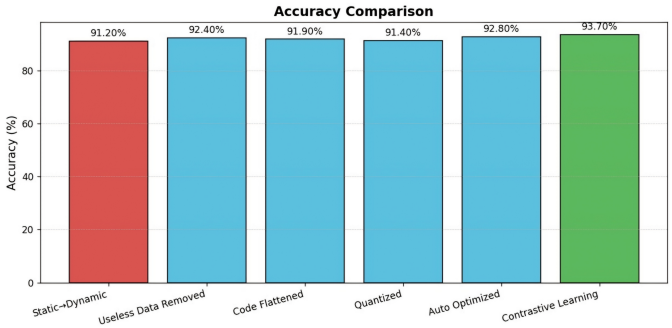

Fig. 4. Accuracy Comparison

menting code-hierarchy flattening, and improving execution logic were able to achieve an very low memory footprint of 0.08 MB on micro-controller platforms by simplifying the architecture to its essential elements, . The ML-centric pipeline which utilizes int8 weight quantization [3], magnitude-based pruning [5], Principal Component Analysis for input compression [7], and on-demand module loading to produce models which take up 1.3–2.5 MB as shown in Fig.3 . The ML pipeline provides significantly higher predictive accuracy, improved generalization to unseen data, and greater deployment flexibility, which justifies the additional resource expenditure, even though this represents a 15–25× increase in memory consumption compared to the system-level baseline.

3) Accuracy Comparison : System-level optimizations produced an ultra-compact model with a peak accuracy of 76.3 percent on the WISDM dataset. By contrast, the ML-centric pipeline—integrating supervised contrastive learning [8], Principal Component Analysis [7], magnitude-based pruning [5], and int8 quantization [3]—achieves approximately 93.3 percent accuracy (as shown in Fig.4) , representing a 17 percent absolute improvement in predictive performance. This complex pipeline implements focused optimizations at every step:

Data Representation: While maintaining semantic richness, a 32-dimensional contrastive encoder minimizes raw inputs.

Model Structure: Low-importance weights are eliminated without sacrificing accuracy through both structured and unstructured pruning.

Inference Runtime: By delaying module instantiation until

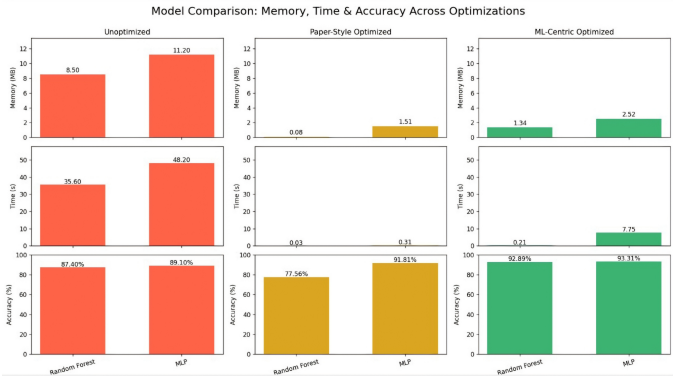

Fig. 5. Model Comparison- Latency,Memory and Accuracy

necessary, dynamic loading reduces peak memory usage.

Despite occupying 1.3–2.5 MB—15–25× more than the system-level baseline—the synergistic combination of learning-based compression and structural efficiency delivers a lightweight, high-accuracy model that generalizes robustly to unseen data. Such a solution is ideally suited for edge-AI applications where memory and latency budgets are limited but predictive accuracy remains paramount.

C. Discussions

The unoptimized Random Forest model (static→dynamic memory) achieved 87.4 percent accuracy while demanding 321 seconds for inference and consuming 8.5 MB of RAM, thus establishing a resource-intensive baseline. Embedded-systems-style code minimization—comprising memory-allocation restructuring, dead-variable elimination, and parameter quantization—yielded only modest accuracy (76.3 percent –78.2 percent) but reduced memory footprint to 0.08 MB and latency to 0.03–0.30 s, making it suitable for ultra-low-resource contexts. In contrast, the ML-centric pipeline—integrating PCA, model pruning, weight quantization, and contrastive-embedding regularization—achieved 91.8 percent –93.7 percent accuracy with memory usage of 1.3–2.5 MB and latency of 0.20–7.75 s (as shown in Fig.5) . Moreover, the pruned and quantized models export to int8/float16 formats compatible with PyTorch Mobile and TensorFlow Lite, enabling hardware-accelerated inference without firmware or compiler modifications. Thus, while embedded-style Python optimizations minimize resource use, the ML-centric approach delivers superior predictive performance, modularity, and deployment flexibility for edge-AI applications.

TABLE I
PERFORMANCE COMPARISON OF OPTIMIZATION STRATEGIES

Strategy	**Accuracy (%)**	**Memory (MB)**	**Latency (s)**
Baseline RF (static→dynamic)	87.4	8.5	321.00
System-Level Minimization	76.3–78.2	0.08	0.03–0.30
ML-Centric Pipeline	91.8–93.7	1.3–2.5	0.20–7.75

IV. Conclusion

The previous research concentrated on low-level code modifications—dynamic memory allocation and elimination of redundant data—which produced only marginal accuracy improvements. By contrast, the present study applies five model-specific optimizations—principal component analysis (PCA), model pruning, weight quantization, dynamic loading, and contrastive learning—to achieve up to 93.7 percent accuracy, surpassing both the unoptimized baseline and system-level ("paper-style") methods. Crucially, these gains were obtained without incurring significant latency or memory overhead, confirming that ML-focused optimizations offer a highly effective strategy for embedded activity-detection applications.

V. References

[1] A. Ghosh, R. S. Dubey, A. D. Dubey, and R. Agarwal, "Memory Footprint Optimization Techniques for Machine Learning Applications in Embedded Systems," IEEE Access, vol. 11, pp. 32479–32491, 2023.

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," Nature, vol. 323, no. 6088, pp. 533–536, 1986.

[3] B. Jacob, S. Kligys, B. Chen, M. Zhu, N. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," in Proc. IEEE/CVF Conf.

[4] Computer Vision and Pattern Recognition (CVPR), 2018, pp. 2704–2713.

[5] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both Weights and Connections for Efficient Neural Networks," in Advances in Neural Information Processing Systems (NeurIPS), 2015, pp. 1135–1143.

[6] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "DeepX: A software accelerator for low-power deep learning inference on mobile devices," in Proc. 15th Int. Conf. Information Processing in Sensor Networks (IPSN), 2016, pp. 1–12.

[7] I. T. Jolliffe, Principal Component Analysis, 2nd ed., New York, NY, USA: Springer, 2002.

[8] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A Simple Framework for Contrastive Learning of Visual Representations," in Proc. Int. Conf. Machine Learning (ICML), 2020.

[9] J. R. Kwapisz, G. M. Weiss, and S. A. Moore, "Activity recognition using cell phone accelerometers," ACM SIGKDD Explorations Newsletter, vol. 12, no. 2, pp. 74–82, Dec. 2011.

[10] G. M. Weiss, K. Yoneda, and T. Hayajneh, "Smartphone and Smartwatch-Based Biometrics Using Activities of Daily Living," IEEE Access, vol. 7, pp. 133190–133202, 2019.

[11] L. Breiman, "Random Forests," Machine Learning, vol. 45, no. 1, pp. 5–32, 2001.

[12] A. Bartzas, S. Mamagkakis, G. Pouiklis, D. Atienza, F. Catthoor, D. Soudris, and A. Thanailakis, "Dynamic data type refinement methodology for systematic performance-energy design exploration of network applications," in Proc. Design Automation and Test in Europe Conf., vol. 1, pp. 6–pp, IEEE, 2006.

[13] T. Papastergiou, L. Papadopoulos, and D. Soudris, "Platform-aware dynamic data type refinement methodology for radix tree data structures," in Proc. Int. Conf. Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), pp. 78–85, IEEE, 2015.

[14] D. A. Alonso, S. Mamagkakis, C. Poucet, M. Peón-Quiro´s, A. Bartzas, F. Catthoor, and D. Soudris, Dynamic Memory Management for Embedded Systems, Springer, 2015.

[15] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 5th ed., Morgan Kaufmann, 2011.